Языки и системы программирования

2 курс 3 семестр

# Логический тип данных

bool

true – истина, false – ложь

!= не равно == равно

int→bool bool→int

0→false false→0

!=0→true true→1

# Арифметические типы данных

## Целочисленный тип(int)

int – по умолчанию signed

Знаковость: signed/unsigned

Размер: short/long

Примеры:

short int s; / short s;

long int s; / long s;

Целочисленные литералы:

208 – десятичный

0207 – восьмеричный

0x208A – шестнадцатеричный

0b01 - двоичный

208 signed int

208L signed long

208U unsigned int

208UL unsigned long

## Типы с плавающей запятой

float – с одинарной точностью(7 знаков после запятой)

double – с двойной точностью(15 знаков после запятой)

long double – с расширенной точностью

Литералы типа с плавающей запятой:

1.25 .13(0.13) 1.2e10 1.23e-15

1.25 – double

1.25f – float

3.05L – long double

# Символьные типы данных

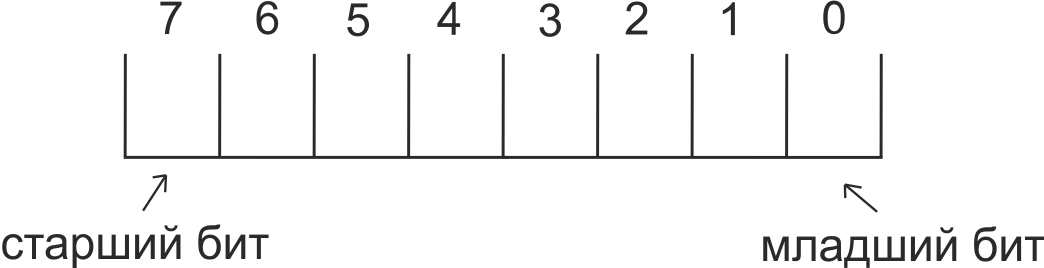
char – обычные символы, почти всегда 8бит

|  |
| --- |
| **char** a=’a’ //хранит код символа ‘a’ ’a’ – символьный литерал |

|  |
| --- |
| std::cout<<”a=” << a<< std::endl<<”код ‘a’ = ” << **int**(a); |

Печатается: a=’a’

код ‘a’ = 97



signed -128..127

unsigned 0..255

unsigned char – байт

wchar\_t – широкие символы

|  |
| --- |
| **wchar\_t** wch=’\x6528’;  **wchar\_t** wch=0x6528; |

(одинаковая запись)

Варианты записи символьных литералов:

‘\n’ – перевод строки ‘\t’ – табуляция

‘\60’ – код символа в 8-ной системе

‘\x60’ – код символа в 16-ной системе

sizeof(char) ≡ 1 < sizeof(short) ≡ 2 ≤ sizeof(int) ≡ 4 ≤ sizeof(long) ≡ 4 или 8

# Размеры фундаментальных типов данных

sizeof – размер переменной в байтах

sizeof(T) ≡ sizeof(signed T) ≡ sizeof(unsigned T)

1 ≤ sizeof(bool) ≤ sizeof(long)

sizeof(char) ≡ 1 < sizeof(short) ≡ 2 ≤ sizeof(int) ≡ 4 ≤ sizeof(long) ≡ 4 или 8

sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)

Непроверенная инфа на всякий случай

bool, char, unsigned char, signed char, \_\_int8 - 1 байт

\_\_int16, short, unsigned short, wchar\_t, \_\_wchar\_t - 2 байта

float, \_\_int32, int, unsigned int, long, unsigned long - 4 байта

double, \_\_int64, long double, long long - 8 байт

\_\_int128 - 16 байт

# Перечисления

|  |
| --- |
| **enum** **keyword**{ASM,AUTO,BREAK}; |

Номера с 0, но можно переопределить:

|  |
| --- |
| **enum** **keyword**; { ASM = 10;  AUTO = 20;  BREAK//21 (+1 к последнему указанному) } |

|  |
| --- |
| **keyword** k=keyword(3) |

Диапазон значений перечислимого типа [0, 2k-1], где k – минимальная степень двойки, для которой все элементы помещаются в диапазон

# Объявления и имена

1. Спецификатор (не обяз.)

2. Базовый тип (обяз.)

3. Декларатор (обяз.)

4. Инициализирующее выражение (не обяз.)

|  |
| --- |
| [<специф.>] <баз. тип> <декл.> [<инициал. Выраж.>] |

|  |
| --- |
| **int** a = 1; **extern**(специф) **double** d; **char**\*(тип) months[ ](декл) = {“jan” , “feb” , …}(инициал); |

1. Имена переменных могут состоять из латин. букв, цифр, символа подчеркивания
2. В качестве имен нельзя использовать ключевые слова (if, case,..)
3. Регистр имеет значение: bool if – нельзя! , bool If – можно

# Инициализация в объявлении

* Любая сущность перед использованием должна быть объявлена
* Объявлений одной и той же сущности может быть **сколько угодно**
* Определение может быть  **только одно**

Объявление говорит, что «есть сущность».

Определение говорит, что эта сущность из себя представляет.

|  |
| --- |
| **int** a; // объявл. и опред. **int** a; // ОШИБКА! **extern** **int** b; // объявл. **extern** **int** b; // нет ошибки, ОК **extern** **double** b; // ОШИБКА, т.к. тип изменен **int** b; |

Можно объявлять несколько переменных:

|  |
| --- |
| **int** a,b = 1, c,d=b; **int** x = x; // Нет ошибки, х принимает случайное значение |

# Понятие области видимости

Область видимости (относ. к люб. объявлению):

1. Глобальное имя – объявление ф-ий, классов или пространств имен

2. Локальное имя – в противном случае

О.в. (1) – файл, в кот. оно объявлено (объявление -> конец файла)

О.в. (2) – блок, в кот. оно объявлено

Блок: { //начало (Тело ф-ии; выдел. пр-во имен, объявл. класса)

… Если Х вне блока, то она глоб.

} // конец Если в блоке, то локал.

Пример:

|  |
| --- |
| **int** x; // глобал. **void** **f**() {  **int** x; //локал. Int x=x; //лок. Х иниц лок. Х  x = 1; //локал. Х == 1 х = ::х; //лок. Х иниц. глоб. х  {  **int** x; // локал. в блоке; она скрывает локал. х  x = 2; // локал. в блоке х == 2; локал х == 1  }  x = 3; // лок. х  ::x = 10; // глоб. х } |

Переменные нужно объявлять, как можно более локально.

# Ключевое слово typedef

Определение, начинающееся с ключевого слова **typedef**, определяет новое имя для типа, а не новую переменную какого-либо существующего типа.

|  |
| --- |
| **typedef** **char**\* Pchar; Pchar p1, p2; // p1 и p2 типа char\* **char**\* p3 = p1; |

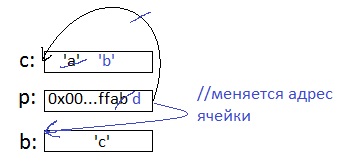
Вводимые т.о. имена являются новыми синонимами для существующих типов, но не новые типы. Их можно использовать параллельно именам существующих типов.

# Тип данных void

1. для функций без возврата результата
2. void\* - указатель на произвольный тип

Нет литералов, невозможно объявить, присвоить значение

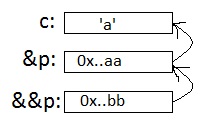
# Указатели, нулевой указатель

Для любого типа Т тип Т\* - указатель на Т

|  |
| --- |
| **char** c = ‘a’; **char** \*p = &c; //&c - взятие адреса \*p = ‘b’; //\*p - разыменование указателя **char** b = ‘c’; p = &b; |

Пример

|  |
| --- |
| **int** \*pi; //pi указывает на целое **char** \*\*pc; //pc указывает на указатель на символ **int** \*ap[15]; //массив из 15 указателей на целое **int** \***f**(**char** \*); //функция, принимающая указатель на символ, возвращающая указатель на целое **int** (\*fp)(**char** \*); //указатель на функцию, принимающую указатель на символ, возвращающая целое |



Нулевой указатель

(NULL //стиль С, в С++ можно int \*p = NULL;

0 //С++ int \*p = 0;)

nullptr //С++11, свой тип nullptr\_t int \*p = nullptr;

При объявлении указателя его необходимо инициализировать. Если нечем - нулевым значением (nullptr).

Пример

|  |
| --- |
| 1)**void** **f**() {  **int** \*p;  \*p = 1; //неопределенное поведение  … } 2) **void** **f**() {  **int** \*p = nullptr;  …  **if** (p != nullptr)  //работаем с p } |

Под глобальные и статические переменные память выделяется при запуске программы и до завершения (какое-то начальное значение типа). Под локальные - в момент прохождения (автоматические). Обязательно инициализировать.

|  |
| --- |
| **void** **f**() {  **int** i; //выделяется память под i  ... } //освобождается память под i |

в этой теме еще new и delete было (хотя это отдельный вопрос), лучше туда заглянуть

Адресная арифметика:

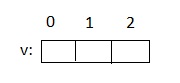
T \*p;

p + 1; указатель на следующий элемент типа T (минус - на предыдущий)

Указатели в рамках одного массива можно вычитать

# Массивы. Инициализация массивов

Массив - это объединение элементов одного типа

T[size] - массив из size элементов типа Т

size - const или константное значение

Пример

|  |
| --- |
| **float** v[3]; **char** \* a[100]; // 100 указателей на символ |

В памяти массив расположен непрерывно. Индексируются элементы с 0

v[0]..v[2]

Многомерные массивы:

int d[10][20]; //массив из 10 массивов по 20 целых чисел



Массивы можно инициализировать (список инициализации):

|  |
| --- |
| **char** v[3] = {‘a’, ‘b’, ‘c’}; **char** v2[3] = {‘a’}; **char** v3[2] = {‘a’, ‘b’, ‘c’}; //ошибкаБезымянный5.jpg |

Если есть список инициализации - количество можно не писать:Безымянный6.jpg

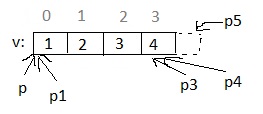
|  |
| --- |
| **char** v4[] = {‘a’, ‘b’, ‘c’, ‘d’}; **int** v5[2][3] = {{1,2,3},{4,5,6}}; //можно записать так:  **int** v5[2][3] = {1,2,3,4,5,6}; |

Присваивание массивов недопустимо:

|  |
| --- |
| v4=v5; //ошибка |

Массивы нельзя сравнивать

Связь указателей и массивов:

Имя массива является константным указателем на его первый элемент

|  |
| --- |
| **int** v[] = {1,2,3,4}; **int** \*p = v; плохая запись того же самого: **int** \*p1 = &v[0]; **int** \*p3 = &v[3]; **int** \*p4 = v+3; **int** \*p5 = &v[4]; //можно получать, но нельзя использовать адрес за последним элементом |

v[i] - i-й элемент

Если i выходит за границу - неопределенное поведение, надо самому за этим следить

