Министерство образования и науки РФ

Томский государственный университет систем управления и радиоэлектроники (ТУСУР)

Кафедра комплексной информационной безопасности электронно-вычислительных систем (КИБЭВС)

А.С. Романов

Системное программирование

Тема № 2 – Комбинированные программы. Связывание разноязыковых модулей

Томск 2020

Цель работы

Познакомиться с основными способами передачи параметров подпрограмм, особенностями передачи управления между модулями, научиться писать комбинированные программы, в которых модули Ассемблера вызываются из модулей, написанных на высокоуровневых языках программирования.

Краткие теоретические сведения

Процедуры на Ассемблере могут получать данные из вызывающей процедуры и могут возвращать или не возвращать ей результаты своей работы. Существует несколько способов передачи параметров в процедуры.

1. **Передача параметров через регистры**. Самый быстрый и самый простой способ. Используется, например, при вызове функций прерываний BIOS.
2. **Передача данных путем прямого обращения к памяти**. При таком способе обмена данными вызываемая и вызывающая процедуры обращаются напрямую к данным, описанным в любом месте (в том числе и в теле любой процедуры) программы.
3. **Передача параметров через таблицу адресов**. В этом случае в памяти вызывающей процедуры создается специальная таблица адресов параметров. В таблицу перед вызовом процедуры записывают адреса передаваемых данных. Затем адрес самой таблицы заносится в один из регистров и управление передается вызываемой процедуре. Вызываемая процедура сохраняет в стеке содержимое всех регистров, которые собирается использовать, после чего выбирает адреса переданных данных из таблицы, выполняет требуемые действия и заносит результат по адресу, переданному в той же таблице.
4. **Передача параметров в потоке кода**. При этом данные размещают прямо в коде программы, сразу после команды CALL. Чтобы прочитать параметр, процедура должна использовать его адрес, который автоматически передается в стеке как адрес возврата. В этом случае вызываемая процедура должна изменить адрес возврата на первый байт после конца передаваемых данных перед выполнением команды RET.
5. **Передача параметров через стек**. Это наиболее распространенный и надежный способ передачи. Подавляющее большинство компиляторов передает аргументы через стек.. Параметры помещают в стек командой PUSH, после чего управление передается вызываемой процедуре. Доступ к параметрам в стеке из вызываемой процедуры осуществляют через регистр BP, в который помещают адрес вершины стека, хранящийся в регистре указателя стека SP. Для обеспечения корректного возврата в вызывающую процедуру старое значение регистра BP помещают в стек первой командой процедуры. Параметры в стеке, адрес возврата и старое значение BP вместе называют активизационной записью процедуры. Вызываемая процедура, зная структуру стека, извлекает параметры в соответствующие регистры, выполняет над ними операции и записывает результат, используя адрес, переданный в стеке.

Единого мнения по вопросам передачи у разработчиков компиляторов нет. Встречаются по крайней мере два различных механизма, именуемые соглашениями C (C convention) и Pascal (Pascal convention).

C-соглашение предписывает заносить аргументы в стек справа налево, т.е. первый аргумент функции заносится в стек последним и оказывается на его верхушке. Удаление аргументов из стека возложено не на саму функцию, а на вызываемый ею код. Главный недостаток - каждый вызов функции утяжеляет программу на несколько байт кода. Зато это соглашение позволяет создавать функции с переменным числом аргументов, т.к. удаляет их из стека не сама функция, а вызывающий ее код, который наверняка знает точное количество переданных аргументов.

Очистка стека обычно выполняется командой ADD ESP,xxx, где xxx - количество удаляемых байт. Поскольку в 32-разрядном режиме каждый аргумент, как правило, занимает четыре байта, количество аргументов функции вычисляется так: n_args = xxx / 4. Оптимизирующие компиляторы для очистки стека от нескольких аргументов могут выталкивать их в неиспользуемые регистры командой POP или и вовсе очищают стек не сразу же после выхода из функции, а там, где это удобнее компилятору.

Pascal-соглашение предписывает заносить аргументы в стек слева направо. Это означает, что первый аргумент функции заносится в стек в первую очередь и оказывается в его "низу". Удаление аргументов из функции поручено самой функции и обычно осуществляется командой RET xxx, т.е. возврат из подпрограммы со снятием xxx байт со стека. Возвращаемое функцией значение в обоих соглашениях передается через регистр EAX (или через регистровую пару EDX:EAX при возвращении 64-разрядных переменных).

