

Структура языка программирования. Выражения

На прошлых лекциях:

Структура языка программирования

- ✓ **алфавит языка:** кодировка символов; символы времени трансляции, символы времени выполнения;
- ✓ **идентификаторы:** правила образования идентификаторов; зарезервированные идентификаторы; литералы; ключевые слова;
- ✓ **фундаментальные (встроенные) типы данных:** предопределенные типы данных, массивы фундаментальных типов;
- ✓ **пользовательские типы данных:** типы, которые может создавать пользователь; создаются на основе фундаментальных типов, описывается их свойства поведение;
- ✓ **преобразование типов:** явное и неявное (автоматическое).
- ✓ **инициализация памяти:** присвоение значения в момент объявления переменной;
- ✓ **константное выражение:** выражение, которое должно быть вычислено на этапе компиляции;
- ✓ **область видимости переменных:** доступность переменных по их идентификатору в разных частях программы; пространства имен.

План лекции «Структура языка программирования. Выражения»:

- определение выражения;
- понятие побочного эффекта;
- выражения lvalue и rvalue;
- символ окончания последовательности;
- ассоциативность операторов;
- укороченные вычисления;
- типы выражений;
- примеры.

1. Выражения

1) **Выражение** – объединение литералов, имен (переменных, функций и пр.), операторов и специальных символов, служащих для вычисления выражения или достижения побочных эффектов (например: при применении в выражении функций).

- ✓ Выражения состоят из операндов, знаков операций и скобок.
- ✓ Операнды выражения задают данные для вычислений.
- ✓ Операции задают действия, которые необходимо выполнить.
- ✓ Операнд может быть выражением, литералом или переменной.
- ✓ Порядок вычисления выражения с операторами одинакового приоритета не определен.

- ✓ Если при вычислении выражения значение переменной, входящей в это выражение, изменилось, то говорят, что произошел побочный эффект.
- ✓ Побочный эффект является изменением состояния среды выполнения.

Побочный эффект (side effect) возникает, когда функция или выражение изменяет состояние объекта или вызывает другие функции, которые имеют побочные эффекты.

Побочные эффекты могут приводить и к неожиданным результатам, так как C++ не определяет порядок, в котором вычисляются операции с одинаковым приоритетом, порядок вычисления аргументов функции.

Пример:

```
int x = 5;
int value = add(x, ++x);
    // при x = 5
    // результат будет 5 + 6 или 6 + 6?
    // результат зависит от компилятора и порядка, в каком он будет
    // обрабатывать аргументы функции
```

2) *Выражения lvalue:*

lvalue (именующее выражение) – это ссылка на значение; могут использоваться в левой и правой части оператора присваивания.

Имя переменной, ссылка на элемент массива по индексу, вызов функции возвращающей указатель, всегда связаны с областью памяти, адрес которой известен.

lvalue :

- ✓ выражения, непосредственно обозначающие объект;
- ✓ выражения ссылочных типов;
- ✓ результат операции разыменования (*);
- ✓ результат *префиксных* операций ++, --;
- ✓ имя массива;
- ✓ строковые литералы.

Примеры lvalue:

```
char a [10];
int i = 1;
i      —> lvalue; значение 1
++i    —> lvalue; значение 2
*&i    —> lvalue; значение 2
a[5]   —> lvalue; элемент массива a с индексом 5
a[i]   —> lvalue; элемент массива a с индексом i
```

3) Выражения *rvalue*:

rvalue (значащее выражение) – может использоваться только в правой части оператора присваивания (не связано с адресом, связано только со значением; это могут быть литералы, вызов функции, возвращающей значение).

rvalue:

- ✓ выражения, обозначающие временные объекты;
- ✓ результат операции взятия адреса (&);
- ✓ результат *постфиксных* операций ++, —;
- ✓ литералы за исключением строковых;
- ✓ константы перечислений.

Примеры *rvalue*:

10	—>	rvalue
i + 1	—>	rvalue
i++	—>	rvalue

Почему *x++* это *rvalue*, а *++x* *lvalue*?

lvalue/rvalue является свойством выражений.

Выражение *++x* является *lvalue*, с именем *x*.

Выражение *x++* *rvalue* (без имени), поскольку оно дает неименованный результат.

Причина, по которой результат постинкремента является *rvalue*:

- для постфиксного инкремента *создается временная копия*, которая существует только на время вычисления выражения, и не является полноценным объектом. Обратиться по имени можно только непосредственно к результату постфиксного инкремента.
- для префиксного инкремента *увеличивается значение самого объекта*.