(mas[0][3] - из первого массива четвертый элемент)

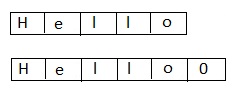
# Строковые литералыБезымянный8.jpg

Строка в С++ - это массив из символов

char \*p = “Hello”; //это строковый литерал

Переменная не хранит длину, поэтому “пока не 0...”

p[1] = ‘a’; //ошибка (достать можно, записывать нельзя)



|  |
| --- |
| **char** p2[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’};// массив из символов **char** p3[] = “Hello”; |

Все строковые литералы в памяти выделяются статически (не меняются во время работы программы)

|  |
| --- |
| **const** **char** \* **get\_error**() {  **return** “memory overflow\n error=...”; } |

“” - пустая строка

|  |
| --- |
| **char** \*mes = “Задача 1\n”  ...  “Автор: …\n”; |

(по сути одна разбитая строка)

# Константы

|  |
| --- |
| **const** **int** MAX\_SIZE = 1024 \* 1024; |

Значение const не меняется в процессе выполнения программы

|  |
| --- |
| MAX\_SIZE = 10; //ошибка компиляции |

const должна быть инициализирована сразу

|  |
| --- |
| **void** **f**(**int** size) {  //...  **const** **int** SIZE = size \* 1024;  //... } |

## Константные массивы

|  |
| --- |
| **const** **int** Fibbonacci[] = {1, 1, 2, 3, 5, 8, ...}; Fibbonaccio[1] = -10; //Ошибка |

## Константные указатели

|  |
| --- |
| **int** a, b; **int** \*p = &a; //можно менять и указатель и значение \*p = 10; //изменение значения p = &b; //изменение указателя  **const** **int** \*cp = &a; //указатель на const cp = &b; //всё ок \*cp = 7; //ошибка. тут мы изменяем константу |

Если требуется запретить изменение адреса, то используется константный указатель

|  |
| --- |
| **int** \***const** pc = &a; //инициализация обязательна |

Значение в этом случае изменять можно

|  |
| --- |
| \*pc = 7; //ОК pc = &b; //ошибка |

Константный указатель на константу

|  |
| --- |
| **const** **int** \***const** cpc = &a; //Тут вообще ничего нельзя менять cpc = &b; //ошибка \*cpc = 7; //ошибка |

## Константные ссылки

|  |
| --- |
| **int** a; **int** &ra = a; //ОК ссылка на a **const** **int** &cra = a; //ссылка на const a = 1; //ок cra = 7; //ошибка **int** &r1 = 1; //ошибка **const** **int** &cr1 = 1; //ок |

При этом последнее раскладывается на

|  |
| --- |
| **const** **int** tmp = 1; **const** **int** &cr1 = tmp; |

# Ссылки

Ссылка – альтернативное имя объекта.

T& – ссылка на T

|  |
| --- |
| **int** a, b; **int** \*pa = &a; //указатель **int** &ra = a; //ссылка (псевдоним переменной) ra = 1; cout << a; //1 |

Ссылку можно задать только при инициализации

Ссылка **должна** быть инициализирована!

|  |
| --- |
| ra = a; //меняется значение pa = &b; //меняется указатель \*pa = 2; //b == 2 (тут идёт разыменование) a = \*pa; //a == 2 cout << ra; //2 |

# Структуры

Структуры – это объединения данных различных типов

|  |
| --- |
| **struct** Addres {  **char** \*name;  **long** number;  **char** \*street;  **char** \*town;  **char** state[2];  **long** zip; }; //точка с запятой обязательна  **void** **f**() {  Addres jd; //достаточно использовать имя типа  jd.name = "John Dorian";  jd.number = 61;  //... } |

Структуры можно инициализировать

|  |
| --- |
| Addres jd = {"John Dorian", 61, "South St.", "New Providence", {'N', 'J'}, 234643}; |

Структуры можно присваивать друг другу

|  |
| --- |
| Adres j2; j2 = jd; |

Структуры нельзя сравнивать

|  |
| --- |
| **if** (jd == j2) //ошибка |

К структуре можно обращаться через адрес

|  |
| --- |
| **void** **print**(Addres \*addr) {  cout << addr->name << "\n" //"->" – доступ к члену структуры  //через указатель  << addr->number << "\n"  << //... };  p->a эквивалентно (\*p).a |

Имя структуры можно использовать сразу

|  |
| --- |
| **struct** Link {  Link \*prev, \*next; //можно, т.к. размер можно вычислить сразу }; |

Но так, как ниже, делать нельзя

|  |
| --- |
| **struct** Bad {  Bad one; //ошибка – рекурс-е определение. Нельзя вычислить размер }; |

Новый тип уникален

|  |
| --- |
| **struct** S1 {  **int** a; }; **struct** S2 {  **int** a; }; |

Выше написаны две разные структуры

|  |
| --- |
| **void** **f**() {  S1 s1 = {1};  S2 s2 = {1};  s2 = s1; //ошибка – разные типы } |

# Выражения. Последовательность вычисления выражений

Ассоциативность

* Левая – все операции, кроме тех, которые правые
* Правая – префиксные, присваивание (первой вычисляется правая часть)

Влияет на следующее

Левая:

|  |
| --- |
| d = a + b + c; d = ((a + b) + c);  3 1 2 |

Правая:

|  |
| --- |
| a = b = c = d = 1; a = (b = (c = (d = 1)));  4 3 2 1 |

|  |
| --- |
| a +++ b (слева направо)   1. (a++) + b – ок 2. a + (++b) |

У любого выражения есть результат

У логических – логический

Если выражение смешанное, то результат будет иметь тип с наибольшей точностью, входящ. в выражение

|  |
| --- |
| **auto** a = 1 + true + 1.0 + 1.0f; //double |

## Последовательность вычислений (!= ассоциативность)

|  |
| --- |
| expr1 + expr2 + expr3 |

Гарантируется, что сложение будет выполнено в следующем порядке:

|  |
| --- |
| (expr1 + expr2) + expr3 |

Не гарантируется последовательность вычисления

|  |
| --- |
| expr1 -> expr2 -> expr3 |

(компилятор может вычислить expr3 раньше других)

|  |
| --- |
| x = f(2) + g(3); |

Т. о. g может быть вызвана раньше, но гарантируется, что присваивание будет позже, чем сумма

Так писать **нельзя**:

|  |
| --- |
| **int** i = 1; v[i] = i++; //v[2] = 1 или v[1] = 1 |

Пример потери точности

|  |
| --- |
| **void** **f**(**unsigned** **int** u) {  **int** i = u; //warning! возможна потеря точности. может стать отрицательным } |

# Приоритет операций

Обозначения:

· name – объявленное имя

· class\_name – имя класса

· member – член класса

· object – объект класса

· pointer – указатель

· expr – выражение

· type – имя типа

· lvalue – леводопустимое выражение

1. Разрешение области видимости **::**

**::name** – доступ к глобальному имени name

**class\_name ::name** – доступ к имени члена класса

2. Выбор члена класса **.**

**object.name** доступ к члену класса

**pointer**  **–> name** доступ к члену класса

**pointer[expr]** – доступ по индексу

**name(expr)** – вызов функции по имени

3. Операции инкремента и декремента

постфиксные

**lvalue++** – инкремент

**lvalue--** – декремент

префиксные

**++lvalue**

**--lvalue**

4.

Размер объекта/типа

**sizeof expr**

**sizeof (type)**

Побитовые операции

**~lvalue** – дополнение

**!lvalue** – отрицание

Унарные +/- , знак числа:

**-expr**

**+expr**

**&lvalue** – адрес

**\*pointer** – разыменование

**new, delete, new[ ], delete[ ]** – работа с динамической памятью

5. Арифметические

**expr \* expr** – умножение

**expr / expr** – деление

**expr % expr** – остаток от деления

6. Арифметические

**expr + expr** – сложение

**expr - expr** – вычитание

7. Побитовые

**expr << expr** – побитовый сдвиг влево (и оп.вставки)

**expr >> expr** – побитовый сдвиг вправо

8. Логические

**expr < expr** – меньше

**expr <= expr** –меньше или равно

**expr > expr** –больше

**expr >= expr** –больше или равно

9. Логические

**expr == expr** – равно

**expr != expr** – неравно

Побитовые

10. **expr & expr** – побитовое И

11. **expr ^ expr** – побитовое исключающее ИЛИ (xor)

12. **expr | expr** – побитовое ИЛИ

Логические

13. **expr && expr** – логическое И

14. **expr || expr** – логическое ИЛИ

15. Условная операция (тернарная)

**expr ? expr1 : expr2** – если **expr** истинно, то **expr1**, иначе **expr2**

16. Присваивания

**lvalue = expr**

**lvalue op= expr**

**op: \*, /, %, +, -, <<, >>, &, |,^.**

17. Генерация исключений

**throw expr**

18. Следование

**expr, expr**

# Арифметические операции

**+expr** — унарный +, знак числа

**-expr** — унарный -, знак числа

**expr \* expr** — умножение

**expr / expr** —деление

**expr % expr** —остаток от деления

**expr + expr** —сложение

**expr - expr** —вычитание

Все бинарные операции можно использовать в «ленивых» присваиваниях  
**lvalue op= expr**

В процессе определения выражения типы с меньшей точностью будут преобразовываться к типам с большей точностью

|  |  |
| --- | --- |
| *Частая ошибка:*  **double** res = 1/2; cout << “res = “ << res; // напечатает 0 | *Как правильно:*  **double** res = 1.0/2.0; cout << “res = “ << res; // напечатает 0.5 |

# Логические операции

**!expr** — отрицание

**expr && expr** — логическое И

**expr || expr** — логическое ИЛИ

**Сравнения — результат логического типа**

**expr < expr** – меньше

**expr <= expr** –меньше или равно

**expr > expr** –больше

**expr >= expr** –больше или равно

**expr == expr** – равно

**expr != expr** – неравно

Нельзя использовать в «ленивых» присваиваниях **lvalue op= expr**

# Побитовые операции

**~lvalue** – дополнение

**!lvalue** – отрицание

**expr << expr** – побитовый сдвиг влево (и оп.вставки)

**expr >> expr** – побитовый сдвиг вправо

\*Побитовый сдвиг невероятно быстр!