Удаление параметров из стека можно организовать по-разному. Если стек освобождает вызываемая процедура по команде RET, то код программы получается более коротким. Если за освобождение стека отвечает вызывающая программа, то становится возможным вызов нескольких процедур с одними и теми же значениями параметров просто последовательными командами CALL.

Второй способ используется в языках Си и Си++ и дает больше возможностей для оптимизации. Вопрос о порядке записи параметров в стек для ассемблера не столь важен, так как и записывает и извлекает подпрограммы все сами. А вот при взаимодействии ассемблера с языками высокого уровня, следует знать особенности связи модулей этих языков. Рассмотрим эти особенности на примере языка Си.

Итак, передача параметров в Си осуществляется через стек. Причем, что именно помещается в стек (значение или адрес) определяется явно средствами языка. При передаче параметров Си руководствуется внутренним представлением данных. Параметры помещаются в стек в соответствии с прототипом в обратном порядке, то есть справа налево.

Так при вызове функции с прототипом:

```
void a(int p1, int p2, long int p3);
```

в стек сначала будет занесен параметр **p3** (длиной 4 байта), затем **p2** и **p1** (по два байта каждый), а затем уже адрес возврата (ближний или дальний в зависимости от используемой модели памяти)

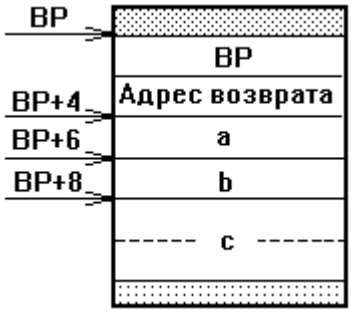

Рисунок 1 – Стек при передаче параметров

После вызова функции стек восстановит вызывающая программа.

Если используется функция с переменным числом параметров, то это отразится только на размере области параметров, так как каждый параметр будет помещен в стек, а удаление параметров будет выполнять вызывающая программа.

Возвращаемые значения должны быть записаны в регистры:

char, short, int, enum, указатели **near** - в регистр AX;

указатели **far, huge** и прочие 4-х байтовые величины - в регистры DX:AX;

float, double - в регистры TOS и ST(0) сопроцессора;

struct - записывается в память, а в регистр записывается указатель (структуры длиной в 1 и 2 байта возвращаются в AX, а 4 байта - в DX:AX).

Существует еще одна особенность внутреннего представления программ на языке Си++: компилятор языка изменяет используемые имена. Этот процесс, называемый обработкой имен, выполнят сохранение информации о типах аргументов функции, путем модификации ее имени таким образом, чтобы оно указывало на типы аргументов. При разработке программ на Си++ обработка имени выполняется автоматически и скрыта от программиста. Однако если какой-то модуль написан на Ассемблере, программист должен самостоятельно выполнить обработку имен функций в этом модуле. Для этого необходимо знать соглашения, принятые в языке Си++.

1. Перед глобальными именами ставится символ подчеркивания;

2. К именам функций в начало добавляется символ **@**, а в конец дописываются знаки **$q** и символы, кодирующие типы параметров функции в виде:

```
@ <имя функции> $q<коды типов параметров>
```

Таблица 1 - Коды типов параметров

Тип	**Эквивалент**	**Тип**	**Эквивалент**	**Тип**	**Эквивалент**
void	v	float	f	long	l
char	zc	double	d	*, [ ]	p
int	i	short	s	…	e

Например:

```
fa(int *s[], char c, short  t)     =>   @fa$qppizcs.
```

3. Для отмены чувствительности Си++ к регистру следует указать опцию Case sensitive …off.

Обработку имен ассемблерных функций можно и не выполнять, например, чтобы избежать несовместимости с последующими версиями компиляторов, в которых возможны изменения алгоритма этой обработки. С этой целью Си++ позволяет использовать стандартные имена функций Си в программах написанных на Си++, например:

```
extern  “C” {
int SUM (int *a,int b)
}
```

Все функции, объявление которых заключено в фигурные скобки, будут иметь имена, соответствующие соглашениям, принятым в языке Си. Приведенная выше функция на Ассемблере SUM будет иметь следующий вид:

```
public _SUM
_SUM proc
```

Таким образом, при объявлении в ассемблерном модуле функций, включенных в блок *extern “C”*, нет необходимости выполнять обработку их имен.

