// Кирилл Юрьевич Богачев 2 занятие

В прошлый раз мы описали типы хранения данных в памяти

Константы - прежде всего важен тип (ниже перед выражениями константы указывается её тип)

Символьные константы:

char 'a' // символьный тип занимает 1 байт, поэтому при приведении к типу int, такое выражение будет интерпретировано как число от 0 до 255 - его номер в таблице ASCII (American standard code for information interchange)

Также одним символом считаются т.н. "escape-последовательности", состоящие из обратной косой черты (\) и определенного символа после неё:

char '\n' // переход на следующую строку
char '\r' // возврат каретки (ставит курсор в начало текущей строки)
char '\t' // переход на следующую позицию табуляции - табуляция определяется текущим эмулятором терминала, редко используется
char '\f' // переход на следующую страницу (прокрутка одного экрана)
char '\a' // звоночек (в современных терминалах, как правило, моргает, а не озвучивается, звук всех раздражает)
char '\кодсимвола' // выводит символ с таким номером из таблицы ASCII
char '\010' // вывод символа из ASCII по номеру в восьмеричной системе исчисления
char '\\' // т.к. обратная черта является специальным символом, с которого начинаются все escape-последовательности, то вывод обратной черты - это тоже escape-последовательность

012 // 8-ричная 0 - octo
0x12 // 16ричная
0b12 // 2ичная

В ASCII (American standard code for information interchange) изначально были пронумерованы 255 символов - для латинского алфавита этого было достаточно

Лучше использовать мнемоники вместо кодов

Вместо явного приведения типов можно указывать расширение типа числовой константы сразу после её выражения:

12U - константа типа unsigned int
12L - константа типа long int
12UL - константа типа unsigned long int

-1U == 2^32 -1 (следует из побитовой формы записи)

Формы записи чисел с плавающей точкой:

double 1.23
double 12.3e-1

double 0.123e1
double 0.123e+1

Удивительный факт: float работает медленнее, чем double, раньше процессор не умел обрабатывать float, после появления модулей векторных преобразований процессоры научились работать быстрее с float, чтобы иметь некоторое ускорение, но преобразование между float и double все равно долгое, короче все продолжают использовать double вместо float

1 53 10 бит - запись числа с плавающей точкой (1 бит на знак, 53 на мантиссу, 10 на порядок)
Поэтому мультипликативные операции идут без потери данных

Очень важно значение eps = 10^-16 // расстояние от 1 до ближайшего числа

Сложение столбиком в 53 битах (при сложении сначала числа приводятся к одному порядку, затем складываются поразрядно в 53 битах мантиссы):
1.0 ... 0
0.0 ... 01
=
1.0 ... 0

Поэтому 1+eps=1 // минимальная машинная точность

Почему расстояние от 1 (удобно масштабировать равенство для вычисления машинной точности в разных порядках):

1 + 10^-16 = 1 => 10^20 + (10^20 * 10^-17) = 20 ^ 20 => Для 10 ^ 20 расстояние до ближайшего числа равняется 1000

В дискретизации числа на вещественной оси распределены неравномерно: Около нуля самая высокая плотность машинных чисел, плотность падает к границам интервала - вещественная ось заполнена дискретизацией чисел неравномерно, ближе всего числа в машинной памяти у нуля, и различимые расстояния между ними (машинная точность) растет по мере увеличения их по модулю

При умножении погрешность не появляется(отдельно умножаются мантиссы, отдельно складываются порядки)

При аддитивных операциях имеют место математически неверные, но считающиеся истинными при выполнении выражения такого вида:
a + eps(a) = a
Где eps(a) - машинная точность, зависящая от величины a по модулю

Целочисленные константы:
const int x = 12;

"Пустой" тип данных:
void y; // нельзя описать, поскольку тип данных не имеет размеров ("error: variable or field ‘x’ declared void")

// Комментарии:

```
/*
...
...
...
*/

/* символ /* был выбран неудачно - такая конструкция иногда появляется в
естественной ситуации */
```

Пример: a/*p (a поделить на значения указателя p, хотя компилятор прочитает как "a, комментарий: p")

Переменная, содержащая адрес другой переменной называется указателем
Указатель - переменная, содержащая адрес памяти, в котором хранится другая переменная

```
int x; // имеет адрес - ячейку памяти, в которой она лежит на протяжении
своего времени жизни
// адрес называется указателем
```