**expr & expr** – побитовое И

**expr ^ expr** – побитовое исключающее ИЛИ (xor)

**expr | expr** – побитовое ИЛИ

Побитовый сдвиг подробнее:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | исходн |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | << 2 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | >> 3 |

То есть побитовый сдвиг

**x << y**  эквивалентен **\*2y**

**x >> y** эквивалентен **/2y**

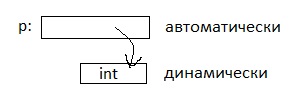
# Операции инкремента и декремента

Операция ++ явно задает инкремент в отличие от неявного его задания с помощью сложения и присваивания. По определению ++lvalue означает lvalue+=1, что, в свою очередь означает lvalue=lvalue+1. Аналогично обозначается операция декремента (--). Операции ++ и -- могут использоваться как префиксные и постфиксные операции. Значением ++x является новое (т. е. увеличенное на 1) значение x. Например, y=++x эквивалентно y=(x+=1). Напротив, значение x++ равно прежнему значению x. Например, y=x++ эквивалентно y=(t=x,x+=1,t), где t - переменная того же типа, что и x.

# Операции работы с памятью new и delete

new - выделяет память, new T - выделяется память под Т

delete - освобождает память



|  |
| --- |
| **void** **f**() {  **int** \*p = **new** **int**;  …  **delete** p; //против утечки памяти } |

Без delete p удаляется, а динамически созданная - нет, и теряется

|  |
| --- |
| **new**[], **delete**[] - массивы  **int** \*mas = **new** **int**[100]; // 100 int … **delete**[] mas; |

# Операции приведения типов

1. Неявное приведение (компилятор)

2. Явное приведение (программист)

Операции явного приведения типов (приводят expr к типу Т):

* Статические (выполняются на этапе компиляции):

o static\_cast <T> (expr) – преобразует выражения одного статического типа в объекты и значения другого статического типа.

o const\_cast <T> (expr) – снятие константности

o reinterpret\_cast <T> (expr) – Приведение типов без проверки. Кaк бы говорит компилятору: «Я знаю, что тебе это не понравится, и все-таки я это сделаю».

* Динамическая (на этапе выполнения):

o dynamic\_cast <T> (expr) – работает либо с указателями, либо со ссылками и позволяет делать понижающие преобразования в иерархии классов

Другой способ:

T (expr) - конструирование

*(T) expr – Не использовать! (стиль Си)*

T () – значение типа Т по умолчанию

Int a = int(); // a=0;

# Условные операторы if, if else, switch

|  |
| --- |
| **if** (условие)  оператор |

или

|  |
| --- |
| **if** (условие)  оператор1 **else**   оператор2 |

пример:

|  |
| --- |
| **void** **f**(**int** \*p)  {  **if** (p) //p! = nullptr  {  //работа с \*p  }  } |

Логические операции, которые могут быть использованы в условии:

<, >, =, !=, <=, >=, &&, ||, !

пример:

|  |
| --- |
| **void** **f**(T \*p) {  **if** (p && p->count > 0)  {  //используем p  } } |

A && B // если А – ложь, то В не проверяется

A || B // если А – истина, то В не проверяется

If можно объединять в цепочки:

|  |
| --- |
| **if** (a > 10)  оператор1 **else** **if** (a < 0)  оператор2 **else** **if** (a == 5)  оператор3 **else**  оператор4 |

## Switch

|  |
| --- |
| **switch** ( выражение) {  **case** значение\_1: оператор\_1;  **case** значение\_2: оператор\_2;  // break - выход из switch  ...  **case** значение\_N: оператор\_N;  **default**: оператор\_0; } |

|  |
| --- |
| **void** **process\_error**(**int** code) {  **case** ERR\_OK  **case** ERR\_COMMON:   cout << “не ошибка”;  **break**;  **case** ERR\_MEMORY:  cout << “ошибка памяти”;  **break**;  . . .  **default**:  cout << “неизвестная ошибка”; } |

# Операторы цикла for, while, do while

|  |
| --- |
| **for** (/\*выражение1\*/; /\*выражение2\*/; /\*выражение3\*/ ) { /\*один оператор или блок операторов\*/; } |

|  |
| --- |
| **while** (/\*условие продолжения цикла while\*/) { /\*блок операторов\*/; /\*управление условием\*/; } |

|  |
| --- |
| **do** // начало цикла do while { /\*блок операторов\*/; } **while** (/\*условие выполнения цикла\*/); // конец цикла do while |

# Оператор goto

|  |
| --- |
| **goto** label; // другой код label: //код |

# Комментарии и отступы

Комментарии можно использовать при тестировании, чтобы сделать неактивными определенные строки кода, однако для этой цели лучше подходят директивы препроцессора #if и #endif, поскольку ими можно окружить код, содержащий комментарии, однако вложение комментариев не допускается.

Комментарии в C++ записываются одним из следующих способов:

* Символы /\* (косая черта и звездочка), за которыми следует любая последовательность символов, включая переводы строки, после чего ставятся символы \*/. Это тот же синтаксис, который используется в ANSI C.
* Символы // (две косые черты), за которыми следует любая последовательность символов. Символ перевода строки, непосредственно перед которым нет обратной косой черты, завершает комментарий, оформленный таким способом. Поэтому такие комментарии часто называют однострочными.

Символы, используемые для оформления комментариев (/\*, \*/ и //), не имеют специального значения внутри символьной константы, строкового литерала, или комментария. Однако вложение комментариев, оформленных первым способом, не допускается.

**Стиль о́тступов** (индентация) — правила форматирования исходного кода, в соответствии с которыми отступы программных блоков проставляются в удобочитаемой манере.

Используемый стиль отступов обычно особо оговаривается в стандарте оформления кода.

Редакторы текста, входящие в состав большинства популярных сред разработки, часто предоставляют средства для поддержки используемого стиля отступов, например, автоматическую вставку пробелов/табуляции при вводе скобок, обозначающих начало/конец логического блока.

# Понятие функции. Объявление функции

<тип возвр. знач-я><имя ф-и>([аргументы])[модификаторы]

|  |
| --- |
| / / Объявление прототипа функции с двумя целыми параметрами / / функция принимает два аргумента и возвращает их сумму **int** **sum**(**int** num1, **int** num2); |

В языках C и C++, функции должны быть объявлены до момента их вызова. Вы можете объявить функцию, при этом функция может возвращать значение или — нет, имя функции присваивает программист, типы данных параметров указываются в соответствии с передаваемыми в функцию значениями. Имена аргументов, при объявления прототипов являются необязательными:

**int** **sum**(**int** , **int** ); // тот же прототип функции

Определение функции :

<объявление функции>

{

тело;

}

# Передача аргументов в функцию

1. По значению
2. По ссылке

|  |
| --- |
| **void** **f**(**int** a){ a=10;} **void** **g**(**int** &a){a=10;} **void** **h**() {  **int** a=0;  **int** b=0;  f(a); g(b);  std::cout << ”a=” << a << ” b=” << b; // а=0 b=10 } |

1. Передача указателя

|  |
| --- |
| **void** **h**(**int** \*a) // через указатель {  \*a=10; //обязательно проверять if(a!=nullptr) \*a=10; }  **void** **e**() {  **int** a =0;  h(&a);  std::cout<<a<<endl; //10  h(0); //ошибка }  **struct** BigOne {  //500 полей } **void** **f**(BigOne val); //копирование **void** **f**(BigOne &val); //не копируется **void** **f**(**const** BigOne &val); // не копируется и нельзя изменить **void** **g**() {  BigOne val;  f(val); } |

передача массивов

|  |
| --- |
| **void** **f**(T\*) {  T t[100];  f(t);  **sizeof**(t)!=**sizeof**(T\*)  //размер всего массива !=размер указателя  **sizeof**(t)/**sizeof**(T) // кол-во элементов } **void** f(T \*t, **int** size); **void** **g**(T \*t[100]); //размер не нужен void h()  {  T t[1000];  f(t,1000);//ok  g(t); //error |

Передача двумерных массивов

|  |
| --- |
| **void** **f**(T \*t, **int** N, **int** M) {  //i, j -> t[i][j]  t[i][j];//error  t[i\*M+j];//ok } T t[10][20]; f(t,10,20); |

|  |
| --- |
| **int** a; **int** &r=a;//ok **int** &r2=0;//error **const** **int** &r3=0;//ok |

|  |
| --- |
| **void** **f**(**int** &arg1, **const** **int** &arg2); **void** **g**() {  f(0,0);//error  **int** a;  f(a,0);//ok } |

# Возврат значения из функции, оператор return

1. T функция с возвратом
2. void функция не возвращает значение

return [выражение] выражение -> T

void f(T &ret) обяз. возвр.

void f(T \*ret) необяз.

ссылки на локальные переменные возвращать нельзя

|  |
| --- |
| **int** \***f**(){  **int** a;  **return** &a; } |

# Перегрузка имен функций

Под перегрузкой функции понимается, определение нескольких функций (две или больше) с одинаковым именем, но различными параметрами. Наборы параметров перегруженных функций могут отличаться порядком следования, количеством, типом.

пример перегрузки:

**void** **print**(**int**);   
**void** **print**(**long**);   
**void** **print**(**double**);   
**void** **print**(**float**);   
**void** **print**(**char** \*);

# Правила разрешения перегрузки в случае одного аргумента

|  |
| --- |
| **void** **print**(**int**); //1 **void** **print**(**const** **char** \*); //2 **void** **print**(**double**); //3 **void** **print**(**long**); //4 **void** **print**(**char**); //5 **void** **h**(**char** c. **int** i, **short** s, **float** f) { Преобразования аргументов: Точное соответствие  print(c); //5 print(i); //1 print(‘a’); //5 print(49); //1  print(“a”); //2 Интегральные продвижения типов  print(s); //1  print(f); //3 **bool** -> **int** **short** -> **int** **char** -> **int** **float** ->**double** Стандартные преобразования типов **int** -> **double** **int** -> **long** **double** -> **int** T\*-> **void** Пользовательские преобразования  } |

пример неоднозначного поведения

|  |
| --- |
| **void** **f**(**char**); **void** **f**(**long**); f(0);  **int** -> **char** **int**->**long** //равнозначие f(**static\_cast**<**char**>(0)); //int->char f(0L); // long |

тип возвращаемого значения не используется в перегрузке функций

|  |
| --- |
| **void** **f**(); **int** **f**(); //Ошибка |

# 

# Правила разрешения перегрузки в случае нескольких аргументов

|  |
| --- |
| **int** **pow**(**int**, **int**); //1 **double** **pow**(**double**, **double**); //2 **complex** **pow**(**double**, **complex**); //3 **complex** **pow**(**complex**, **int**); //4 **complex** **pow**(**complex**, **double**); //5 **complex** **pow**(**complex**, **complex**); //6 **int** **pow**(**char**, **int** b=10); //7 **int** **pow**(**int**); //8 **void** **pow**(**int**, …); //9 **void** **k**(**complex** z); { **int** i = pow(2, 2); //1 **double** d = pow(2.0, 2.0); //2 **int** j = pow(‘a’, ’b’); //1 pow(z, 2); //4 pow(2.0, 2); //неоднозначный вызов, 1 или 2 } |

На самом деле достаточно 3х функций:

|  |
| --- |
| **int** **pow**(**int**, **int**); **double** **pow**(**double**, **double**); **complex** **pow**(**complex**, **complex**); |

В процессе выбора среди перегруженных функций с двумя и более аргументами на основе правил отбираются функции с наилучшими соответствиями по каждому аргументу. Вызывается в итоге та из них, у которой для одного аргумента соответствие наилучшее, а требующиеся для других аргументов преобразования не хуже необходимых преобразований у остальных функций. Если такой функции не находится, то вызов отвергается как неоднозначный. В случае pow(2.0, 2) из примера выше вызов неоднозначен, так как первый аргумент наилучшим образом соответствует варианту pow(double, double), а второй pow(int, int).

