// Кирилл Юрьевич Богачев 3 занятие

Мы рассмотрели функции printf и scanf, а также заметили, что в таком виде scanf использовать нельзя - необходимо проверять возвращемое им значение. Для этого, в частности, нужно использовать условный опреатор:

```
if (<выражение>) <оператор1>;

if (<выражение>) <оператор1>
else <оператор2>;
```

В качестве операторов могут выступать все изученные операторы, в т.ч. блочный оператор.

// Чаще всего даже один оператор указывают внутри блочного, расставляя фигурные скобочки на ОТДЕЛЬНЫХ строчках, ОЧЕНЬ сильно повышает читаемость кода

Порядок выполнения условного оператора:

1. Вычисляется <выражение>
2. <выражение> сравнивается с 0. Если не равно 0, то выполняется <оператор1>. Если равно 0, то если имеется else, то выполняется <оператор2>

```
int x = 0;
if(x); // все хорошо

double y = 1;
if(y) // сравнение вещественных чисел не является безопасной операцией
("comparing floating-point with ‘==’ or ‘!=’ is unsafe")
```

Так происходит, потому что вещественное число округляется к ближайшему представимому: сравнение побитовое - если есть хоть один не равный 0 бит, то число не ноль - почти всегда не то, чего вы хотите.

```
|y| < eps // правильный способ спросить "y - ноль?"

a<b // никаких проблем, потому что операция не побитовая, берется знак (первый бит) разницы (побитовой)
```

Для адекватного пользования условным оператором также необходимы бинарные операторы сравнения:

```
<выражение1> < <выражение2>
>
>=
<=
==
```

// ошибка начинающих: написать => вместо >=
"Меньше Равно", "Больше равно", как говорим - так и пишем, без симметрии на письме (знаки <= и >= несимметричные)

```
int x, y;
if (x == y) // ОК

double a, b;
if(a==b) // ошибка сравнения вещественных |a-b|<eps
```

Для корректного сравнения делают eps зависимой от амплитуды a и b, плотности вещественных чисел в этом диапазоне:

```
|a-b|<eps(|a|+|b|) // проверка на равенство

|a-b|>eps(|a|+|b|) // проверка на неравенство
```

ВАЖНО: при использовании >= и <= произойдет та же ошибка компиляции, что при сравнении через ==, только строгие неравенства безопасно сравнивают вещественные числа

Тип результата любой логической операции - int // в C не реализована логика отдельным типом (как bool в C++)

Побитовое сравнение и арифметика зависит от поведения процессоров - процессоры ведут себя так еще, боюсь, до рождения ваших родителей.

Почему 2 знака равенства? Во многих языках сравнение - одинарный знак равенства. Различие сильнейшее:

= // операция присваивания, имеет тип и возвращаемое значение в языке C

Во многих языках это оператор - без возвращаемого значения

Тип - тип левого значения
Значение - значение левого выражения
Правое значение не используется

```
a = b;
a = b = c;

// Почти единственная операция, которая трактуется справа налево (обратная ассоциативность)

a = b + c;
(a = b) + c; // ранее такая запись экономила значение в регистре, но сейчас уже не актуально и снижает читаемость кода - компиляторы уже сами используют регистры для оптимизации лучше, чем это делает человек
```

Почему снижает читаемость кода: выглядит как арифметическое выражение, но на самом деле в ходе его вычисления меняется значение a
Т.е. так писать можно, но не нужно

Компиляторы - очень сложная программа, все из текущих на рынке разрабатываются уже лет по 30

```
if(x=y) // в x идет y, используется значение x (т.е. y)
```

Такая запись - почти всегда не то, что вы хотели. Увидев единственный знак равенства, компилятор даже не выполняет - сразу ругается: "suggest parentheses around assignment used as truth value"

Иногда это логично с точки зрения прочтения кода, но чаще всего нет, и компилятор не компилирует - его можно заставить выполнить такой код вручную, если обернуть подозрительное выражение в отдельные скобки:
if((x=y)) // такую запись компилятор съест и прокомпилирует

