Тема 10. Язык системного программирования С

Программа, написанная на языке Си, состоит из одной или нескольких функций. Обязательно присутствует функция main(). Данной функции всегда передается управление при запуске программы. Общая структура программы имеет вид:

<Директивы препроцессора>

<Описание типов пользователя>

<Прототипы функций>

<Описание глобальных переменных>

<Тело функций>

Функции в языке Си не могут быть вложены друг в друга.

При создании исполняемого файла к исходной программе подключаются файлы, содержащие различные библиотеки. Такие файлы содержат уже откомпилированные функции и как правило имеют расширение *lib*. На этапе компиляции компоновщик извлекает из библиотечных файлов необходимые функции. Для осуществления связи с библиотечным файлом к программе подключается заголовочный файл, который содержит информацию об именах и типах функций расположенных в библиотеке. Подключение заголовочных файлов осуществляется с помощью директив препроцессора (#include <math.h>).

Каждый байт памяти компьютера имеет свой адрес. При запуске программы она занимает некоторый участок памяти. Для всех переменных выделяются участки памяти, размером, соответствующим типу переменной. Например, для переменной типа int выделяется участок размером 4 байта. В дальнейшем, когда мы работаем с этой переменной, компилятор работает с участком памяти, адрес которого ассоциирован с именем переменной. Программист имеет возможность работать непосредственно с адресами, для этого определен соответствующий тип – указатель – переменная, содержащая какой либо адрес. Указатель имеет следующий формат:

```
Тип *имя указателя
```

Например :

```
int *a;     double *b, *d;      char *c;
```

Указатели как правило используются при работе с динамической памятью (heap или куча). Динамическая память -- эта специальная область памяти, в которой во время выполнения программы можно выделять и освобождать место в соответствии с текущими потребностями. Доступ к выделяемым участкам памяти осуществляется через указатели. Для работы с динамической памятью в языке Си определены следующие функции:

void *malloc(size) (#include malloc.h) − выделяет область памяти размером size байт. Возвращает адрес выделенного блока памяти. Если

недостаточно свободного места для выделения заданного блока памяти, то возвращает NULL.

void *calloc(n,size) (#include malloc.h) − выделяет область памяти размером n блоков по size байт. Возвращает адрес выделенного блока памяти. Если недостаточно свободного места для выделения заданного блока памяти, то возвращает NULL. Вся выделенная память заполняется нулями.

void *realloc(*u) (#include malloc.h) − изменяет размер ранее выделенной памяти, связанной с указателем u на новое число байт. Если память под указательu не выделялась, то функция ведет себя как malloc. Если с указателем u связан блок выделенной памяти, выделяется новый участок размером size байт и туда переносятся значения, связанные с указателем u. Указатель u будет теперь указывать на новый участок памяти, а старый блок памяти очищается. Если недостаточно свободного места для выделения заданного блока памяти, то возвращает NULL.

void free(*u) (#include malloc.h) – освобождает участок памяти,

связанный с указателем u.

В языке C++ для выделения и освобождения памяти определены операции new и delete. Имеется две формы операций:

1. тип *указатель = new тип (значение) – выделение участка памяти в соответствии указанным типом и занесение туда указанного значения.

delete указатель – освобождении выделенной памяти.

2. тип *указатель = new тип[n] – выделение участка памяти размером n блоков указанного типа.

delete []указатель – освобождении выделенной памяти.

Операция delete не уничтожает значения связанные с указателем, а разрешает компилятору использовать данный участок памяти.

Тип данных позволяет определить, какие значения могут принимать переменные, какая структура и какое количество ячеек используется для их размещения и какие операции допустимо на ними выполнять

Данные можно разбить на две группы: скалярные (простые) и структурированные (составные).

К ***скалярному типу*** относятся данные, представляемые одним значением (числом, символом) и размещаемые в одной ячейке из нескольких байтов.

Структурированные типы определяются пользователем как комбинация скалярных и описанных ранее структурированных типов.

Базовыми типами данных являются: целый, действительный и символьный тип.

Данные в силу особенностей их использования могут быть ***константами*** и ***переменными.*** В отличии от переменных, константы не могут изменять свое значение во время выполнения программы.

Структура - это составной тип данных, в котором под одним именем объединены данные различных типов. Отдельные данные структуры называются ***полями***. Из структур как правило организуют массивы.

Объявление структуры осуществляется с помощью ключевого слова **struct**, за которым идет ее имя и далее список элементов, заключенных в фигурные скобки:

```
struct имя
{
        тип_элемента_1  имя_элемента_1;
        тип_элемента_2  имя_элемента_2;
            ...
        тип_элемента_n  имя_элемента_n;
};
```

Перечисление (**enum)** задает множество значений для заданной пользователем переменной.