# Аргументы функции по умолчанию

|  |
| --- |
| **void** **print**(**int** v, **int** base); **void** **f**() {  print(1, 10);  print(1, 16);  print(100, 10);  print(777, 10);  … } |

|  |
| --- |
| **void** **print**(**int** v, **int** base = 10); |

если второй аргумент не указан, он принимает значение 10.

|  |
| --- |
| print(100); print(777); |

Ограничения:

1. если у функции есть объявление и в нем есть аргументы по умолчанию, то в определении их указывать нельзя.
2. аргументов по умолчанию может быть сколько угодно, но все должны быть справа в порядке уменьшения частоты использования.

# Функции с переменным числом аргументов

|  |
| --- |
| **int** **printf**(**const** **char** \*msg …); // int printf(const char \*msg, …); - тоже можно |

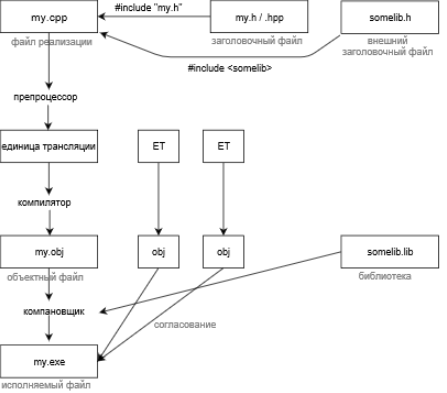
Предположим, что есть функция печати сообщения об ошибке:

|  |
| --- |
| **void** **error**(**int** severity, …){ // массив строк, заканчивающийся 0  va\_list ap; //макроопределение  va\_start(ap, severity); //инициализация работы со стеком,  //severity - последний известный аргумент  //(или пустота)  **for** ( ; ; )  { **char** \*p = va\_arg(ap, **char**\*);  **if** (p == 0) **break**;  cerr << p << ‘ ‘;  }  va\_end(ap); //завершение работы со стеком  cerr << ‘\n’;  **if** (severity) exit(severity); } |

# Указатели на функции

|  |
| --- |
| **void** **error**(**const** **char**\*); // функция **void** (\*pf)(**const** **char**\*); //указатель pf = error; pf = &error; //это и предыдущая запись - одно и то же pf(“Hello”); std::function<**void**(**const** **char**\*)> pfunc = error; pfunc(“Hello”); |

# Общая схема программы. Заголовочные файлы



Что не нужно вносить в заголовочный файл:

1. Определения функций(объявления можно) исключение inline

inline int f() {return 0;}

int a = f(); →компилятор→ int a = 0;

inline функции должны быть определены в заголовочном файле

1. Определение переменных (имена, массивы и т.д.)
2. Неименованные пространства имен

Что следует выносить в заголовочный файл:

1. Именованные пространства имен
2. Определение типов

struct T {...};

class T {...};

1. Шаблоны

template

1. Объявление функций, определение inline функций
2. Объявления переменных (extern int a;)
3. Константы (const int A=10;)
4. Перечисление (enum)

Заголовочные файлы подключаются с помощью Директивы препроцессора

#директива

#include "имя файла" - подключение своих заголов. файлов

#include <имя файла> - подключение библиотек(внешних файлов)

my.cpp my.h

#include "my.h" **int** a;

**int** b;

...

#include "my.h"

Единица Трансляции

**int** a;

**int** b;

...

**int** a; //ошибка

# Компиляция. Внешние заголовочные файлы

#include <имя файла> - подключение библиотек(внешних файлов)

Пространство имен (namespace)

std - пространство имен стандартной библиотеки

using namespace std; или std::cout;

|  |  |
| --- | --- |
| my.h  int get\_error\_code()  my.exe //мой код  get\_error\_code | lib.h  int get\_error\_code()  somelib.h //их код  get\_error\_code |

get\_error\_code разные функции!! включаем пространство имен

|  |  |
| --- | --- |
| my.h  namespace My  {  get\_error\_code();  }  My::get\_error\_code() | lib.h  namespace Lib  {  get\_error\_code();  }  Lib::get\_error\_code() |

namespace [<имя пр-ва имен>]

{

объявления и определения

}

Если у пространства имен есть имя, то оно именованное, если нет - неименованное.

Обращение:

**namespace** A

{

**struct** S{};

S f(**const** S&s);

}

A::S

A::f(**const** S&s)

Пространство имен является открытым

Открытость namespace

|  |  |
| --- | --- |
| file1.cpp  **namespace** A  {  **int a;**  } | file2.cpp  **namespace** A  {  **int b;**  } |

Вложенность namespace

|  |
| --- |
| **namespace** A  {  **namespace** B  {  **int** val;  **void** f();  }  }    A::B::f() |

Пространство имен может быть анонимным

Директивы включения:

* Объявление

|  |
| --- |
| **using** A::val;  val = 10;  A::f(); |

* Директива

|  |
| --- |
| **using** **namespace** A  val = 10;  f() |

# Компоновка. Правило одного определения

Компоновка (Linkage) - процесс согласования имен, имен функций и тд.

1. Внутренняя(только в рамках своей единицы трансляции)
2. Внешняя

|  |  |
| --- | --- |
| file1.cpp  **int** x=1;  **int** b=1;  **extern** **int** c; | file2.cpp  **int** x;  **extern** **double** b;  **extern** **int** c; |

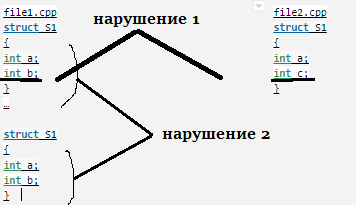
не скомпонуется т.к.

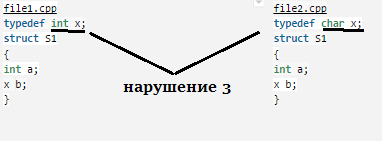
1. x объявлен дважды
2. c не определена(extern-объявление)
3. b не согласовано(разный тип)

Правило одного определения(ODR One Definition Rule)

Два определения класса, шаблона или встраиваемой функции (inline) считается одним и тем же определением тогда и только тогда, когда:

1. Они полексемно идентичны
2. Они находятся в разных единицах трансляции
3. Смысл этих лексем одинаковый в разных единицах трансляции





Решение:

|  |  |
| --- | --- |
| file.h  **struct** S1  {  **int** a;  **int** b;  ...  } | |
| file1.cpp  #include “file.h” | file2.cpp  #include “file.h” |

# Классы и объекты. Переменные и функции члены классов

Класс - тип, определяемый пользователем

**class** <имя> объявление класса

{

члены класса

}

Объект(класса) - переменные

<имя класса> - это имя типа

class Date{...};

Date (класс) date(объект);

Различия между структурами и классами

Struct Foo = Class Foo

{....} {public:}

↑ Класс у которого все члены открыты по умолчанию

Члены класса:

* переменные-члены (поля);
* функции-члены (методы);

//код в следующем вопросе

# Управление режимом доступа к членам классов

private - закрытые члены класса,

public - открытые члены класса(интерфейс).

|  |
| --- |
| **class** Foo  {  //закрытые члены класса(private) **public:**   //открытые члены класса  **private:**   //закрытые члены класса  **public:**   //открытые члены класса }; |

Члены класса:

* переменные-члены (поля);
* функции-члены (методы);

|  |
| --- |
| сlass Date  { **public**:   **void** **init**(**int** dd, **int** mm, **int** yy);   **void** **add\_day**(**int** n);   **void** **add\_month**(**int** n);   **void** **add\_year**(**int** n);  //……  **private**:   **int** d,m,y;  }; |

|  |
| --- |
| Data date;  date.init(22,11,2016);  date.add\_day(1);  date.d +=1; //ошибка, доступ к закрытому члену класса запрещен |

# Конструкторы класса

## Жизненный цикл объекта класса

|  |
| --- |
| {  **Date** date; //создание объекта   //…… использование  }; //удаление объекта класса |

Создание объекта:

1. Выделение памяти под объект.
2. Инициализация выделенной памяти (делает конструктор).

Конструктор - это специальный член класса, который вызывается средой после выделения памяти. Его назначение - инициализировать выделенную память.

Удаление объекта:

1. Деинициализация выделенной памяти (делает деструктор).
2. Освобождение выделенной памяти.

Деструктор - это функция, которая вызывается средой и которая деинициализирует память. Он НЕ освобождает память.

|  |
| --- |
| **class** Date { **public:**  //конструкторов может быть много  Data(); //конструктор по умолчанию   Date(**int** dd, **int** mm, **int** yy); //конструктор с аргументами //деструктор один   ~Date(); //….. };   **Date** date; //вызов конструктора по умолчанию;  **Date** **today** (22,11,2016); //вызов конструктора с аргументами |

Если есть другие конструкторы, то конструктор по умолчанию не создается.

## Исключения в конструкторе и деструкторе

|  |
| --- |
| File(**const** **char** \*name)  {   //открытие файла   **if** (файл не открыт) **throw** exeption; //говорит о том, что создание объекта завершилось c ошибкой и его использование невозможно. };   //….  **try**  {  File **f**(“путь”);   //использование f }  **catсh** (exeption &e) {   //обработка  } |

Исключений в деструкторе быть не должно! Это нарушает цепочку освобождения памяти.

## Массивы динамических объектов

**new T[размер]** - выделяется память,

**delete[]** - освобождение памяти.

foo = new Foo:

1. выделение памяти
2. инициализация(Foo())

delete foo:

1. деинициализация (~Foo())
2. освобождение памяти.

foo = new Foo[100]:

1. выделение памяти
2. инициализация(конструктор Foo() вызывается для каждого элемента)

delete[] foo:

1. деинициализация (деструктор ~Foo() вызывается для каждого элемента)
2. освобождение памяти.

delete foo - не ошибка, если нет динамических структур данных.