Регистры
EAX = 00000001 EBX = 00C74000 ECX = 00000001 EDX = 003F9588 ESI = 003F1055 EDI = 00F3F9F8 EIP = 003F178A ESP

Память 2
Адрес: 0x00F3F9D8 &pst

Память 1 | Регистры
Rvalue.cpp | Rvalue

```

3
4 #include "stdafx.h"
5 #include <iostream>
6
7 int main()
8 {
9
10     int i = 0;
11     int t = 0;
12
13     int psf = i++;
14     int n = ++t;
15
16     system("pause");
17     return 0;
18 }
19
20

```

Дизассемблированный код
Адрес: main(void)

Параметры просмотра

```

int i = 0;
003F175E mov     dword ptr [i],0
int t = 0;
003F1765 mov     dword ptr [t],0

int psf = i++;
003F176C mov     eax,dword ptr [i]
003F176F mov     dword ptr [psf],eax
003F1772 mov     ecx,dword ptr [i]
003F1775 add     ecx,1
003F1778 mov     dword ptr [i],ecx

int n = ++t;
003F177B mov     eax,dword ptr [t]
003F177E add     eax,1
003F1781 mov     dword ptr [t],eax
003F1784 mov     ecx,dword ptr [t]
003F1787 mov     dword ptr [n],ecx

system("pause");
003F178A mov     esi,esp
003F178C push    offset string "pause" (03F6B30h)
003F1791 call    dword ptr [__imp__system (03FA168h)]
003F1797 add     esp,4
003F179A cmp     esi,esp
003F179C call    __RTC_CheckEsp (03F1127h)
return 0;
003F17A1 xor     eax,eax
}
003F17A3 pop     edi
003F17A4 pop     esi
003F17A5 pop     ebx
003F17A6 add     esp,0F0h

```

Контрольные значения 1

Имя	Значение	Тип
t	1	int
psf	0	int
i	1	int
n	1	int
psf	0	int

Префиксная и постфиксная формы операторов инкрементации и декрементации:

`x = 10; y = ++x;` // переменной `y` будет присвоено значение 11

`x = 10; y = x++;` // переменная `y` будет равна 10

// в обоих случаях переменная `x` будет равна 11

Каким будет результат вычисления этого выражения?

`x[i]=i+++1;`

4) Символ окончания последовательности:

Символ окончания последовательности (в языке программирования C++ – это точка с запятой) определяет точку последовательности (sequence point), в которой завершились все вычисления и побочные эффекты.

Точка последовательности – момент времени, когда побочные эффекты вычисленных выражений уже случились, а побочные эффекты следующих в последовательности выражений еще не начались.

Точка последовательности в C++ обозначается символом точка с запятой (;).

Выражения могут иметь двоякое толкование и могут в разных реализациях компилятора вычисляться по-разному. Профессиональные программисты не применяют такие выражения.

Список точек следования в C++:

- ✓ в конце выражения (обычно, она расположена на точке с запятой);
- ✓ после вычисления всех аргументов в вызове функции и до выполнения любых выражений в ее теле;
- ✓ после копирования возвращаемого значения функции и до выполнения любого выражения вне функции;
- ✓ после вычисления первого выражения в $a \&\& b$, $a || b$, $a ? b : c$ или a, b ;
- ✓ после инициализации каждого базового класса и члена в списке инициализации конструктора.

5) Ассоциативность оператора:

Ассоциативность оператора – порядок вычисления (справа налево или слева направо).

Оператор *присваивания* правоассоциативен и вычисляется справа налево:

```
ival = jval = kva1 = lval;
```

Арифметические операции левоассоциативны, например:

```
ival + jval + kva1 + lval;
```

Встроенные операторы C++, приоритет и ассоциативность операторов

6) Примеры.

Примеры выражений, которые могут быть по-разному интерпретированы различными компиляторами:

```
#include "stdafx.h"
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    int i = 0, k = 0, n = 1;
    i = 3, k+=i, n = k+i;
    system("pause");
    return 0;
}
```

i	3
k	3
n	6

```
#include "stdafx.h"
#include <iostream>

int i = 1, k1 = 0, k2 = 0, k3 = 0;
int f(int x) { i = x+i; return i;};

int _tmain(int argc, _TCHAR* argv[])
{
    k1 = f(i)+f(i)+f(i);
    system("pause");
    return 0;
}
```

Значение k1 ??

```
#include "stdafx.h"
#include <iostream>

int i = 1, k1 = 0, k2 = 0, k3 = 0;

int f(int x) { i = x+i; return i;};

int _tmain(int argc, _TCHAR* argv[])
{
    k1 = f(i)+f(i)+f(i);

    system("pause");
    return 0;
}
```