Для того чтобы скомпоновать модули на ассемблере с программой, написанной на Си, необходимо следовать определенным соглашениям.

При компиляции исходной программы на Си создаются следующие сегменты:

- сегмент кода;
- сегмент данных;
- сегмент неинициализированных данных.

Используемая модель памяти влияет не только на тип вызываемой функции и указателей на данные, но и на то, какие сегменты будет использоваться программой. В таблице 2 приведены имена сегментов, используемые Си для различных моделей памяти.

Таблица 2 - Имена сегментов, используемые различными моделями памяти

Модель памяти	**Сегмент кодов**	**Сегмент инициированных данных**	**Группа сегментов данных, адресуемых DS**
Tiny	_TEXT	_DATA	DGROUP
Small	_TEXT	_DATA	DGROUP
Compact	_TEXT	_DATA	DGROUP
Middle	<имя файла>_TEXT	_DATA	DGROUP
Large	<имя файла>_TEXT	_DATA	DGROUP
Hugo	<имя файла>_TEXT	<имя файла>_DATA	<имя файла>_DATA

Си позволяет ассемблеру увеличивать список глобальных переменных, доступных для всех модулей. Это достигается за счет размещения переменных в сегменте данных, отведенном для глобальных переменных, и описания его внутренним **public.** Имя такой переменной по правилам Си должно начинаться со знака подчеркивания. Прочие модули, использующие данное имя, должны включать его описание как **extrn** (на Ассемблере) или **extern** (на Си).

Приведем примеры программы.

Пример 1 - Определение минимального значения из двух заданных (реализация с переменным количеством параметров функции).

Листинг модуля C++:

```
#include <stdio.h>
extern int amin(int count,int v1,int v2,...); //первый параметр -
счетчик
void main()
{
  int a=3,b=5,c;
  c=amin(5,a,b,1,10,0);
  printf("c=%d",c);
}
```

Листинг модуля на Ассемблере:

```
_TEXT segment byte public 'CODE'
  assume CS:_TEXT
  public @amin$qiiie
@amin$qiiie proc near
  push BP
```

```
  mov  BP,SP
  mov  AX,0
  mov  CX,[BP+4]  ; в CX заносится количество значений
  cmp  CX,AX
  jle  exit
  mov  AX,[BP+6]  ; в AX заносится первое значение из списка
  jmp  short ltest
compare:
  cmp  AX,[BP+6]
  jle  ltest
  mov  AX,[BP+6]
ltest:
  add  BP,2
  loop compare
exit:
  pop  BP
  ret
@amin$qiiie endp
_TEXT  ends
end
```

Пример 2 - Определение среднего арифметического последовательности из 10 чисел. Си вызывает функцию на ассемблере для суммирования чисел, а ассемблер вызывает функцию на Си для выполнения операции деления в вещественной арифметике.

Листинг модуля C++:

```
#include <stdio.h>
extern float Average(int far * ValuePtr, int NumberOfValues);

#define NUMBER_OF_TEST_VALUES 10
int TestValues[NUMBER_OF_TEST_VALUES] = {1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
main()
{
  printf("The average value is: %f\n",
  Average(TestValues, NUMBER_OF_TEST_VALUES));
}
float IntDivide(int Dividend, int Divisor)
{
  return( (float) Dividend / (float) Divisor );
}
```

Листинг модуля на Ассемблере:

```
.MODEL  SMALL
  EXTRN   @IntDivide$qii:PROC
```

```
.CODE
  PUBLIC  @Average$qnii
@Average$qnii    PROC
  push BP
  mov  BP,SP
  les  BX,[BP+4]  ; загрузка в ES:BX адреса массива значений
  mov  CX,[BP+8]  ; загрузка количества чисел
  mov  AX,0       ; обнуление суммы
AverageLoop:
  add  AX,ES:[BX] ; добавление очередного значения
  add  BX,2       ; переход к следующему значению
  loop AverageLoop
  push WORD PTR [BP+8] ; количество чисел в стек (второй параметр)
  push AX         ; запись в стек суммы чисел (первый параметр)
  call @IntDivide$qii  ; вызов функции на Си
  add  SP,4            ; удаление параметров
  pop  BP
  ret             ; среднее значение находится в регистре
@Average$qnii    ENDP
end
```