Декларация:

enum имя {набор значений};

Например:

enum otc {NEUD, UD, HOR, OTL};

Функция – это последовательность операторов, оформленная таким образом, что ее можно вызвать по имени из любого места программы. При вызове функции в нее передаются определенные данные, а из нее получают результат вычислений. Как правило, в виде функций оформляются группы операторов, которые встречаются в программе более одного раза.

Функция описывается следующим образом:

```
тип возвращаемого значения имя функции (список параметров)
{
  тело функции
}
```

Первая данного описания называется *заголовком функции*. Тип возвращаемого значения может быть любым, кроме массива или функции. Допустимо не возвращать никакого значения (тип void). Если тип возвращаемого значения не указан, то по умолчанию он имеет тип int.

Список параметров представляет собой список конструкций следующей формы:

```
тип параметра   имя параметра
```

При работе важно соблюдать следующее правило: при объявлении и вызове функции параметры должны соответствовать по количеству, порядку следования и тепам. Функция может не иметь параметров, в этом случае после имени функции обязательно ставятся круглые скобки. Существует три основных способа передачи параметров: передача по значению, по ссылке и по указателю.

Препроцессором называется специальная часть компилятора, которая обрабатывает директивы до начала процесса компиляции программы. Директива препроцессора начинается с символа #, за которым следует название

директивы. В конце директивы точка с запятой не ставится. В случае необходимости перенести директиву на следующую строку применяется знак “\”.

Директивы препроцессора используются в следующих случаях:

1. Подключение к программе заголовочных файлов с декларацией стандартных библиотечных функций, используемых в программе.

Если идентификатор файла заключен в угловые скобки, то поиск файла будет проводится в стандартном каталоге, если – в дойные кавычки, то поиск проводится в следующем порядке:

- каталог, в котором содержится файл, включивший директиву;
- каталоги файлов, которые были уже включены директивой;
- текущий каталог программы;
- каталоги, указанные опцией компилятора \I;
- каталоги, заданные переменной окружение INCLUDE.

Если файл записан с расширением, то поиск сразу ведется в указанном каталоге, и другие каталоги не рассматриваются.

В стандартном C++ заголовочные файлы не имеют расширения, а для файлов, унаследованных от Си следует указывать расширение Например, #include <math.h>.

Обработка препроцессором директивы #include сводится к тому, что директива убирается, а на ее место заносится копия указанного файла.

2. Использование #define для определения символических констант. Например, если определить в начале программы:

#define PI 3.14159265359

то во всем тексте при компиляции идентификатор PI будет заменен текстом 3.14159265359. Замена идентификатора константы не производится в комментариях и в строках. Если замещающий текст в директиве не задан, то во всем тексте соответствующий идентификатор стирается.

Данная конструкция характерна для язык Си, однако применение ее в C++ не рекомендуется. Для определения констант используется ключевое слово const. Например:

const double pi = 3.14159265359;

3. Написание макросов.

#define ***имя(параметры) реализация***

В программе во всех местах, где будет найдено указанное имя будет вставлена строка реализации. Например:

#define MAX(A,B) ((A)>(B)?(A):(B))

В программе можно использовать таким образом:

s=MAX(a,b);

Перед компиляцией каждая макрокоманда заменяется соответствующим макроопределением. Следует задавать круглые скобки вокруг каждого включенного параметра, т.к. их отсутствие может спровоцировать ошибку.

Например создадим макрос:

#define SQR(A) (A*A),

при использовании в программе:

s= SQR(a+b)

сформирует строку:

s= a+b*a+b

что будет ошибкой.

Надо:

#define SQR(A) ((A)*(A)),

тогда стока будет выглядить

s= (a+b)*(a+b).

Достоинством такой работы является экономия ресурсов, т.к. здесь в отличии от функций не тратится время на передачу управления и подготовку параметров.

Недостатки:

- снижение надежности из за отсутствия проверки типов аргументов;

- из-за того, что макрос подставляется во все места, где он используется увеличивается размер текста программы и соответственно размер исполняемого файла;

- В вышеприведенном примере s= (a+b)*(a+b). сумма вычисляется два раза. При подключении функции сумма вычислялась бы только один раз. Если в качестве формальный параметров передаются сложные функции, то из-за дополнительных вычислений может существенно замедляться работа программы.

- Если использовать s= SQR(++a) , то будет сформирована строка s= ++a*++a. При использовании макрокоманды предполагается, что перед вычислением а уменьшится на еденицу. На самом деле а уменьшится на два.

Для отмены действия директивы #define используется директива #undef. Синтаксис этой директивы следующий

#undef идентификатор

Директива отменяет действие текущего определения #define для указанного идентификатора. Например:

#undef MAX

Не является ошибкой использование директивы #undef для идентификатора, который не был определен директивой #define.

Кроме вышерассмотренных, директивы используются для условной компиляции и для изменения номеров строк и идентификатора файла.