Способ объявить указатель:
<тип>* <имя>; // указатель <имя> на тип <тип>

```
int x = 1; // переменная
int* y; // указатель

*a // унарная операция - разрешение указателя
sizeof(<имя>) // Возвращает целое число, равное размеру типа данных
переменной <имя> в байтах
sizeof(<тип>) // Возвращает целое число, равное размеру типа данных <тип>
в байтах
```

sizeof - операция, в каком-то смысле унарная (оператор, отоличающийся от остальных в Си тем, что выполняется на этапе компиляции - на месте в программе подставляется в виде константы)

```
sizeof(int*) // Вернет 8, т.е. указатель на целые числа может хранить
2^64-1 адресов памяти

sizeof(double*) // 8 в 64 битной
sizeof(double) // 8 в 64 битной
```

Разрешение указателя - это унарная операция:

```
*p // *p - то, что лежит по адресу *p, имеет тип <тип>, если p имел тип
<тип>* (значение указателя имеет тот тип, на который этот указатель
указывал)

// с указателями связано больше всего ошибок: попытка сослаться на
невыделенную ячейку памяти, ошибка в арифметике, из-за которой
перезаписываются регистры на стеке, ломается таблица выделения памяти,
программа падает
```

Пример конфликта комментариев с указателями:

```
int x;
int* p;
x = *p;

x/*p;
```

Поэтому в большинстве стандартов принято любые знаки бинарных операций отделять пробелами

Унарная операция & - операция взятия адреса:

```
<тип> x;
&x;

&1  // ошибка
& (x + 1) // ошибка
& (x + 0) // ошибка
& x + 0 // & имеет приоритет больше арифметических
```

Значение - это то, что лежит по адресу переменной

Унарные операции * , & находятся на вершине приоритета выполнения операций

```
int x = 1;
int* p = &x;
*p = 2; // то же самое, что написать x=2;

y = *p + 1;

y = 2**p; // 2 умножить на *p
```

Разбор выражений : компилятор идет слева направо и пытается взять токен(неделимый элемент)

```
2 * (*p);
```

Простое правило : не надо так писать, писать нужно так, чтобы понятно было сразу вам, компилятор знает все правила, в отличии от вас
Чаще всего то, как вы ожидаете исполнение выражения и то, как его интерпретирует компилятор - различаются

Человечество не научилось возводить в степень иначе, как:
x^ y = exp(ylnx);

Поэтому операция возведения в степень достаточно дорогая

```
int x = 1;
double* y = &x;

// компилятор выдает предупреждение, которое нельзя игнорировать:
```

В x 4 байта, в y 8 байтов, x перезаписывается в начало побитовой записи y - число изменяется

В C++ это выдаст ошибку - такая запись никогда не сработает

```
char c = 'a'; // объявление символьной константы, равной 'c'

const char* s = "ab"; // объявление константной строки со значением "ab"
```

(строка в си - указатель на адрес первого символа в строке, по адресам s+1, s+2 ... s+(n-1) от которого лежат следующие символы строки, а по адресу s+n лежит '\0' - символ конца строки (это не '0', а специальный символ с отдельным кодом в ASCII))

Конец строки отмечается символом с нулевым кодом
'a' 'b' '\0' - каждый по 8 байт // длина - 2, байтов - 3

Длина строки - число символов в ней
Число байт на хранение - на 1 больше, чем длина

// также строки бывают динамическими: когда происходит создание указателя, выделение и освобождение памяти под строку (через malloc и free)

ФУНКЦИИ:

Для описания функции заранее необходим прототип
Там, где написано тело функции - её реализация - определение функции

<тип возвр значения> имя функции(список типов аргументов); // декларация, точка появления - позволяет использовать адрес с функцией

Имя функции - идентификатор (см. опредление прошлой лекции)
(последовательность символов, начинающаяся с буквы или с _)

Целочисленные значения возрващаеются в целочисленном регистре процессора
Потому возврат целого значения - очень легкая операция, которая почти ничего не стоит

void
Подпрограмма - функция без возвращаемого значения

регистр с плавающей точкой, любой из стандартных типов - возврат бесплатно

Вообще нет никаких накладных расходов на образование возвращаемого значения

Хотя вызов функции это все равно очень медленно

Нет аргументов - тоже надо писать void, неудачное соглашение:
() - C++
(void) - в C