// Пример грамотной реализации scanf:

```
int printf(const char* s, ...);
int scanf(const char* s, ...);

int main(void)
{
	double a, b;
	printf("Input 2 values:");
	if (scanf("%lf%lf", &a, &b) != 2)
	{
		printf("Cannot read! \n");
		return 1; // переход к тому, кто нас вызвал, для main - к операционной системе
	}
	printf("%f+%f=^f\n", a, b, a + b);
	return 0;
}
```

// scanf в случае успеха возаращает количество введенных полей (указанных в аргументе format)
// else здесь усложняет чтение кода: если случился return, то else излишен

// Способ запуска (эмуляция консоли в тексте):

```
$gcc file.c
$./a.out
1
2
2+3 = 3
```

$echo $? // ? - специальная переменная окружения, которая показывает статус возврата текущей функции

$echo 1 2 // напишет 1, 2 - echo выводит свой аргумент на строку

$echo "Hello" // выведет Hello

У переменных спецсимвол - доллар - так принято на командной строке. Таким же символом обозначается начало командной строки.

Можно вывод echo подавать на вход программы:

$echo 1 2 | ./a.out

| это pipe - от слова конвейер

Когда работает мастер, срабатывает двойной человеческий фактор: сначала мастер вводит неправильные значения, а потом не может понять, откуда появился неправильный результат - основное достоинство конвейерного запуска через echo в сохраняющейся истории связанных друг с другом аргументов и выводов через стрелку вверх - все входные и выходные данные собраны вместе, их удобно отследить и прочитать при отладке программы. Это очень хороший процесс автоматизации при работе с консолью.

Унарные операции:

Те операции, которые имеют 1 аргумент :
<знак операции><выражение>

1) Не все операции применимы над любыми операндами
2) Тип результата
3) Значение

- // применима к любым типам
+ // ничего не делает
!<выражение> // операция отрицания: не 0, если выражение 0, 0, если выражение не 0
// не 0 - реально никто не знает, что это, просто не 0 - в ABI (Application Binary Interface) - могут быть все биты 1..1 - зависит от системы запуска - может быть 1, -1 (90% случаев)

double a;
!a // предупреждение - побитовые выражения с вещественными числами это сравнение с нулем на равенство/неравенство

~<выражение> // только для целых, тип тот же, значение - каждый бит инвертируется 0->1 1->0

unsigned int x = 0;
~x;

~0 == 2^32-1 // 4 байта - 32 бита - на один int

~x+1 // унарные операции имеют приоритет выше бинарных

Дальше все зависит от того, включены ли арифметические исключения (в компиляторе у К. Ю. все включено). Если включены, то компилятор подаст сигнал floating point exception - тот же сигнал, что при ошибке для чисел с плавающей точкой, отдельного для этих регистров нет. Это исключительная ситуация переполнения. Если не включены исключения, то оно "завернется".

& //взятие адреса - уже смотрели - применим ко всем объектам, имеющим адрес, так называемым lvalue(left value) - то, у чего можно поменять значение
&1, &(1+2) // ошибка

Тип - указатель ячейки памяти, откуда взяли адрес, операнд должен иметь связанную ячейку памяти

* // то же самое, все уже знаем

sizeof(<выражение>); // возвращает размер в байтах

Имеет тип size_t:

long long unsigned // в 64 битных
int // в 32

Оператор явного приведения:

(<тип>)<выражение>; // операция приведения типа

Тип - <тип>
Значение - значение <выражение>

Преобразования явно вещественных, где мы этого хотели:

int x = 1;
double y = 2.3;
x = y; // предупреждения от компилятора за переполнение и падение при включенном предупреждении арифметических исключений

x = (int)y; // компилятор счастлив от того, что вы были уверены в своих действиях (но если прийдет 1..1 - то чуда не произойдет, программа уронится от плохих значений, не влезающих в тип или вызывающих ошибку)

Бинарные операции:
<выражение1><знакоперации><выражение2>

<выражение1><выражение2> одного типа - тип результата
Если разные, то работает правило неявного преобразования типа

Тип, представляющий меньший диапазон, приводится к типу, представляющему больший диапазон

int i = 1;
double x = 2;
i <операция> x; // имеет тип double

Бинарные операции сравнения
<
>
<=
>=
==
!=
Тип результата - всегда int

Бинарная операция присваивания =
Тип результата - всегда левое выражение

Для арифметических операций правило неявного преобразования

Арифметические операции: у компилятора нет проблем отличать унарные операции от бинарных. Это у вас проблема...
+
-
*
/

