Глава 3

Структуры данных

В данном разделе рассматриваются реализации классических структур данных — стека, дека, очереди, списка, дерева. Каждая из этих структур позволяет хранить некоторое количество элементов, добавлять новые, извлекать и удалять существующие, а также предоставлять непосредственный доступ к отдельным значениям. Структуры различаются между собой по правилам доступа к элементам и внутренними логическими связями.

В задачах, приведенных в этом разделе, предлагается построить структуры данных для хранения элементов некоторого абстрактного типа `Type`. Можно считать, что для программы на языке C этот тип определен при помощи оператора `typedef`. Например, для элементов, являющихся целыми числами, имеем `typedef int Type;` В этом случае программная реализация каждой структуры данных должна представлять собой отдельный файл, в котором содержатся объявления некоторого набора глобальных статических объектов (для реализации внутреннего представления данных) и определения функций (для реализации требуемых действий с данными), а также заголовочный файл с описаниями используемых типов данных и прототипами функций.

При использовании языка C++ реализация выливается в построение некоторого класса (или иерархии классов) с соответствующими скрытыми членами и общедоступными методами. Для обеспечения возможности использовать реализацию для различных типов данных, можно воспользоваться параметризацией построенных классов с помощью конструкции `template<class Type>`.

Практическая реализация каждой структуры данных должна сопровождаться тестирующей программой, которая вызывала бы все функции реализации и проверяла их работу в различных ситуациях.

3.1 Стек, дек, очередь

Стек. Так называется структура данных, в которую можно последователь-

но добавлять элементы, но в в каждый момент времени доступным является только элемент, добавленный в стек последним. Только этот элемент, называемый вершиной стека, можно извлечь (или удалить) из стека, после чего становится доступным предыдущий добавленный элемент. Эта дисциплина доступа часто обозначается аббревиатурой LIFO (Last In, First Out), т.е. последний добавленный элемент извлекается из стека первым.

3.1. Построить реализацию стека элементов абстрактного типа Type.

Идеи реализации. Для реализации достаточно выделить участок памяти для размещения необходимого количества элементов, размещать поступающие элементы последовательно один за другим рядом друг с другом и хранить указатель на элемент, являющийся текущей вершиной. При добавлении (извлечении) элементов указатель текущей вершины будет перемещаться к следующему свободному (предыдущему занятому) элементу выделенного участка памяти.

Решение. В качестве примера возможного стиля программирования приведем здесь две реализации стека (для определенности тип `Type` есть `int`).

Реализация на языке C

```
/*---   Файл  stack.h  ------------------------------------------*/
typedef int  Type;
int     St_Init (int maxsize);  /* создать стек на maxsize эл-тов */
void    St_Term (void);         /* закончить работу               */
int     St_Push (Type val);     /* добавить элемент  val          */
int     St_Del  ();             /* удалить вершину                */
int     St_Pop  (Type *dst);    /* удалить вершину и поместить ее
                                    значение по указателю dst     */
Type  * St_Top  (void);         /* указатель на вершину           */
int     St_Size (void);         /* текущее кол-во элементов       */
int     St_Room (void);         /* кол-во свободных элементов     */
/*---   конец файла stack.h   -----------------------------------*/
```

Функции `St_Init`, `St_Push`, `St_Del`, `St_Pop` возвращают 0 при успешном выполнении требуемой операции и -1 в противном случае.

```
/*---   Файл  stack.c  ------------------------------------------*/
#include <stdlib.h>
#include "stack.h"
static  Type    *mem;  /* начало области элементов стека */
static  Type    *top;  /* указатель на вершину стека */
static  int     size;  /* текущее количество элементов */

int     St_Init (int maxsize)
{  size = 0;
   if ( top=mem=(Type*)malloc(sizeof(TYPE)*maxsize) )
      top+=maxsize;
   return  (mem) ? 0 : -1;
}
void    St_Term (void)
```

```
{
   if(!mem)free (mem);
   size = 0;
   return;
}
int     St_Push (Type val)
  { return (top!=mem) ? *(--top)=val, ++size, 0 : -1; }
int     St_Pop (Type *dst)
  { return (size) ? *dst=*(top++), --size, 0 : -1; }
int     St_Del ()
  { return (size) ? ++top, --size, 0 : -1; }
Type  * St_Top  (void)
  { return top; }
int     St_Size (void)
  { return size; }
int     St_Room (void)
  { return top-mem; }
/*---   конец файла stack.c   ------------------------------------*/
```