() - А это кто? // любое количество аргументов любого типа, не реализовано ни в одной библиотеке

За 20 лет никто ни разу не реализовал (мы можем быть первыми :)

(...) // тоже любое количество аргументов любого типа, но ПОСЛЕ именованных аргументов любого типа

Примеры прототипов:

```
int f1(int);
void f2(int);
int f3(void);
void f4(const char *);
void f5(const char *, ...); // функции с переменным количеством аргументов, мы почти не будем использовать
```

В прототипе можно написать имена переменных для удобства живых читателей, чтобы было понятно, за что отвечают указанные типы для ввода // компилятор их игнорирует, это для придания смысла тексту в коде
Примеры:

```
int f1(int EtoCeloeChisloNaVvod);
void f2(int ThisIsIntegerNumberForInput);
int f3(void); // тут нельзя написать имя, компилятор не пропустит
void f4(const char * OgoStroka);
void f5(const char * WowString, ...);
```

Реализация функции:
копия прототипа + имена аргументам + тело

```
<тип> <имя> (<тип1> <имя1>, ...)
{

} // 1 блочный оператор, который является реализацией функции
```

```
double f(double x, double y)
{
      return x + y;
}
```

// те имена, что написаны в прототипе, никак не связаны с реализацией функций, в них вообще может не быть ничего написано, могут совпадать имена, могут не совпадать - без разницы, имена из прототипов это комментарии для читателя, имена из реализации - названия переменных аргументов

При вызове функции выделяются ячейки памяти под её аргументы, преобразуется все внутри скобок к типам аргументов, им присваиваются переданные значения (передача через копию)

Порядок вызова функции:

1) создать временные ячейки под аргументы (время жизни аргументов - от момента вызова до возврата значения или конца подпрограммы)
Стек - область памяти, где размещаются все локальные переменные, регистры вызова, возвращаемые значения

Неявное использование при вызове функций - куда нужно вернуться (при рекурсии происходит переполнение стека - когда в него не влезают все адреса, куда возвращать значения функций)

2) Скопировать значения переданных (фактических) аргументов во временные ячейки памяти

3) Сохранить адрес возврата (следующая инструкция после вызова - переход на следующую строку)

4) Переход к выполнению функции

Имя функции является константным указателем на начало функции

5) Начало работы функции, появление аргументов (начало времени их жизни), идентификация аргументов по номеру (то, в каком поряде они были перечиселн на вызове - то, в каком порядке они будут присвоены)

ABI - Application Binary Interface // договоренность между компилятором и операционной системой, чтобы функции могли взаимодействовать между собой по адресам вызова и возврата

6) Результат складывается туда, где его ожидают - либо целый регистр, либо вещественный
Другие структуры - отдельные соглашения (несколько регистров или возврат через память)

7) Возврат по сохраненному адресу (из пункта 3)

8) Удаление памяти в пункте 1

9) Использование возвращаемого значения

Что наши функции, что библиотечные, живут по одинаковым правилам
Все аргументы передаются по значению (т.е. передается копия аргумента)

Если нужно что-то другое, кроме передачи через копию, то вы должны сделать это самостоятельно
Есть языки, отдельно выделяющие как синтаксическую конструкцию передачу по значению и по ссылке

```
void f(int x)
{
      x = 1;
}

int  main(void)
{
      int x = 2;
```

```
	f(x);
}

// при вызове функции создается временная ячейка памяти, в неё
записывается 1, и по выполнении функции ячейка памяти удаляется -
переменная икс не меняет своего значения

void f(int**)
{
	*x = 1;
}

int main(void)
{
	int x = 2;
	f(&x);
}

// по завершении x будет изменен
```

Если хотим изменить объект - то передаем его адрес
Еси передаем адрес, чтобы работало быстрее, то пишем const:

```
int f(const char* s)
{ }

// подчеркивает, что передали указатель не для того, чтобы менять, а
чтобы передать то, что там лежит

double f(double a, double b)
{
	double c = a + b;
	return c * c;
}

int main(void)
{
	double x = 1, y = 2, z;
	z = f(x, y);
}
```