# Статические члены класса

В классе может быть 2 типа статических членов: переменная и функция.

date.cpp:

|  |
| --- |
| **class** Date  { **public:**  **Date**();   //…  **static** **void** **set\_default**(**int** d, **int** m, **int** y); **private:**  static Date default\_date;   //... }; //….. Date Date::default\_date(1,1,1900); **void** Date::set\_default(**int** d, **int** m, **int** y ) {  default\_date=Date(d,m,y) }; |

my.cpp

|  |
| --- |
| Date::set\_default(1,1,1900); // 1 на всех Date d1,d2; //d1==d2==1,1,1900 |

# Копирование объектов класса

|  |
| --- |
| Date(**const** Date &other); //конструктор копирования Date dt = today;//вызов конструктора копирования  dt = today; //операция присваивания |

Конструктор копирования нужен нам для того, чтобы создавать «реальные» копии объектов класса, а не побитовую копию объекта. Иногда это принципиально важно. Такую «реальную» копию объекта надо создавать в нескольких случаях:

* когда мы передаем объект в какую-либо функцию в виде параметра;
* когда какая-либо функция должна вернуть объект класса в результате своей работы;
* когда мы в главной функции один объект класса инициализируем другим объектом класса.

|  |
| --- |
| **class** Table  { **public:**   **Table**(**int** sz); //создание таблицы   ~Table(); //уничтожение **private:**  **int** size; //кол-во элементов  **int** \*p;//память };  Table::Table(**int**):size(sz)//:size(sz)-список инициализации  {  p = **new** **int**[sz]; } Table::~Table() {  **delete** p; };  //….. {  Table **t1**(1000);   Table t2 = t1;  //….. }; //падение, вызов деструктора будет дважды, первый раз он освободит память, на //которую указывает p, а во второй раз будет неопределенное поведение, т.к. p и //в t1 и в t2 указывает на один и тот же участок памяти |

решение проблемы:

|  |
| --- |
| **class** Table  {  **public:**   //…  Table(**const** Table &other);  Table **operator**=(**const** Table &other);  //… Table (**const** Table &othe) {  size = other.size;   p = **new** **int**[sz];   memcpy(p,other.p,size\***sizeof**(**int**)); }; Table & **operator**=(**const** Table &other) {  **if** (&other != **this**)   {  **delete**(p);   size = other.size;   p=**new** **int**[size];   memcpy(p,other.p,size\***sizeof**(**int**));   } } } |

# Константные функции-члены

Объявление функции-члена с ключевым словом const указывает, что функция доступна только для чтения и не изменяет объект, для которого вызывается. Функция-член-константа не может изменить никакие нестатические элементы данных или вызвать какую-либо функцию-член, не являющуюся константой. Чтобы объявить функцию-член-константу, поместите ключевое слово const после закрывающей скобки списка аргументов. Ключевое слово const необходимо указать как в объявлении, так и в определении.

|  |
| --- |
| **class** Date  {  **public**:   Date( **int** mn, **int** dy, **int** yr );   **int** **getMonth**() **const**; // read-only функция  **void** **setMonth**( **int** mn ); // не read-only функция. Не м.б. const **private**:   **int** month;  };    **int** Date::getMonth() **const**  {   **return** month; // Ничего не изменяет }  **void** Date::setMonth( **int** mn )  {   month = mn; // Изменяет месяц }  **int** **main**()  {   Date **MyDate**( 7, 4, 1998 );   **const** Date **BirthDate**( 1, 18, 1953 );   MyDate.setMonth( 4 ); // Okay   BirthDate.getMonth(); // Okay   BirthDate.setMonth( 4 ); // C2662 Error  } |

# Структуры и классы

Структура - объединение данных различных типов.

|  |
| --- |
| **struct** Address {  **char** \*name;  **long** number;  **char** \*street;  **char** \*town;  **char** state[2];  **long** zip; };  **void** **f**() {  Address jd;  jd.name = "Jim";  jd.number = 322;  ... } |

Структуру можно инициализировать:

|  |
| --- |
| Address jd = {"Jim", 61, "Street", "NewYsman", {'N', 'Y'}, 4710};  Address jd2; jd2 = jd1; //Копирование полей |

Сравнивать структуры нельзя!

|  |
| --- |
| **if** (jd2 == jd1) //Ошибка  ... |

|  |
| --- |
| **void** **print**(Address \*addr) {  cout << (\*addr).name;  cout << addr->name;// -> доступ к члену структуры через указатель  ... } |

Класс - это тип, определяемый пользователем.

|  |
| --- |
| **class** <имя класса> {  члены класса }; <имя класса> - это имя типа  **class** Date{...}; Date date;//Date - класс, date - объект |

Различия между структурами и классами

|  |
| --- |
| **struct** Foo {  ... };  **class** Foo {  **public**:  ... }; |

Структура - класс, у которого все члены являются открытыми.

Интерфейсом класса являются его открытые члены.

Интерфейс структуры - все члены.

В классе по умолчанию все члены скрыты.

private - закрытые члены

public - открытые члены

Члены класса:

* переменные-члены класса(поля);
* функции-члены класса(методы).

|  |
| --- |
| **class** Date {  **public**:  **void** **init**(**int** dd, **int** mm, **int** yy);  **void** **add\_day**(**int** n);  ...  **private**:  **int** d,m,y; };  **void** **g**() {  Date date;  date.init(22,11,2016);  date.add\_day(1);  ...  date.d += 1;//Ошибка } |

# Определение функций членов класса

Определить функции-члены класса можно внутри описания класса или за его пределами.

В первом случае функция считается встраиваемой (если разрешена соответствующая опция в установках компилятора).

Встраиваемая функция характерна тем, что компилятор С++ обрабатывая вызовы этой функции в программе, заменяет их на непосредственно объектный код, соответствующей определению этой функции. Вследствие этого нужно помнить, что встраиваемые функции, как правило, имеют короткие определения.

В показанном ниже объявлении класса конструктор Account является встраиваемой функцией.

|  |
| --- |
| **class** **Account**  {  **public**:   Account(**double** initial\_balance) { balance = initial\_balance; }   **double** **GetBalance**();   **double** **Deposit**( **double** Amount );   **double** **Withdraw**( **double** Amount );  **private**:   **double** balance;  }; |

Для определения функции-члена класса за пределами описания класса, необходимо определить ее где-нибудь в программе после определения класса, членом которого она является.

Но, для того чтобы показать, что эти функции принадлежат классу, нужно перед именем функции записать имя класса и оператор разрешения области видимости ::.

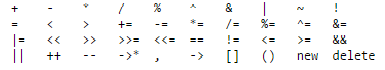
Определение класса вводит область видимости, которой принадлежат имена членов класса. Определение функции-члена класса также принадлежит области видимости класса. Отсюда следует, что в теле функции-члене класса можно обращаться к другим членам этого класса просто по имени.

|  |
| --- |
| **double** Account::GetBalance()  {   **return** balance;  } |

# 

# Функции-операции класса

Можно описать функции, определяющие интерпретацию следующих операций:



Нельзя изменить приоритеты этих операций, равно как и синтаксические правила для выражений. Так, нельзя определить унарную операцию % , также как и бинарную операцию !. Нельзя ввести новые лексемы для обозначения операций, но если  
набор операций вас не устраивает, можно воспользоваться привычным обозначением вызова функции.

Именем операторной функции является служебное слово operator, за которым идет сама операция, например, operator<<. Операторная функция описывается и вызывается как обычная функция. Использование символа операции является просто краткой формой записи вызова операторной функции:

|  |
| --- |
| **void** **f**(**complex** a, **complex** b) {  **complex** c = a + b; // краткая форма  **complex** d = **operator**+(a,b); // явный вызов } |

# Перегрузка унарных операций класса

@ – операция

@a – префиксная унарная операция (++a, --b)

|  |
| --- |
| 1. a.**operator**@() //функция-член класса без аргументов 2. operator@(b) //глобальная функция |

a@ – постфиксная унарная операция (a--, b++)

|  |
| --- |
| 1. a.**operator**@(**int**) //функция-член класса 2. operator@(b, int) //глобальная функция |

int – не аргумент здесь, а просто говорит о том, что это постфиксная операция.

|  |
| --- |
| **class** Date { … **public**:  Date **operator**++(**int**)  {  Date tmp =\* **this**;  \*this += 1; //предположим, что операция += уже реализована  **return** tmp;  }   Date& **operator**++()  {  **return** \***this** += 1 ;  } } Date **d**(6, 12, 2016); d++; ++d; //лучше, дешевле |

# Перегрузка бинарных операций класса

@ – операция

a@b – бинарная операция (примеры: a+b, a\*b)

A, B – классы, a, b – объекты (A a; B b;)

|  |
| --- |
| 1. a.**operator**@(b) //член класса 2. **operator**@(a, b) //глобальная функция |

|  |
| --- |
| y=A\*x+b; //синтаксический сахар y.**operator**=(**operator**+(A.**operator**\*(x), b)); Vector Vector.**operator**=(Vector) Vector **operator**+(Vector, Vector) Vector Matrix::**operator**\*(Vector) |

Вывод на экран произвольного объекта.

|  |
| --- |
| cout<<”a=”<<a; Matrix m; cout<<”M=”<<m; |

Что нужно сделать:

|  |
| --- |
| ostream& **operator**<<(ostream &os, **const** Matrix &m) {  **for**(...)  **for**(...)  os << val << ’ ’; //val – фундаментальное  **return** os; } |

Какие операции нельзя перегружать:

1. ::
2. . (выбор члена) (А оп-цию –> можно перегружать)
3. .\* (выбор члена класса через указатель на классовый член)
4. ?: (тернарная условная операция)

|  |
| --- |
| **void** **operator**+(Matrix, Matrix) {  //форматирование диска  //ха-ха-ха  } **void** **operator+**(Matrix, Matrix) {  //должна Matrix + Matrix } |

a+=b ➝ a.operator+=(b)

a=a+b

↙ ↘

a.operator=(a) a.operator+(b)

|  |
| --- |
| **class** **complex** {  **double** re, im; **public**:  **complex** **operator**+=(**double** a)  {  re += a;  **return** \***this**;  } } **complex** **operator**+(**double** a, **complex** b) {  **complex** tmp = b;  **return** tmp += a; } **void** **f**(**complex** x, **complex** y) {  2 + x; //operator+(double, complex)  x + 2; //operator+(complex, double)  x + y; //operator+(complex, complex) } |

# Дружественные функции класса

(Нет в лекциях)

Функция – не член класса, – имеющая доступ к его закрытой части, называется другом этого класса. Функция может стать другом класса, если в его описании она описана как friend (друг). Например:

|  |
| --- |
| **class** matrix; **class** vector { **float** v[4]; // ... **friend** vector **multiply**(**const** matrix&, **const** vector&); }; **class** matrix { vector v[4]; // ... **friend** vector **multiply**(**const** matrix&, **const** vector&); }; |

Функция-друг не имеет никаких особенностей, за исключением права доступа к закрытой части класса. В частности, в такой функции нельзя использовать указатель this, если только она действительно не является членом класса. Описание friend является настоящим описанием. Оно вводит имя функции в область видимости класса, в котором она была описана, и при этом происходят обычные проверки на наличие других описаний такого же имени в этой области видимости. Описание friend может находится как в общей, так и в частной частях класса, это не имеет значения. Теперь можно написать функцию multiply, используя элементы вектора и матрицы непосредственно:

|  |
| --- |
| vector **multiply**(**const** matrix& m, **const** vector& v) {  vector r;  **for** (**int** i = 0; i<3; i++) { // r[i] = m[i] \* v;  r.v[i] = 0;  **for** ( **int** j = 0; j<3; j++)  r.v[i] +=m.v[i][j] \* v.v[j];  }  **return** r; } |

Отметим, что подобно функции-члену дружественная функция явно описывается в описании класса, с которым дружит. Поэтому она является неотъемлемой частью интерфейса класса наравне с функцией-членом. Функция-член одного класса может быть другом другого класса:

|  |
| --- |
| **class** x  {  // ...  **void** **f**(); }; **class** y  {  // ...  **friend** **void** x::f(); }; |

Вполне возможно, что все функции одного класса являются друзьями другого класса. Для этого есть краткая форма записи:

|  |
| --- |
| **class** x  {  **friend** **class** y;  // ... }; |

В результате такого описания все функции-члены y становятся друзьями класса x.