Debugger window: k1 = 14

```
int i = 1, k1 = 0, k2 = 0, k3 = 0;

int f(int x) { i = x+i; return i;};

int _tmain(int argc, _TCHAR* argv[])
{
    k1 = (i = f(i*2))=(i = f(i*3)) =(i = f(i*4));

    system("pause");
    return 0;
}
```

ОТВЕТ:

```
int i = 1, k1 = 0, k2 = 0, k3 = 0;

int f(int x) { i = x+i; return i;};

int _tmain(int argc, _TCHAR* argv[])
{
    k1 = (i = f(i*2))=(i = f(i*3)) =(i = f(i*4));

    system("pause");
    return 0;
}
```

Debugger window: k1 = 60

```
#include "stdafx.h"
#include <iostream>

int i = 10, k = 10, n = 10;

int _tmain(int argc, _TCHAR* argv[])
{
    i = ++i - ++i;
    k = ++k - k--;
    n = ++n - n++;

    system("pause");
    return 0;
}
```


Ответ

```
#include "stdafx.h"
#include <iostream>

int i = 10, k = 10, n = 10;

int _tmain(int argc, _TCHAR* argv[])
{
    i = ++i - ++i;
    k = ++k - k--;
    n = ++n - n++;

    system("pause");
    return 0;
}
```

i	0
k	-1
n	1

```
#include "stdafx.h"
#include <iostream>

int i = 3, k = 3, n = 3;
int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int _tmain(int argc, _TCHAR* argv[])
{
    i = v[i++];
    k = v[++k];

    system("pause");
    return 0;
}
```

Ответ

```
#include "stdafx.h"
#include <iostream>

int i = 3, k = 3, n = 3;
int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int _tmain(int argc, _TCHAR* argv[])
{
    i = v[i++];
    k = v[++k];

    system("pause");
    return 0;
}
```

i	4
k	4
n	3

```

#include "stdafx.h"
#include <iostream>

int i = 3, k = 3;

int _tmain(int argc, _TCHAR* argv[])
{
    i = i++ + 1;
    k = k-- - 1;

    system("pause");
    return 0;
}

```

ОТВЕТ

```

#include "stdafx.h"
#include <iostream>

int i = 3, k = 3;

int _tmain(int argc, _TCHAR* argv[])
{
    i = i++ + 1;
    k = k-- - 1;

    system("pause");
    return 0;
}

```

Имя	Значение
i	5
k	1

```

#include "stdafx.h"
#include <iostream>

int i = 3;

int _tmain(int argc, _TCHAR* argv[])
{
    i = i+=i++;

    system("pause");
    return 0;
}

```

```

#include "stdafx.h"
#include <iostream>

int i = 3;

int _tmain(int argc, _TCHAR* argv[])
{
    i = i+=i++;

    system("pause");
    return 0;
}

```

Имя	Значение
i	7

7) Укороченное вычисление

Укороченное вычисление – вычисление, значение которого может быть вычислено по части выражения.

Таблица истинности для логических операторов:

x	y	x && y	x y	! x
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

```
#include "stdafx.h"
#include <locale>
#include <iostream>

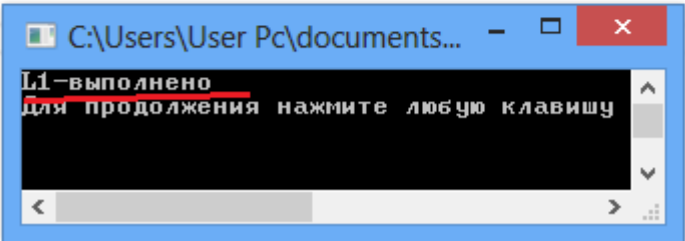
bool L1 ()
{
    std::cout << "L1-выполнено"<<std::endl;
    return true;
};

bool L2 ()
{
    std::cout << "L2-выполнено"<<std::endl;
    return true;
};

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "rus");

    bool b = L1() || L2();

    system("pause");
    return 0;
}
```



```

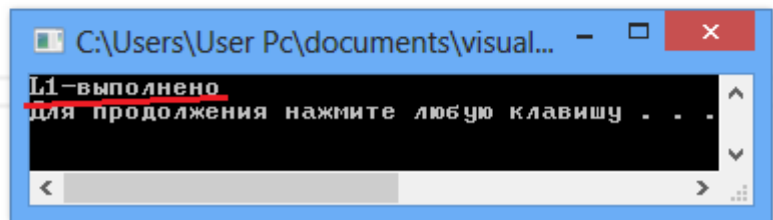
#include "stdafx.h"
#include <locale>
#include <iostream>

bool L1 ()
{
    std::cout << "L1-выполнено"<<std::endl;
    return false;
};
bool L2 ()
{
    std::cout << "L2-выполнено"<<std::endl;
    return true;
};
int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "rus");

    bool b = L1() && L2();

    system("pause");
    return 0;
}