При подключении модуля на ассемблере в Visual C++, его необходимо предварительно откомпилировать. Затем полученный объектный модуль, в котором находится ассемблерная процедура, необходимо подключить к приложению в файл проекта следующим образом:

```
extern  "C" void __<конвенция>  ADD1(int a,intb,int &c);
```

Модуль на Ассемблере необходимо транслировать с опциями:

```
ML /c /coff program.asm
```

или

```
tasm /ml program.asm
```

Пример использования конвенции **stdcall**:

```
extern  "C" void __stdcall  ADD1(int a,intb,int &c);
```

```
. 386
. model flat
. code
  public _ADD1
  _ADD1 proc
  push EBP
  push EBP,ESP
  mov  EAX,dword ptr [EBP+8]
```

```
  add EAX, dword ptr [EBP+12]
  mov EDX, dword ptr [EBP+16]
  mov [EDX], EAX
  pop EBP
  ret 12   ;  стек освобождает сама процедура
_ADD1 endp
end
```

Пример использования конвенции **fastcall**:

```
extern “C” void __fastcall  ADD1(int a,intb,int &c);
. 386
. model  flat
. code
public  @ADD1@12
@ADD1@12 proc
  add EAX, EDX
  mov [EDX], EAX
  ret ;  стек освобождает вызывающая программа
@ADD1@12   endp
end
```

Другим вариантом комбинирования кода, написанного на языке высокого уровня и на Ассемблере является использование ассемблерных вставок.

Посмотрим, как использовать ассемблерные вставки в Visual Studio 2019. Решаемая задача - установить 1 в 4-ых битах всех элементов массива из 10 байтов. Определить сумму элементов полученного массива. Как и раньше, необходимо с помощью asm обозначить сегмент вставки, в котором будут работать команды ассемблера.

```
#include <iostream>
using namespace std;
int setforthbit(unsigned char* data)
{
  short int sum = 0;
  _asm
  {
    mov ebx, data
    mov ecx, 10

    mov ax, 0
    m1:
    mov al, [ebx]
    or al, 00001000b
    mov [ebx], al

    add sum, ax
    inx ebx
```

```
    loop m1
  }
  return sum;
}

int main()
{
  unsigned char array[10] = {0,1,2,3,4,5,6,7,8,9};
  cout << "initial array is:" << endl;
  for (int i=0; i<10; i++)
  {
    cout << (int)array[i] << "\t";
  }
  cout << endl;
  int sum = setforthbit(array);
  cout << "final array is:" << endl;
  for (int i=0; i<10; i++)
  {
    cout << (int)array[i] << "\t";
  }
  count << endl << sum << endl;
  system("pause");
}
```

В случае использования компилятора gcc ассемблерные команды нужно брать в двойные кавычки. При этом в конце каждой команды добавлять суффикс, состоящий из символов переноса строки и табуляции ‘\n’ ‘\t’. Для оформления вставки можно использовать ключевые слова asm или __asm__.

Формат ассемблерной вставки для gcc в общем случае выглядит так:

```
asm (assembler template           /* ассемблерная встравка*/
     : output operands            /* выходные операнды */
     : input operands             /* входные операнды */
     : list of clobbered registers /* разрушаемые регстры */
    );
```

Шаблон ассемблера (assembler template) состоит из инструкций ассемблера.

Двоеточие отделяет шаблон ассемблера от первого выходного операнда, а другие разделители (тоже двоеточие), отделяют последний выходной операнд от первого входного, если таковые имеются. При этом, если выходных операндов нет, но есть входные операнды, то необходимо поместить подряд два двоеточия там, где должны быть выходные операнды.

Директива .intel_syntax меняет синтаксис AT&T на синтаксис Intel; необходимо дополнительно указывать смену синтаксиса для операндов инструкций, noprefix, что позволит писать код в более близком к диалекту nasm виде, не используя при записи имен регистров префикс %.