# Операция индексирования operator[]

|  |
| --- |
| Table **t**(100); //таблица из 100 int |

|  |
| --- |
| **int** a = t[10]; операция индексирования  t[0] = 100; lvalue (леводопустимая) |

Операция индексирования – это операция, возвращающая ссылку на int и имеющая вид:

|  |
| --- |
| **class** Table { **private**:  **int** size; //количество элементов  **int** \*p; //память … **public**:  …  **int**& **operator**[](**int** idx) // вместо int (тип аргумента) может быть  // любой тип   {  **return** p[idx];  } } |

# Операция функционального вызова operator()

|  |  |
| --- | --- |
| void f(int);  f(0); //вызов функции | Foo f; //объект  f(0); //вызов объекта |

|  |
| --- |
| Объект, допускающий вызов, называется функциональным объектом (функтором). Операция функционального вызова - функция-член класса. **void** **f**(vector <**complex**> vc, **complex** a) → {vc[0] + a, vc[1] + a, ...} for\_each(vc.begin(), vc.end(), Add(a)); //begin, end - итераторы, Add - операция //Add - функтор. |

|  |
| --- |
| **class** Add {  **double** a;  **public**:  **complex**(**double** \_a): a(\_a) {}  **void** **operator**()(**complex** &z)  {  z +=a;  }  … } |

# Операции инкремента и декремента

|  |
| --- |
| **class** Date {  …  **public**:  Date **operator**++(**int**)  {  Date tmp = \***this**; //tmp - временная переменная  \***this** += 1; //в лекции tmp += 1;, но правильно вроде так  **return** tmp;  }  Date &**operator**++()  {  **return** \***this** += 1;  } } |

# Наследование классов. Производные классы

Наследование классов - это способ расширять функциональность классов.

|  |  |
| --- | --- |
| Base | **class** Base; **class** Derived: **public** Base; //здесь public - способ наследования |
| ↑ |
| Derived |

Способы наследования: public, protected, private. Derived - это подкласс Base. Производный объект может выступать как базовый.

protected - защищенное, private - закрытое.

Пример.

|  |
| --- |
| **class** Car: **public** Transport; //автомобиль является транспортом  **class** Car: **private** Coleso, **private** Kuzov, **private** Electrika; //автомобиль состоит из колеса, кузова и электрики |

Пример. Класс с описанием данных о сотруднике.

|  |
| --- |
| **class** Employee;  **class** Manager: **public** Employee;    **class** Employee  {   **public**:   string first\_name, last\_name;   **char** middle\_name;   date hiring\_date;   **short** department;   **void** **Print**();  };    **class** Manager: **public** Employee  {   **public**:   **short** level;   list <Employee> group;   **void** **Print**();  };    Manager m;  Employee e;  e=m; //ok  m=e; //ошибка |

# Печать информации о сотрудниках.

|  |
| --- |
| **void** Employee::Print()  {   std::cout<<"First name: "<<first\_name<<"\n"   <<"Last name: "<<last\_name<<"\n"   ...   <<"Departament: "<<department<<   <<std::endl;  }  **void** Manager::Print()  {   Employee::Print();   std::cout<<"Level: "<<level<<"\n"  }    Manager m;  m.first.name=...  m.Print();  Employee e=m;  e.Print(); |

# Конструирование и уничтожение объектов при наследовании классов

|  |
| --- |
| **class** Base; **class** Derived: **public** Base; |

# 1. Инициализация: Base()**→**Derived()

2. Деинициализация: ~Derived()**→~**Base()

Конструкторы не наследуются!

Пример.

|  |
| --- |
| **class** A  {   **public**:   A(**int** a) {...}  };    **class** B: **public** A  {   **public**:   B():A(0) {...}   B(**int** b):A(b) {...}  };    **class** C: **public** B  {   **public**:   C() {...}  }; |

# Доступ к полям и функциям из базового класса к производному.

|  |  |
| --- | --- |
| **class** A  {   **private**:   **int** a;   **public**:   **int** b;   **protected**:   **int** c;  }; | **class** B: **public** A  {   **public**:   **void** **f**()   {   a=1; //ошибка   b=2; //ок   c=3; //ок   }  }; |

|  |
| --- |
| A a;  a.a=1; //ошибка  a.b=2; //ок  a.c=3; //ошибка |

# В списке инициализации нельзя писать поля базового класса.

|  |  |
| --- | --- |
| **class** A  {   **public**:   A(**int** a): val(a) {}   **private**:   **int** val;  }; | **class** B: **public** A  {   **public**:   B(**int** b):val(b) { } //ошибка   B(**int** b):A(b) { } //ок  }; |

Порядок инициализации переменных класса зависит от порядка их объявления в классе, а не в списке инициализации.

|  |
| --- |
| **class** A  {   **public**:   A(): d(0.0), a(0) {} //ошибка, неверный порядок   **private**:   **int** a;   **double** d; }; |

# Контроль доступа при наследовании классов

|  |  |  |  |
| --- | --- | --- | --- |
| **y:тип наследования→**  **x: тип доступа↓** | **public** | **private** | **protected** |
| **public** | public | private | protected |
| **private** | private | private | private |
| **protected** | protected | private | private |

|  |
| --- |
| **class** A { **public**:  **int** a; **private**:  **int** b; **protected**:  **int** c; }; //x и y берём из таблицы **class** B : y A { x:  **int** a;  x:  **int** b; x:  **int** c; }; |

# Полиморфизм. Виртуальные функции

Полиморфизм базируется на двух китах:

* виртуальные функции;
* косвенная адресация – обращение к переменной по ссылке(указатель, который автоматически разыменуется) или указателю.

|  |
| --- |
| **class** A { **public**:  **int** val; }; A a; a.val = 1; //прямая адресация A &ra = a; A \*pa = &a; ra.val = 1; //косвенная адресация pa->val = 1;  **class** Employer { **public**:  ... **void** **print**(); };  **void** **f**(Employer e) { e.print(); }  **void** **g**(Employer e, Manager m) {  f(e);  f(m); //информация о manager не напечатается.Вызов два раза из Employer } |

Чтобы такого не было делаем:

|  |
| --- |
| **void** **f**(Employer &e) **class** Employer {  **public**:  **virtual** **void** **print**(); }; |
|  |

**Виртуальная функция** – функция, адрес которой может быть вычислен на этапе выполнения.

**Полиморфизм = виртуальные функции + косвенная адресация.**

|  |
| --- |
| **void** **f**(Employer e)//срезка {  e.print();//всегда вызывается Employer::print(); } |

|  |
| --- |
| **void** **f**(vector<Employer\*> &v) {  **for** (Employer \*e : v)  e->print(); }  //эквивалентно vector<Employer \*>::iterator it; **for** (it = v.begin(); it != v.end(); ++it) (\*it)->print();  **class** A { **public**:  **virtual** **void** **func**()  /\*если убрать virtual, то класс станет не полиморфным  =>таблицы всех методов не будет\*/  {  cout << "A::func()";  } };  **class** B :**public** A { **public**:  **void** **func**()  {  cout << "B::func()";  } }; **void** **f**(A \*a) {  a->func(); }  **void** **f**(A a) {  a.func(); }  **void** **g**() {  B b;  f(&b);//B::func()-полиморфизм  A a;  f(&a);//A::func() } |

Untitled Diagram (2).png

# Абстрактные классы

|  |
| --- |
| **class** Shape { **public**:  **void** **draw**() {};//нарисовать фигуру  **void** **rotate**(**double**) {};//повернуть фигуру };  **void** **f**() {  Shape s;  s.draw(); } |

Нужно использовать чистые виртуальные функции (pure virtual).

Класс, в котором есть хотя бы одна чистая виртуальная функция, называется абстрактным.

Экземпляры этого класса создавать нельзя.

|  |
| --- |
| **class** Shape //абстрактный { **public**:  **virtual** **void** **draw**() = 0;  **virtual** **void** **rotate**() = 0; }; |

**Абстрактные классы=интерфейсы**

|  |
| --- |
| **class** Polygon :**public** Shape//абстрактный {  list<pair<**double**, **double**>>points; };  **class** Triangle :**public** Polygon { **public**:  **void** **draw**() { ... }  **void** **rotate**(**double** angle)  {  ...  } }; |

|  |
| --- |
| Triangle t;//ок **class** Circle {  pair<**double**, **double**>center;  **double** radius; **public**:  **void** **rotate**(**double** angle) {}  **void** **draw**(){...} }; |

Представим, что есть алгоритм, который поворачивает и рисует список фигур.

|  |
| --- |
| **void** rotate **Draw**(list<shape\*>b, **double** angle) {  **for** (Shape \*s : l)//for по коллекции  {  s->rotate(angle);//интерфейс  s->draw();//интерфейс  } };  **void** **f**(A &a) {  **try**  {  B &b = **dynamic\_cast**<B&>(a);  }  **catch** (std::bad\_cast)  {  //B не производная от A  } } |

Вот в чём отличие от указателей: ссылки не могут быть нулевыми.

