ОП

Прототип функции

Для того, чтобы к функции можно было обратится в том же файле должно находится объявление функции(прототип). Для функций описаных выше прототипами являются

```
    double line(double, double, double, double);
    double square(double, double, double);
    bool triangle(double, double, double);
```

При наличии прототипов вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией, а могут быть оформлены в виде отдельных модулей и хранится в откомпилированном виде в библиотеке объектных модулей. Это относится и к функциям из стандартных модулей. В этом случае определение библиотечных функций уже оттранслированы и оформлены в виде объектных модулей, хранятся в библиотеке компилятора, а описание функций необходимо включать в программу дополнительно. Это делается с помощью комманды `#inklyude`.

Локальные и глобальные переменные

Переменные, которые используются внутри функции называются локальными. Память под них выделяется в стеке. Поэтому после окончания работы функции они удаляются из памяти.</br>
Нельзя возвращать указатель на локальную переменную, т.к. память, выделенная этой переменной будет освобождаться.

```
int *f() {
    int a;
    ...
    return &a; // БЛЯЯЯТЬ
}
```

<mark>Глобальные переменные</mark> - это переменные, которые описаны вне функций, они видны во всех функциях, где нет локальных переменных с такими-же временами.

```
int a,b;
void Change() {
    int r, a = 5;
    r = a;
    a = b;
    b = r;
    cout << a << b;
}

int main() {
    cin >> a >> b;
    Change();
    cout <<a<<b;
    return 0;
}
```

Глобальные переменные можно использовать для передачи данных между функциями, но делать этого не рекомендуется, так как это затрудняет отладку программы и премядствует помещению функций в библиотеки. Нужно стремится к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

Параметры функции

Существует два способа передачи параметров в функцию. По значению и по адрессу, при передачи параметров по значению выполняются следующие действия.

1. Вычисляются значения выражений стоящие на месте формальных параметров.
2. В стеке выделяется память от формальных параметров функции.
3. Каждому фактическому параметру присваивается значение формального параметра. При это проверяется соответствие типов и при необходимости и возможности выполняется преобразование.

```
void Change(int a, int b) { //Передача по значению
    int r = a;
    a = b;
    b = r;
}
...
int x = 1, y = 5;
Change(x,y);
cout << x << y;
```

При передаче по адрессу в стек заносятся копии адрессов параметров, и следовательно, у функции появляется доступ к ячейке памяти с которой находится фактический параметр и она может его изменить.

Для передачи по адрессу также могут использоваться ссылки. при передаче по ссылке в функцию передаётся адресс указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются.

```
void Change(int &a, int &b) {
    int r = a;
    a = b;
    b = r;
}
...
int x = 1, y = 5;
Change(x, y);
cout << x << y;
```

Использование ссылок вместо указателей улучшает читаемость программы, так как не надо применять операцию разыменовывания, кроме того, использование ссылок более эффективно, так как не требует копирования параметов, если требуется запретить изменения параметра внутри функции используется модификатор <mark>**const**</mark>.

Передача массивов как параметров функции.

при использовании массива как параметров в функцию передаётся указатель на первый элемент. т. е. массив всегда передаётся по адрессу, при этом теряется информация о количестве элементов массива, поэтому размерность массива следует передавать как отдельный параметр.

```
int form(int x[100]) {
    int m;
    cin >> m;
    for(int i = 0; i < m; i++)
        x[i] = rand()%100;
    return m;
}

void print(int x[100], int m) {
    for(int i = 0; i < m; i++)
        printf("%d ", x[i]);
}


int main() {
    int a[100];
    int n;
    n = form(a);
    print(a, n);
    return 0;
}
```