Для написания более содержательных программ неплохо было бы вывести результат вычислений, а еще уметь их вводить: для этого есть стандартные библиотечные функции:

Ввод данных:

```
int printf(const char* format, ...); // format - как интерпретировать
свои аргументы

printf("Hello!\n"); // выводит на экран - на стандартное устрйоство
вывода
```

Работа:
1) Читает символы строки format

2) Если символ не %, то вывести (с учетом символов \n \t ...)
3) Если есть %, то
    а) читает следующий за % символ

        если это d (%d) %u unsigned
        если это i (%i)
        если это o (%o)
        если это x (%x)
    то
        значит, есть еще один аргумент
        он имеет тип int // для взятия аргумента надо знать его длину (8 байт, 4 байт)
        его надо вывести на экран как целое
            %d, %i - decimal, integer - по основанию 10 // синонимы для приятности в восприятии
            %o - по основанию 8
            %x - по основанию 16

        если это c
            1) есть еще один аргумент
            2) он имеет тип char
            3) вывести как символ

        если это s
            1) есть еще один аргумент
            2) он имеет тип const char*
            3) вывести как строку

        если это f %f // уже при си нельзя передать как аргумент float - стек - правило выравнивания, нельзя положить произвольное число, только double, хотя функция получает по значению
            1) есть еще один аргумент
            2) он имеет тип double
            3) вывести на экран
             - 1.234567 // 6 знаков после запятой (стандарт)
            -10000.1234556

        если это e % e
            1) есть еще один аргумент
            2) он имеет тип double
            3) вывести на экран в экспоненциальной форме
                -1.000012e+4

// float неявно преобразуется в double - даже при передаче константы она все равно во временную ячейку памяти преобразуется как переменная

        если это z %2
            1) есть еще один аргумент
            2) он имеет тип sizeof()
                size_t // size of type - достаточно памяти для хранения размера типа unsigned long int, unsigned long long int - достаточно памяти для представления любого размера
            3) вывести его без знака

!ВАЖНО: непосредственная проверка показала, что просто %z работать не будет - надо писать %zd (z - расширение спецификатора, как l в %ld, указывающее на то, что целое число имеет такой размер, какой имеет size_t (к примеру, часто совпадает с %lu))

Форматирование вывода:
При выводе (при вводе не используется): % цифры тип
%3d

цифра - число позици
К примеру, %3d 12 это пробел12
%3.3d 012
%.2f 12.00
%10.2f 4пробела12.00

Форматированный вывод - чтобы было покрасивше

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

int main(void)
{
	printf("Size of long int = %z\n", sizeof(int)); // способ понять, винда у вас или нет, линукс 8, винда 4
	printf("Size of int* = %z\n", sizeof(int*)); // 64 бит 8 32 бит 4
	return 0;
} // этот код не работает (его нельзя скомпилировать), можете проверить сами, но он будет работать, если %z заменить на %zd
```

Линукс либо msys2

```
gcc g.c
./a.out ./a.exe
```

Возврат printf - число выведенных байтов

```
int scanf(const char* format, ...)
```

Работа:
1) читает строку format
2) если это не %, то
3) если это %, то
	а) есть еще один аргумент - указатель, куда положить введенные данные
	б) тип указателя - тип вводимых данных по следующей букве
	%d %i %o %x %u
	%e %f
scanf умная, сама разберется, какой из них был введен - можно вводить любой (в рамках одного типа)
%c - char*
%s - char* // так нельзя, будет предупреждение (при таком вводе не контролируется размер в байтах, длина вводимой строки)

Чтобы сформировать строку,

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

int main(void)
{
	double a = 0, b = 0, c = 0;

	printf("Input 2 numbers:");
	scanf("%lf%lf", &a, &b);

	c = a + b;

	printf("%c+%c=%e\n", a, b, c);

	return 0;
}

// так программы писать нельзя, необходимо предусмотреть проверку возвращаемого значения scanf - её работа зависит от того, что было введено пользователем

Правило работы scanf: все, что написано в format, надо ввести (если написать там \n, то scanf будет честно ждать, пока в неё введут дополнительное нажатие на клавишу "Enter", такая ошибка появляется при написании \n в конец строки format по привычке, как это делается в printf)
```