# Множественное наследование

|  |
| --- |
| **class** A { **public**:  **int** val; }; **class** B :**public** A; **class** C :**public** A, **public** B; |

Copy of Untitled Diagram.pngUntitled Diagram (1).png

|  |
| --- |
| **class** C {  ...  **void** **f**()  {  val = 1;/\*неизвестное поведение ,т.к val  встречается в С дважды, т.е  A::val или B::val\*/  } };   //правильно  A::val = 1;//правая ветвь  // или  B::val = 2;//левая ветвь   C c;  c.val = 1;//ошибка  с.A::val = 2;//правильно  с.B::val = 3; |

# Виртуальное наследование

(нет в лекциях, источник - Википедия)

**Виртуа́льное насле́дование** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *virtual inheritance*) в [языке программирования C++](https://ru.wikipedia.org/wiki/C%2B%2B) — один из вариантов [наследования](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), который нужен для решения некоторых проблем, порождаемых наличием возможности [множественного наследования](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BD%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (особенно «[ромбовидного наследования](https://ru.wikipedia.org/wiki/%D0%A0%D0%BE%D0%BC%D0%B1%D0%BE%D0%B2%D0%B8%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BD%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)»), путём разрешения неоднозначности того, методы которого из [суперклассов](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%BF%D0%B5%D1%80%D0%BA%D0%BB%D0%B0%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) (непосредственных классов-предков) необходимо использовать. Оно применяется в тех случаях, когда множественное наследование вместо предполагаемой полной композиции свойств классов-предков приводит к ограничению доступных наследуемых свойств вследствие неоднозначности. Базовый класс, наследуемый множественно, определяется виртуальным с помощью ключевого слова virtual.

|  |
| --- |
| **class** A {  **public**:   **int** **foo**() {   **return** 1;   }  };    **class** B : **public** **virtual** A {};    **class** C : **public** **virtual** A {};    **class** D : **public** B, **public** C {};    **int** **main** () {   D d;   std::cout << d.foo();   **return** 0;  } |

Если убрать ключевое слово *virtual*, то метод *foo()* не может быть определён однозначно и в результате не будет доступен как объект класса *D* и код не скомпилируется.

# Статическое и динамическое приведение типов

**dynamic\_cast** - только для полиморфных типов.

|  |
| --- |
| **void** **f** (A \*a)  {   B \*b=**dynamic\_cast** <B\*>(a);  } |

если В - потомок А, то b!=nullptr, иначе b==nullptr

|  |
| --- |
| **void** **f** (A \*a)  {   B \*b=**static\_cast** <B\*>(a);   **if** (b!=0)   b→callfunc(); //падение  } |

|  |
| --- |
| **void** **g** ()  {   C c;   f (&c);  } |

# Операция typeid

(нет в лекциях)

Операция typeid обеспечивает динамическую идентификацию типов. В языке C++ оператор typeid возвращает ссылку на объект type\_info, описывающий тип объекта, к которому принадлежит оператор typeid. общая форма записи оператора typeid такова.

typeid(*объект*)

Если оператор typeid применяется к указателю полиморфного класса, он автоматически возвращает тип объекта, на который он указывает. (полиморфный класс — это класс который содержит хотя-бы одну виртуальную функцию.) Следовательно, оператор typeid можно использовать для определения типа объекта, адресуемого указателем на базовый класс.

# Определение шаблона класса

Шаблоны обеспечивают непосредственную поддержку обобщенного программирования, то есть программирования с использованием типов в качестве параметров.

Виды: шаблоны классов, шаблоны функций.

|  |
| --- |
| **template** <параметры шаблона>  **class** имя\_класса  {  } |

Объявление:

* для обычного класса: имя\_класса имя\_объекта
* для шаблонного класса: имя\_класса <конкретные\_типы> имя\_объекта

|  |
| --- |
| **template** <**class** T, **int** max>  **class** Buffer  {  T v[ma];  ...  **public**:  ...  }; |

# Конкретизация шаблона

Генерация конкретного класса из обобщенного определения шаблона называется конкретизацией шаблона.

|  |
| --- |
| **template** <**class** T, **int** width> **class** Table { … }  Table<**int**, 10> table; //конкретизация шаблона |

При генерации классов они конструируются в необходимом количестве:

|  |
| --- |
| Table<**int**, 10> table1; //конкретизация 1 Table<**double**, 10> table2; //конкретизация 2 Table<**int**, 11> table3; //конкретизация 3 Table<**int**, 10> table4; //конкретизация 1 |

# Параметры шаблонов

Параметры шаблона могут быть двух видов:

1. Шаблонные (типы с ключевыми словами class/typename).
2. Обычные.

Обычные параметры шаблона должны быть обязательно определены на этапе компиляции, то есть в качестве обычного выражения может быть использовано только константное выражение.

# Определение функциональных шаблонов

Функциональные шаблоны выглядят так:

|  |
| --- |
| **template** <аргументы шаблона> <тип возвращаемого значения> <имя функции>(<аргументы функции>) { … } |

Например:

|  |
| --- |
| //сортировка Шелла **template** <**class** T> **void** **sort**(vector<T> &v) {  **auto** n = v.size();  **for**(**auto** gap = n/2; 0 < gap; gap /= 2)  {  **for**(**auto** i = gap; i < n; ++i)  {  **for**(**auto** j = i - gap; 0 <= j; j -= gap) {  **if** (v[j + gap] < v[j])  {  std::swap(v[j], v[j + gap]);  }  **else**  {  **break**;  } }   }   } } |

# Аргументы функциональных шаблонов

Аргументы функциональных шаблонов выводятся из аргументов функции:

|  |
| --- |
| **void** **f**(vector<Foo> &v) {  sort(v); //вместо sort<Foo>(v) } |

Но это возможно сделать не всегда:

|  |
| --- |
| **template**<**class** T, **class** U> T **implicit\_cast**(U u) {  **return** u; }  **void** **g**(**int** i) {  implicit\_cast(i); //ошибка, не определен тип Т  implicit\_cast<**double**>(i); //T => double; U => int;  implicit\_cast<**double**, **char**>(i); //произойдет неявное приведение int в char } |

# Перегрузка функциональных шаблонов

К параметрам шаблонной функции нельзя применять никаких преобразований типа. Вместо этого при необходимости создаются новые варианты функции:

|  |
| --- |
| **template**<**class** T> T **sqrt**(t); **void** **f**(**int** i, **double** d, **complex** z) {  **complex** z1 = sqrt(i); // sqrt(int)  **complex** z2 = sqrt(d); // sqrt(double)  **complex** z3 = sqrt(z); // sqrt(complex)  // ... } |

Здесь для всех трех типов параметров будет создаваться по шаблону своя функция sqrt. Если пользователь захочет чего-нибудь иного, например вызвать sqrt(double), задавая параметр int, нужно использовать явное преобразование типа:

|  |
| --- |
| **template**<**class** T> T **sqrt**(T); **void** **f**(**int** i, **double** d, **complex** z) {  **complex** z1 = sqrt(**double**(i)); // sqrt(double)  **complex** z2 = sqrt(d); // sqrt(double)  **complex** z3 = sqrt(z); // sqrt(complex)  // ... } |

В этом примере по шаблону будут создаваться определения только для sqrt(double) и sqrt(complex). Шаблонная функция может перегружаться как простой, так и шаблонной функцией того же имени. Разрешение перегрузки как шаблонных, так и обычных функций с одинаковыми именами происходит за три шага. Эти правила слишком строгие, и, по всей видимости будут ослаблены, чтобы разрешить преобразования ссылок и указателей, а, возможно, и другие стандартные преобразования. Как обычно,

при таких преобразованиях будет действовать контроль однозначности.

[1] Найти функцию с точным сопоставлением параметров ($$R.13.2); если такая есть, вызвать ее.

[2] Найти шаблон типа, по которому можно создать вызываемую функцию с точным

сопоставлением параметров; если такая есть, вызвать ее.

[3] Попробовать правила разрешения для обычных функций ($$r13.2); если функция найдена по этим правилам, вызвать ее, иначе вызов является ошибкой. В любом случае, если на первом шаге найдено более одной функции, вызов считается неоднозначным и является ошибкой. Например:

|  |
| --- |
| **template**<**class** T> T **max**(T a, T b) { **return** a>b?a:b; }; **void** **f**(**int** a, **int** b, **char** c, **char** d) {  **int** m1 = max(a,b); // max(int,int)  **char** m2 = max(c,d); // max(char,char)  **int** m3 = max(a,c); // ошибка: невозможно  // создать max(int,char) } |

Поскольку до генерации функции по шаблону не применяется никаких преобразований типа (правило[2]), последний вызов в этом примере нельзя разрешить как max(a,int(c)). Это может сделать сам пользователь, явно описав функцию max(int,int). Тогда вступает в силу правило [3]:

|  |
| --- |
| **template**<**class** T> T **max**(T a, T b) { **return** a>b?a:b; } **int** **max**(**int**,**int**); **void** **f**(**int** a, **int** b, **char** c, **char** d) {  **int** m1 = max(a,b); // max(int,int)  **char** m2 = max(c,d); // max(char,char)  **int** m3 = max(a,c); // max(int,int) } |

Программисту не нужно давать определение функции max(int,int), оно по умолчанию будет создано по шаблону. Можно определить шаблон max так, чтобы сработал первоначальный вариант нашего примера:

|  |
| --- |
| **template**<**class** T1, **class** T2> T1 **max**(T1 a, T2 b) { **return** a>b?a:b; }; **void** **f**(**int** a, **int** b, **char** c, **char** d) {  **int** m1 = max(a,b); // int max(int,int)  **char** m2 = max(c,d); // char max(char,char)  **int** m3 = max(a,c); // max(int,char) } |

Однако, в С и С++ правила для встроенных типов и операций над ними таковы, что использовать

подобный шаблон с двумя параметрами может быть совсем непросто. Так, может оказаться неверно задавать тип результата функции как первый параметр (T1), или, по крайней мере, это может привести к неожиданному результату, например для вызова

max(c,i); // char max(char,int) Если в шаблоне для функции, которая может иметь множество параметров с различными арифметическими типами, используются два параметра, то в результате по шаблону будет порождаться слишком большое число определений разных функций. Более разумно добиваться преобразования типа, явно описав функцию с нужными типами.

# Параметры шаблонов по умолчанию

Шаблоны класса могут иметь аргументы по умолчанию для параметров типов или значений.Аргументы по умолчанию задаются со знаком "равно" (**=**) с последующим именем или значением типа.При наличии нескольких аргументов шаблона все аргументы после первого аргумента по умолчанию должны иметь аргументы по умолчанию.При объявлении объекта класса шаблона с аргументами по умолчанию опускайте аргументы, чтобы использовались аргументы по умолчанию.При наличии аргументов, отличных от аргументов по умолчанию, не опускайте пустые угловые скобки.