```



```

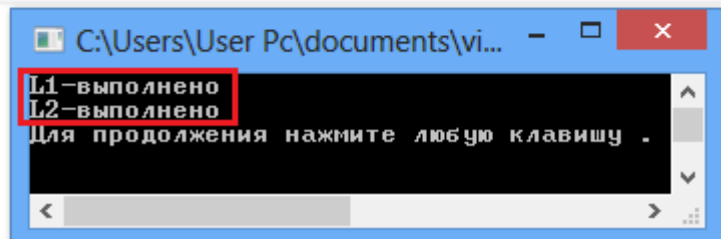
#include "stdafx.h"
#include <locale>
#include <iostream>

bool L1 ()
{
    std::cout << "L1-выполнено"<<std::endl;
    return false;
};
bool L2 ()
{
    std::cout << "L2-выполнено"<<std::endl;
    return true;
};
int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "rus");

    bool b = L1() || L2();

    system("pause");
    return 0;
}

```



8) Типы выражений:

Унарные выражения – выражения с одним операндом.

Унарные операторы действуют только на один операнд в выражении, имеют ассоциативность справа налево.

Примеры унарных операторов:

- оператор косвенного обращения (*)
- оператор взятия адреса (&)
- оператор (+) унарного сложения
- оператор унарного отрицания (-)
- оператор логического отрицания (!)
- оператор дополнения (~)
- оператор префиксного инкремента (++)
- оператор префиксного декремента (--)
- оператор sizeof
- оператор new
- оператор delete

```
#include "stdafx.h"
#include <locale>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "rus");

    int x1 = 1, r = 0;
    int *pr;

    r = ++x1;
    r = --x1;
    pr = &r;
    r = *pr;
    r = -x1;
    r = +x1;

    system("pause");
    return 0;
}
```

```
#include "stdafx.h"
#include <locale>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "rus");

    int xx[] = {1, 2, 3, 4 };
    int *yy;
    int r, l;

    r = xx[2];
    yy = new int[5];

    l = sizeof(yy);

    system("pause");
    return 0;
}
```

```

#include "stdafx.h"
#include <locale>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    short x1 = 1, r;
    float f;
    struct SSS {int k; short n;} s = {1,1};

    SSS* ps = &s;
    r = ~x1;
    f = (float)x1;
    r = s.k;
    r = ps->n;

    system("pause");
    return 0;
}

```

9) Бинарные выражения – выражения с двумя операндами.

Мультипликативные операторы (ассоциативность слева направо):

умножение (*), деление (/), остаток (%)

Аддитивные операторы (ассоциативность слева направо):

сложение (+), вычитание (-)

Операторы сдвига

сдвиг вправо (>>), сдвиг влево (<<)

Операторы отношения и равенства (ассоциативность слева направо):

меньше (<), больше (>), меньше или равно (<=),
больше или равно (>=), равно (==), не равно (!=)

Побитовые операторы

Логические операторы

логическое и (& &), логическое или (||)

```

#include "stdafx.h"
#include <locale>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    int n = 24, ir;
    float f1 = 3.222, f2 = 20.1E-5, fr;

    fr = f1/f2;
    fr = f1+f2;
    fr = f1-f2;
    fr = f1*f2;
    ir = n%7;
    ir = n<<8;
    ir = 8>>n;

    system("pause");
    return 0;
}

```

```

#include "stdafx.h"
#include <locale>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    int n = 24, ir;
    float f1 = 3.222, f2 = 20.1E-5, fr;

    bool b;

    b = f1 < f2;
    b = f1 <= f2;
    b = f1 > f2;
    b = f1 >= f2;
    b = f1 == f2;
    b = f1 != f2;

    system("pause");
    return 0;
}

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    char c1 = 0x0101, c2 = 0x1010, c3;
    bool b1 = true, b2 = false, b3;

    c3 = c1|c2;
    c3 = c1&c2;
    c3 = c1^c2;
    b3 = b1&b2;
    b3 = b1||b2;

    system("pause");
    return 0;
}