Рассмотрим простейший пример

```
int f() {
  int a=10, b;
  __asm__ (".intel_syntax noprefix\n\t"
                "mov eax, %1\n\t"
                "mov %0, eax\n\t"
                :"=r"(b)
                :"r"(a)
                :"%eax"
                );
   return b;
}
```

В первой ассемблерной инструкции в регистр eax помещается значение входного операнда вставки %1. Во второй инструкции в выходной операнд вставки %0 пересылается значение регистра eax. Для выходных операндов строка ограничения типа должна начинаться с символа `=`. Следующий в строке символ указывает, куда компилятору нужно поместить значение соответствующей переменной. Способ кодировки одинаковый для входных и выходных операндов вставки. После завершения работы вставки значение переменной b будет находиться в регистре eax и его потребуется записать в переменную b.

Приведем более сложные пример программы, транспонирующей случайно сгенерированную матрицу. При этом собственно транспонирование выполнил в виде ассемблерной вставки.

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
    srand(time(0));
    long long n = rand() % 9 + 2;
    //long long n = 100;
    long long array[n][n];
    cout<<"\tArray"<<endl;
        for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            array[i][j] = rand() % 9 + 1;
            cout<<array[i][j]<<"  ";
        }
        cout<<endl;
    }
    long long *ptr_array = &array[0][0];
    long long *end_array = &array[n-1][n-1];
```

```
    //cout<<endl<<"end array = "<<end_array<<endl;
    //cout<<endl<<"ptr array = "<<ptr_array<<endl;
    if (n<=1)
    {
    cout<< "Bad array size!"<<endl;
    return -1;
    }
    asm(
        "mov $8, %%rax\n\t"  // Поместили 8 в rax
        "mulq %[n]\n\t"      // Умножили rax на n (получили сдвиг для
перехода по строкам)
        "mov %[ptr_array], %%rbx\n\t"// Поместили указатель на начало
матрицы в rbx
        "mov %%rbx, %%rcx\n\t" // Поместили указатель на начало матрицы в
rcx
        "add %%rax, %%rcx\n\t" // Прибавили к rcx содержимое rax
(сдвинули указатель вниз на одну строку)
        "add $8, %%rcx\n\t"     // Прибавили к rcx 8 (сдвинули указатель
вправо на один столбец)
        "push %%rcx\n\t" // Поместили в стек rcx (адрес следующего
элемента главной диагонали)
                "push %%rax\n\t"     // Поместили в стек rax (сдвиг для
перехода по строкам)
        "mov %[n], %%rcx\n\t" // Поместили в rcx значение n
        "dec %%rcx\n\t" // Уменьшили на 1 значение rcx (получили в rcx n-
1)
        "mulq %%rcx\n\t" // Умножили rax (сдвиг для перехода по строкам)
на rcx (n-1)
        "add %%rax, %[ptr_array]\n\t"// Прибавили к ptr_array содержимое
rax (получили указатель на первый элемент последней строки матрицы)
        "mov %%rbx, %%rcx\n\t" // Вернули в rcx указатель на начало
матрицы
    "loop_start:\n\t"

        "add $8, %%rbx\n\t" // Увеличили адрес в rbx на 8 (переместились
на один элемент вправо)
        "pop %%rax\n\t" // Взяли в rax из стека смещение на следующую
строку
        "add %%rax, %%rcx\n\t" // Увеличили значение в rcx на смещение на
следующую строку (переместились вниз на один элемент)
        "push %%rax\n\t" // Вернули в стек смещение на следующую строку
        "push (%%rbx)\n\t" // Обмен значений элементов матрицы через стек
        "push (%%rcx)\n\t"
        "pop (%%rbx)\n\t"
        "pop (%%rcx)\n\t"
```