Приоритет операций - стандартный из алгебры

Порядок выполнения для всех бинарных операций, кроме явно указанного, - справа налево, порядок вычисления коммутативных аргументов не определен
a+b - не значит, что в начале вычисляется a или b
a+b*c // он сам определяет, сначала умножать или складывать

Порядок операций равного приоритета определяется компилятором, он соблюдает математические правила, но реализовывает их так, как наиболее эффективно с точки зрения кода

```
int i = 1, j = 2;
double x = 3., y = 4;

i+j+x // медленный регистр с плавающей точкой, компилятор может сначала сложить отдельно в целом регистре,
(i+j)+x // не меняет порядок действий компилятора
i+(j+x) // не влияет на решение
j+x+i // тоже не влияет

// скобки у операций равного приоритета удаляются
// скобки, которые не меняют приоритет математически, игнорируются

x*i/j // если компилятор захочет выполнить справа, то i/j == 0, x*0 == 0
Здесь получится либо 1.5, либо 0 в зависимости от прихоти компилятора
```

Смешивание объектов разных типов в операции равного приоритета - коммутативных, перестановочных - преступление, т.к. не гарантируется порядок выполнения слева направо (потому что это не эффективно, в таких языках, где гарантирован порядок выполнения слева направо, программисту приходится думать над эффективностью, а компилятор это делает лучше, чем человек: компилятор располагает информацией о том, кто в каких регситрах находится, вы такой информации не располагаете, люди находятся в неравных условиях относительно компилятора)

Поставить скобки - не работает - первое, что делает компилятор - это удаляет скобки :)

Достаточно явного приведения типа (достаточно обойтись одним приведением):
i + j + x

Порядок не гарантируется!!
Деление приводит к катастрофическому изменению ответа

```
int i = -3, j = 2;
i/j // -1
```

С берет целую часть не так, как это делается в математике - С отбрасывает дробную часть с любым знаком - это максимально быстрая операция, которая выбрасывает все лишнее

СЛЕДУЮЩИЕ ОПЕРАЦИИ ОПРЕДЕЛЕНЫ ТОЛЬКО ДЛЯ ЦЕЛЫХ ЧИСЕЛ
(int и иже с ними)

Бинарные операции: %
Только для целых операндов - остаток от деления: поделили, получили целую часть и дробную, которую раньше отбрасывали
-3%2 // ответ -1, остаток тоже не математический, а взятый из положительных чисел с отрицательным знаком

<< битовый сдвиг влево

```
unsigned int i = 3;
```

Операция побитового сдвига - это умножения на 2
i<<j // i*(2^j) лучше так не думать, потому что это не всегда так, к примеру при переполнении единички слева уходят, к тому же приоритет этой операции меньше, чем у мультипликативной
i<<j+1 <=> i<<(j+1)

Неудачный приоритет операции

Сдвиг побитового представления вправо
>>

Первый бит (отвечающий за знак) не трогается при побитовом сдвиге, его как будто нет, сдвигаются все оставшиеся биты, те, которые выходят за границу, удаляются

Операцию деления иначе как в столбик человечество реализовывать не умеет, это занимает около 100-200 тактов процессора.
<< занимает 1 такт процессора.
Процессора делает миллиард тактов в секунду.
Таким образом, степени двойки получают мгновенное представление на процессоре.

Побитовые операции:
& побитовая "и"
| побитовая "или"
^ побитовая "xor"

```
int i, j;
```

Таблица истинности для побитовых операций:
биti битj i&j i|j i^j
0 0 0 0 0
0 1 0 1 1

1 0 0 1 1
1 1 1 1 0

Пример выполнения:
i = 3 011 3
j = 5 101 5
i&j 001 1
i|j 111 7
i^j 110 6

взять jй бит i
i>>j // j бит в 1 позицию

(i>>j)&1U

Остаток от деления на 2:

i%2 // 200 тактов (компилятор умный, не станет так делать)
i&1U // остаток от деления на 2
i&7U // остаток от деления на 8
i&((1<<j)-1)U // остаток от деления на 2^j