Шаблон, объявленный многократно, не может задавать аргумент по умолчанию более одного раза.В следующем коде демонстрируется ошибка.

|  |
| --- |
| template <**class** **T** = **long**> **class** **A**; template <**class** **T** = **long**> **class** **A** { /\* . . . \*/ }; // Generates C4348. |

В приведенном ниже примере шаблон класса массива определяется с типом по умолчанию **int** для элемента массива и значением по умолчанию для параметра шаблона, определяющего размер.

|  |
| --- |
| // template\_default\_arg.cpp // compile with: /EHsc #**include** <iostream> **using** **namespace** std;  **template** <**class** T = **int**, **int** size = 10> **class** Array {  T\* array; **public**:  Array()  {  array = **new** T[size];  memset(array, 0, size \* **sizeof**(T));  }  T& **operator**[](**int** i)  {  **return** \*(array + i);  }  **const** **int** **Length**() { **return** size; }  **void** **print**()  {  **for** (**int** i = 0; i < size; i++)  {  cout << (\***this**)[i] << " ";  }  cout << endl;  } };  **int** **main**() {  // Explicitly specify the template arguments:  Array<**char**, 26> ac;  **for** (**int** i = 0; i < ac.Length(); i++)  {  ac[i] = 'A' + i;  }  ac.print();   // Accept the default template arguments:  Array<> a; // You must include the angle brackets.  **for** (**int** i = 0; i < a.Length(); i++)  {  a[i] = i\*10;  }  a.print(); } |

# Специализация шаблонов

|  |
| --- |
| **Пример нерационального шаблонного кода** |
| **template** <**class** T>  **class** vector;  **void** f()  {  vector <**int**> vi; //  vector <**int**\*> vip; // компилятор  vector <**double**> vd; // создает 6 классов  vector <**double**\*> vdp; //  vector <Foo> vf; //  vector <Foo\*> vfp; //  } |

Специализация шаблонов - способ сокращения генерируемого шаблонного кода. Бывает полной и частичной

|  |
| --- |
| **Пример полной** |
| **template** <>  **class** vector <**void**\*>; //специализация для указателя void\*    vector <**void**\*> vvp; //полная специализация  vector <**int**\*> vip; //обычный шаблон |

|  |
| --- |
| **Пример частичной** |
| **template** <**class** T>  **class** vector <**T\***>:**private** vector <**void**\*> |

|  |
| --- |
| **template <>**  **class vector <void\*>**  **{**  **void \*\*p;**  **public:**  **void\*& operator[](int pos);**  **...**  **}**    **template <class T>**  **class vector <T\*>:private vector<void\*>**  **{**  **public:**  **typedef vector<void\*> Base;**  **...**  **T\*& operator[](int pos)**  **{**  **return reinterpret\_cast<T\*&>(Base::operator[](pos))**  **}**  **}** |

|  |
| --- |
| **Специализация шаблонов функций** |
| **template <class T>**  **bool less(T a, T b)**  **{**  **return a<b;**  **}**    **template <class T, class Comp=less<T>>**  **void sort (vector<T> &v)**  **{**  **...**  **//if(v[j+gap]<v[j])**  **if (Comp(v[j+gap, v[j]))**  **...**  **}**    **template <class T>**  **bool great (T a, T b)**  **{**  **return a>b;**  **}**    **void f(vector <int> &v)**  **{**  **sort(v);//по возрастанию**  **sort<int, great <int>(v);//по убыванию**  **}**    **template <>**  **bool less<>(const char \*a, const char \*b)**  **{**  **return strcmp(a, b)<0;//const char нельзя сравнивать стандартным**  **}** |

# Объявление пространств имен

using namespace std;

std - пространство имен стандартной библиотеки

|  |  |
| --- | --- |
| my.h  int get\_err\_code(); | lib.h  int get\_err\_code(); |



Решение - в my.h использовать namespace My {...}, в lib.h использовать   
namespace Lib {...}

namespace [<имя>] //именованные и неименованные(анонимные)

{

//объявление и определение

}

|  |
| --- |
| **namespace** A  {  **int** val;  }  **void** f()  {  A::val=1; //A:: - разрешение области видимости  } |

|  |
| --- |
| **Внутри могут быть и функции** |
| **namespace** A  {  **int** val;  **void** f();  }    **namespace** A  {  **struct** S{...};  S f(**const** S &S);  }    A::S A::f(**const** S &S)  {  ...  } |

|  |
| --- |
| **Пространства имен являются открытыми** |
| file1.cpp  **namespace** A  {  **int** a;  }    file2.cpp  **namespace** A  {  **int** b;  }  //объединение множеств |

|  |
| --- |
| **Пространства имен могут быть вложенными** |
| **namespace** A  {  **namespace** B  {  **int** val;  **void** f();  }  }    A::B::f(); |

::X - глобальное пространство имен

Анонимные пространства имен используются в тех случаях, когда нужно сделать объявления переменных невидимыми для кода в других файлах (т. е. сделать им внутреннюю компоновку)

Внутренняя компоновка:

|  |  |
| --- | --- |
| file1.cpp namespace  {  int a;  int b:  } | file2.cpp namespace  {  int a;  int b:  } |

# Использование пространств имен

**Директивы включения**

* объявление  
  **using** A::val;  
  val:=10;   
  A::f();
* директива  
  **using** **namespace** A;

val:=10;

f();

Как использовать std cin/cout?

1) Ужасный

using namespace std;

cout<<...;

2) Нормальный

using std::cout;

cout<<...;

3)Хороший

std::cout<<...;

Объединение пространств имен:

|  |
| --- |
| **namespace** A  {  ...  }  **namespace** B  {  **using** **namespace** A;  } |

Если компилятор не может найти функцию, он ищет определение функции в пространствах имен аргументов.

# Генерация исключений. Ключевое слово throw

(нет в лекции)  
Рассмотрим в качестве примера как для класса Vector можно представлять и обрабатывать особые ситуации, вызванные выходом за границу массива:

|  |
| --- |
| **class** Vector {  **int**\* p;  **int** sz;  **public**:  **class** Range { }; // класс для особой ситуации  **int**& **operator**[](**int** i);  // ... }; |

Предполагается, что объекты класса Range будут использоваться как особые ситуации, и запускать их

можно так:

|  |
| --- |
| **int**& Vector::**operator**[](**int** i) {  **if** (0<=i && i<sz) **return** p[i];  **throw** Range(); } |

# Обработка исключений. Ключевые слова try и catch

(нет в лекции)  
(пример продолжается из пункта выше)  
Если в функции предусмотрена реакция на ошибку недопустимого значения индекса, то ту часть функции, в которой эти ошибки будут перехватываться, надо поместить в оператор try. В нем должен быть и обработчик особой ситуации:

|  |
| --- |
| **void** **f**(Vector& v) {  // ...  **try** {  do\_something(v); // содержательная часть, работающая с v  }  **catch** (Vector::Range) {  // обработчик особой ситуации Vector::Range  // если do\_something() завершится неудачно,  // нужно как-то среагировать на это  // сюда мы попадем только в том случае, когда  // вызов do\_something() приведет к вызову Vector::operator[]()  // из-за недопустимого значения индекса  }  // ... } Обработчиком особой ситуации называется конструкция **catch** ( /\* ... \*/ ) {  // ... } |

Ее можно использовать только сразу после блока, начинающегося служебным словом try, или сразу после другого обработчика особой ситуации. Служебным является и слово catch. После него идет в скобках описание, которое используется аналогично описанию формальных параметров функции, а именно, в нем задается тип объектов, на которые рассчитан обработчик, и, возможно, имена параметров. Если в do\_something() или в любой вызванной из нее функции произойдет ошибка индекса (на любом объекте Vector), то обработчик перехватит особую ситуацию и будет выполняться часть, обрабатывающая ошибку. Например, определения следующих функций приведут к запуску обработчика в f():

|  |
| --- |
| **void** **do\_something**() {  // ...  crash(v);  // ... } **void** **crash**(Vector& v) {  v[v.size()+10]; // искусственно вызываем ошибку индекса } |

Чтобы исследовать недопустимое значение индекса, в обработчике нужно дать имя объекту, представляющему особую ситуацию:

|  |
| --- |
| **void** **f**(Vector& v) {  // ...  **try** {  do\_something(v);  }  **catch** (Vector::Range r ) {  cerr << "недопустимый индекс" << r.index << '\n';  // ...  }  // ... } |

# Обработка нескольких исключений

(нет в лекциях)

В общем случае в программе могут возникать разные ошибки на этапе выполнения. Этим ошибкам можно сопоставить несколько типов исключений с различающимися именами. Я предпочитаю для обработки исключений использовать специально предназначенные для этого типы (чтобы предельно ясно выразить свои намерения). В частности, с этой целью я никогда не использую встроенные типы (вроде int). В большой программе не будет эффективных способов удостовериться, что такие обработчики имеют дело исключительно с предназначенными для них ошибками. То есть возможна путаница в обработке ошибок из разных источников.

Программа калькулятор должна обрабатывать два типа ошибок: синтаксические и попытки деления на 0. Нет необходимости передавать обработчику ошибки деления на 0 какую-либо дополнительную информацию , так что этот тип ошибки может быть представлен простейшим (пустым) типом:

|  |
| --- |
| **struct** Zero\_divide { }; |

С другой стороны, обработчик синтаксических ошибок наверняка захочет узнать, каков характер ошибки. В этом случае мы передаем строку:

|  |
| --- |
| **struct** Syntax\_error {  **const** **char**\* p;  Syntax\_error (**const** **char**\* q) {p=q;} }; |

Пользователь синтаксического анализатора может осуществить обработку этих двух исключений, добавив обработчики обоих типов к try-блоку. По ситуации будет выполняться один из них. По выходе из тела обработчика управление передается коду, следующему за самым последним обработчиком в списке:

|  |
| --- |
| **try** {  //...  expr(false);  //сюда попадаем только если expr() не возбуждает исключение  //... } **catch**(Syntax\_error) {  //обработка синтаксической ошибки } **catch**(Zero\_divide) {  //делаем что-то в ответ на попытку деления на нуль } //сюда попадаем, если expr не вызвала исключения или если были исключения //Syntax\_error или Zero\_divide, а их обработчики не изменили потока управления //с помощью return, throw или каким-либо иным способом |

Список обработчиков выглядит почти как оператор switch, только не требуется оператор break. Это отличие в синтаксисе призвано подчеркнуть тот факт, что каждый обработчик образует отдельную область видимости.

Функции не обязаны перехватывать все возможные типы исключений. Например, предыдущий try-блок не имеет обработчика потенциально возможных исключений от операции ввода. Эти исключения просто оставляются для дальнейшего поиска обработчиков (передаются наверх по стеку вызовов функций).

С точки зрения синтаксиса языка исключение считается обработанным сразу же после входа в обработчик. Это сделано для того, чтобы исключения, сгенерированные в теле этого обработчика относились к вышестоящему коду, вызвавшему соответствующий try-блок. Например, в следующем коде нет бесконечного цикла обработки исключений:

|  |
| --- |
| **class** Input\_overflow {/\*...\*/}; **void** **f**() {  **try**  {  //..  }  **catch** (Input\_overflow)  {  //..  **throw** Input\_overflow();  } } |

Обработчики исключений могут быть вложенными:

|  |
| --- |
| **class** XXII {/\*...\*/}; **void** **f**()  {  //..  **try**  {  //..  }  **catch**(XXII)  {  **try**  {  //что-нибудь нетривиальное  }  **catch**(XXII)   {  //код нетривиального обработчика сам упал  }  }  //.. } |

Однако такая вложенность редко встречается в коде, написанном человеком, и вообще-то является плохим стилем.