```
        "cmp %%rcx, %[ptr_array]\n\t"// Проверили что не достугнута
последняя строка матрицы
        "je next_iteration\n\t" // Если достигнута - переход на
next_iteration
        "jmp loop_start\n\t" // Если НЕ достигнута - переход на
start_loop
    "next_iteration:\n\t"
        "add $8, %[ptr_array]\n\t" // Сдвинули указатель на один элемент
последней строки матрицы вправо
        "mov %[ptr_array], %%rax\n\t"// Переместили указатель в rax
        "cmp %%rax, %[end_array]\n\t"  // Проверили что достигнут конец
матрицы
        "je end_asm\n\t" // Если достигнут - завершение, иначе
продолжение
        "pop %%rax\n\t" // Взяли в rax из стека смещение на следующую
строку
        "pop %%rbx\n\t" // Взяли в rbx из стека указатель на следующий
элемент главной диагонали, он станет началом новой "усеченной" матрицы
        "mov %%rbx, %%rcx\n\t" // Поместили в rcx указатель на следующий
элемент главной диагонали
        "add %%rax, %%rcx\n\t" // Сдвинули указатель в rcx на одну строку
вниз
        "add $8, %%rcx\n\t"     // Сдвинули указатель в rcx на один
элемент вправо
        "push %%rcx\n\t"        // Вернули rcx в стек
        "mov %%rbx, %%rcx\n\t"  // Поместили в rcx значение из rbx
        "push %%rax\n\t"        // Вернули rax в стек
         "jmp loop_start\n\t"  // Переход на loop_start
    "end_asm:\n\t"
        :
        :[ptr_array]"m"(ptr_array), [end_array]"m"(end_array), [n]"m"(n)
        :"%rax", "%rbx", "%rcx"
    );
      cout<<endl<<"--------------------------------"<<endl;
    cout<<"\tArray(T)"<<endl;
   for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout<<array[i][j]<<"  ";
        }
        cout<<endl;
    }
    return 0;
}
```

Скомпилиурем, набрав в командной строке:

```
gcc -fno-pie -no-pie lab3.cpp -o lab3 -g -lstdc++
```

```
root@551defe7e323:/# gcc -fno-pie -no-pie lab3.cpp -o lab3 -g -lstdc++
root@551defe7e323:/# ./lab3
        Array
7  9  2  7  7  2  9  8
5  9  6  6  1  1  8  1
7  7  9  3  2  9  3  8
3  4  4  8  7  1  5  4
1  6  8  7  5  6  3  9
5  8  3  5  7  8  3  4
5  2  4  5  2  6  3  2
7  4  9  5  5  3  6  3