Менять конкретные биты:

i&1U //1 на последнем бите
i&(~1U) // 0 на последнем бите
i|((i<<j)-1) // j бит на 1
i&(~(i<<j)-1) // j бит на 0

Логические операции:
Все то же самое, только они <выражение1> <операция> <выражение2>
&&
||
^^

Таблица та же самая, даже повторяться не хочется.

i = 1, j = 2
i&j == 0
i&&j == не0 //мб все биты по 1, мб 1, мб -1

// ВАЖНО СЛЕДИТЬ за количеством амперсендиков, чтобы не перепутать операции && и &, как это случилось на семинаре

if(i&j) // на последних версиях выдает ошибку в арифметике в качестве условного выражения

Математическое образование деградирует, поэтому компилятор потерял доверие к вашим математическим способностям.
При попытке написать: a>b && a>c || a!=0 // компилятор говорит "поставьте скобки, я в вас не уверен"

Краткие операции присваивания:
a = b;

```
a += a + b; <=> a += b; // облегчает создание кода, 1 формирование адресного выражения
a -= a - b; <=> a -= b;
a *= a * b; <=> a *= b;
a /= a / b; <=> a /= b;
a %= a % b; <=> a %= b;
a &= a & b; <=> a &= b;
a |= a | b; <=> a |= b;
a ^= a ^ b; <=> a ^= b;
a ~= a ~ b; <=> a ~= b;
И так далее...
```

Префиксная и постфиксные формы (интересная мнемоника):

```
a = a+1;
a += 1;
a++; // здесь в качестве значения сначала используется a на месте этого выражение, и только после его использования a увеличивается на 1
++a; // здесь сначала a увеличивается на 1, потом в качестве значения используется a+1 на месте этого выражения

a = a - 1;
a -= 1;
a--; // здесь в качестве значения сначала используется a на месте этого выражение, и только после его использования a уменьшается на 1
--a; // здесь сначала a уменьшается на 1, потом в качестве значения используется a-1 на месте этого выражения
```

Основное правило - никто никогда не использует значения этих выражений, потому что тогда все становится нечитаемым - как правило, это счетчики циклов

Инкремент - арифметическая операция a+a++ // оторвать руки - непонятно, что это такое

Приоритеты операций(от высшего к низшему):

```
()

+
- ! ~ *

sizeof (приведение типа)

* / %

+ -

<< >>

<> <= >= == !=

& ^ |
```

&& ^^ ||

=
*= /+ ...

,

Оператор запятая - способ записать вместо одного выражения два - значение берется от второго (первое вычисляется, результат выбрасывается, второе вычисляется, возвращается его значение)

x = (a=b, c) // здесь в x идет c, в a идет b, это своеобразный способ заинициализировать много переменных в одну строку

Оператор цикла:

for([<выражение1>];[<выражение2>];[<выражение3>]) <оператор>; // в квадратных скобках - то, чего может не быть
for(;;); // можно вообще ничего не подставлять в качестве выражений или оператора

Порядок выполнения условного оператора:

1. вычисляется <выражение1> если есть
2. вычисляется <выражение2> если есть
3. если <выражение2> не 0, то
   а) выполнить <оператор>
   б) выполнить <выражение3>, если есть
   в) вычислить <выражение2>, если есть

Эти выражения называются выражениями начальных значений, условия, приращения

Пример использования условного оператора:

```
int f(int n)
{
	int s = 0, i;
	for (i = 1; i <= n; i++)
		s += i * i;
	return s;
}
```

// это цикл с предусловием

Побитовый вывод числа:

```
int printf(const char*, ...);
void printf_bits(unsigned int x)
{
	int len = sizeof(x) * 8; // длина аргумента в битах
	int i;
	for (i = len - 1; i >= 0; i--)
```

```
	{
		printf("%d", (x >> i) & 1U);
	} // фигурные скобки, потому что так принято, можно было без них
	pritnf("\n");
}
```

Первое домашнее задание сегодня. Условия будут присланы. Выполнить до вечера следующей пятницы.

На kirill.yu.bogachev@gmail.com необходимо прислать свой email, ФИО и номер группы.