```

10) Выражения: присваивание

Присваивание стирает старое значение из переменной, затем записывает в нее новое значение.

```
#include "stdafx.h"
#include <locale>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    int n1 = 1, n2 = 2, n3 = 3;

    n1 = n2;

    n1 += n2;

    n1 -= n2;

    n1 *= n2;

    n1 /= n2;

    n3 %= n2;

    n1 >>= 3;

    n2 <<= 2;

    n1 &= n2;

    n1 |= n3;

    n2 ^= n1;

    system("pause");
    return 0;
}
```

11) Выражения: запятая

Оператор запятая (,) – оператор последовательного вычисления:

`x = (y = 3, y = 1);` // чему равен x?

```
int _tmain(int argc, _TCHAR* argv[])
{
    int n1 = 1, n2 = 2, n3 = 3, r1, r2;

    r1 =  n1+=3, n2 *=3, n3/=2, n1+n2+n3;    // r1 = ?

    r2 =  (n1+=3, n2 *=3, n3/=2, n1+n2+n3);  // r2 = ?
}
```



```

int _tmain(int argc, _TCHAR* argv[])
{
    int n1 = 1, n2 = 2, n3 = 3, r1, r2;

    r1 = n1+=3, n2 *=3, n3/=2, n1+n2+n3;
    r2 = (n1+=3, n2 *=3, n3/=2, n1+n2+n3);

    system("pause");
    return 0;
}

```

Оператор запятая (,) имеет самый низкий приоритет среди всех операций языка C++ (*ассоциативность слева направо*). У этой операции может быть 2 и более операндов. Вначале вычисляется первый операнд, затем второй и так далее, а в качестве результата возвращается последний операнд.

```

func_one( x, y + 2, z );    // в функцию передаются 3 параметра: x, y + 2 и z
func_two( (x--, y + 2), z ); // в функцию передаются 2 параметра: y + 2 и z

```

func_two: первым параметром является результат операции «запятая», т.е. значение выражения $y + 2$

12) Выражения: тернарный оператор

Тернарный оператор (?) принимает три операнда:

- первый операнд неявно преобразуется в **bool**;
- если первый операнд имеет значение **true** (1), то вычисляется второй операнд;
- если первый операнд имеет значение **false** (0), то вычисляется третий операнд.

Условные выражения имеют ассоциативность справа налево.

```
#include "stdafx.h"
#include <locale>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    int n1 = 1, n2 = 2, n3 = 3, r1;

    r1 = n1 > n2? n1:n3;

    system("pause");
    return 0;
}
```

4) Выражения: перегружаемые операторы

Перегрузка операторов в программировании – это реализация в одной области видимости нескольких различных **вариантов** применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Синтаксис:

<тип> operator <символ_оператора> (список_параметров)

Перегруженные операторы реализуются в виде функции, например:

```
#include "stdafx.h"
#include <locale>
#include <iostream>

enum SERVER{STARTING, SHOOTDOWN, WORK, STOP};
enum CLIENT{CONNECT, SEND, RECV, DISCONNECT};
enum EVENT {WAIT,DO, ERROR};

EVENT operator>>(CLIENT c, SERVER s)
{
    EVENT rc = ERROR;
    if ((c == CONNECT || c == SEND || c == RECV)&&(s == STARTING || s == STOP)) rc = WAIT;
    else if ((c == CONNECT || c == SEND || c == RECV)&&(s == WORK)) rc = DO;
    return rc;
};

int _tmain(int argc, _TCHAR* argv[])
{
    EVENT e1 = CONNECT>>STARTING;
    EVENT e2 = CONNECT>>WORK;
    EVENT e3 = SEND>>SHOOTDOWN;
    system("pause");
    return 0;
}
```

e1	WAIT (0)
e2	DO (1)
e3	ERROR (2)

```

#include <locale>
#include <iostream>

struct POINT { int x; int y; };
struct VECTOR { int x; int y; };

VECTOR operator-(POINT p1, POINT p2)
{
    VECTOR rc = {p1.x-p2.x, p1.y-p2.y};
    return rc;
};

VECTOR operator+(VECTOR v1, VECTOR v2)
{
    VECTOR rc = {v1.x+v2.x, v1.y+v2.y};
    return rc;
};

int _tmain(int argc, _TCHAR* argv[])
{
    POINT p1 = {10, 20}, p2 = {15, -10};
    VECTOR v1, v2, v3;

    v1 = p1 - p2;  {x=-5 y=30 }
    v2 = p2 - p1;  {x=5 y=-30 }
    v3 = v1 + v2;  {x=0 y=0 }

    system("pause");
    return 0;
}

```