--------------------------------
        Array(T)
7  5  7  3  1  5  5  7
9  9  7  4  6  8  2  4
2  6  9  4  8  3  4  9
7  6  3  8  7  5  5  5
7  1  2  7  5  7  2  5
2  1  9  1  6  8  6  3
9  8  3  5  3  3  3  6
8  1  8  4  9  4  2  3
root@551defe7e323:/#
```

Рисунок 2 – Результат выполнения программы, скомпилированной gcc

Задание

1. Изучить краткие теоретические сведения, материалы лекций по теме практического занятия и приведенные выше примеры программ.

2. Все действия, описанное далее выполнять в docker контейнере, образ - любой привычный linux.

3. Получить индивидуальное задание у преподавателя. Решение основной задачи реализовать на ассемблере в виде процедуры. Далее получить объектный модуль. Вызывать из кода на C++ процедуру из объектного файла и скомбинировать их в один исполняемый файл. Альтернативный вариант - использовать ассемблерные вставки.

4. Написать отчет и защитить у преподавателя. Залить код, докерфайл в git, на мудл залить архив с отчетом и всеми исходными кодами, dockerfile.

Варианты индивидуальных заданий

1. Напишите программу, в которой создается два числовых массива одинакового размера. Необходимо вычислить сумму попарных произведений элементов этих массивов. Так, если через a_k и b_k, обозначить элементы массивов (индекс $0 \le k < n$), то необходимо вычислить сумму $\sum_{k=0}^{n-1} a_k b_k$

2. Напишите программу, в которой создается числовой массив и для этого массива вычисляется сумма квадратов элементов массива.

3. Напишите программу, в которой создается двумерный числовой массив и для этого массива вычисляется сумма квадратов его элементов.

4. Напишите программу, в которой создается квадратная матрица (реализуется через двумерный массив). Матрицу необходимо заполнить числовыми значениями (способ заполнения выбрать самостоятельно), а затем выполнить транспонирование матрицы: матрица симметрично «отражается» относительно главной диагонали, в результате чего элемент матрицы a_{ij} становится элементом a_{ji} и наоборот.

5. Напишите программу, в которой создается квадратная матрица(реализуется через двумерный массив). Матрица заполняется случайными числами, после чего выполняется «поворот по часовой стрелке»: первый столбец становится первой строкой, второй столбец становится второй строкой, и так далее.

6. Напишите программу, в которой для двумерной квадратной числовой матрицы вычисляется след — сумма диагональных элементов матрицы.

7. Напишите программу, в которой для одномерного числового массива вычисляется наибольший (наименьший) элемент и позиция, на которой элемент находится в массиве.

8. Напишите программу, в которой создается одномерный числовой массив. После заполнения значениями (например, случайными числами) массива в нем нужно выполнить циклическую перестановку элементов. Количество позиций для циклической перестановки вводится пользователем с клавиатуры.

9. Напишите программу, в которой создается и заполняется натуральными числами двумерный массив. Заполнение начинается с левого верхнего элемента слева направо, сверху вниз (то есть заполняется сначала первая строка, затем вторая, и так далее).

10. Напишите программу, в которой создается и заполняется натуральными числами двумерный массив. Заполнение начинается с левого верхнего элемента сверху вниз, слева направо (то есть заполняется сначала первый столбец, затем второй и так далее).

11. Напишите программу, в которой создается квадратная матрица. Элементам на обеих диагоналях присваиваются единичные значения, все остальные элементы нулевые.

12. Напишите программу для вычисления произведения прямоугольных матриц (количество строк и столбцов в матрицах различное). Перемножаются матрицы A (размерами m на n) и B (размерами n на l). Результатом является матрица C (размерами m на l). Рабочая формула для вычисления значений матрицы C имеет вид $C_{ij} = \sum_{k=0}^{n} a_{ik} b_k j$, где $1 \le i \le m$ и $1 \le j \le l$.

13. Напишите программу, в которой создается символьный массив для записи текста. В массив записывается текст, после чего необходимо изменить порядок следования символов в тексте на противоположный: последний символ становится первым, предпоследний символ становится вторым и так далее.

14. Напишите программу, в которой создается двумерный числовой массив и заполняется случайными числами. Для каждой строки (или столбца) двумерного массива определяется наибольший (или наименьший) элемент, и из таких элементов создается одномерный числовой массив.

15. Напишите программу, в которой создается символьный массив для записи текста. Подсчитать для записанного в массиве текста количество слов и длину каждого слова. Словами считать набор символов между пробелами.

16. Напишите программу, в которой создается динамический массив. Размер массива — случайное число (то есть генерируется случайное число, которое определяет размер массива). Заполнить массив «симметричными» значениями: первый и последний элемент получают значение 1, второй и предпоследний элемент получают значение2, и так далее.

17. Напишите программу, в которой создается три одномерных числовых массива одинакового размера. Первые два массива заполняются случайными числами. Третий массив заполняется так: сравниваются элементы с такими же индексами в первом и втором массиве и в третий массив на такую же позицию записывается большее (или меньшее) из сравниваемых значений.

18. Напишите программу, в которой создается два динамических массива (разной длины). Массивы заполняются случайными числами. Затем создается третий динамический массив, и его размер равен сумме размеров первых двух массивов. Третий массив заполняется так: сначала в него записываются значения элементов первого массива, а затем значения элементов второго массива.

19. Напишите программу, в которой создается два динамических массива одинакового размера. Массивы заполняются случайными числами. Затем создается третий динамический массив, и его размер в два раза больше размера каждого из первых двух массивов. Третий массив заполняется поочередной записью элементов из первых двух массивов: сначала записывается значение элемента первого массива, затем значение элемента второго массива, затем снова первого и снова второго, и так далее.

20. Напишите программу, в которой создается двумерный числовой массив. Массив заполняется случайными числами. На его основе создается новый массив, который получается «вычеркиванием» из старого одной строки и одного столбца. Номера «вычеркиваемых» столбца и строки определяются вводом с клавиатуры или с помощью генератора случайных чисел.

Контрольные вопросы

1. Передача параметров через регистры.
2. Передача данных путем прямого обращения к памяти.
3. Передача параметров через таблицу адресов.
4. Передача параметров в потоке кода.
5. Передача параметров через стек.
6. Для чего предназначены команды CALL и RET в языке Ассемблер?
7. Соглашение для имен функция в C++.
8. Как можно подключить модуль, написанный на Ассемблере в C++?