Реализация на языке C++.

```
//---   Файл  stack.h  ---------------------------------------
#define     DEFAULT_SIZE   10

template <class Type>
class  Stack
{  private:
      Type   *mem, *top;
      int       size;
   public:

// Конструктор и деструктор
   Stack ( int maxsize = DEFAULT_SIZE )
   {  mem = new Type [maxsize];
      top = mem + maxsize; size = 0;
   }
   ~Stack () { delete [] mem; }

// Методы работы со стеком
   int  Push (Type value)
   {  return (top!=mem) ? *(--top)=value, ++size, 0 : -1; }
   int  Pop (Type &dst)
   {  return (size) ? dst=*(top++), --size, 0 : -1; }
   int  Del () { return (size) ? ++top, --size, 0 : -1; }
   Type & Top () { return *top; }
   int  Empty() { return size==0; }
   int  Room () { return top-mem; }
   int  Success () { return (mem) ? 1 : 0; }
};
//---   конец файла stack.h   ---------------------------------
```

Замечание. Функция `Success()` добавлена для того, чтобы мы могли обнаружить отказ в выделении памяти под стек не привлекая механизма исключений языка C++.

3.2. Напишите программу для распечатки полного содержимого стека в наглядной форме. Составьте тестирующую программу, которая бы позволяла в режиме диалога вызывать все функции стека и выводила на экран состояние стека после каждой операции. Проверьте работоспособность реализации для типов `int` и `char`.

Замечание. Работа со стеком на основе глобальных переменных в некоторых случаях может оказаться обременительной. Более гибкие и защищенные от ошибок конструкции можно реализовать, поместив соответствующие переменные в структуру и передавая в функции указатель на ее содержимое.

3.3. Построить реализацию стека элементов абстрактного типа Type на основе структуры следующего вида:

```
struct Stack{
  Type * mem;          /* начало области элементов стека */
  int size;            /* текущее количество элементов */
  int maxsize;         /* размер стека */
};
```

3.4. Пусть имеется некоторое строковое выражение, содержащее скобки трех видов `{,},[,],(,)`. На основе стека с элементами типа `char`, реализуйте функцию проверки баланса скобок.

Идеи реализации. Будем посимвольно анализировать имеющееся выражение и добавлять все встечающиеся скобки в стек. Если при добавлении очередной закрывающей скобки вершина содержит открывающую скобку такого же вида, то оба элемента из стека удаляются. Выражение корректно, если по завершении стек окажется пустым.

3.5. Постройте реализацию 2-х стеков, элементами которого являются целые числа.

Идеи реализации. В функции `main()` создадим массив переменных

```
struct Stack st[2];
```

и при вызове базовых функций `St_Init`, `St_Push`, `St_Del`, `St_Pop` будем дополнительно передавать в них адрес структуры `&st[k]` активного в данный момент k-го стека.

3.6. Постройте реализацию N стеков с элементами типа `Type`.

3.7. Пусть имеется стек чисел (аргументов) и стек бинарных операций. Напишите функцию вычисления результата соответствующего арифметического выражения по правилу: пока стек операций не пуст, прменяем соответствующее действие для двух верхних чисел из стека; удаляем использовавшиеся аргументы из стеков; результат вычисления укладываем в стек аргументов.

3.8. Напишите функцию формирования двух стеков из задачи 3.7 по корректной математической формуле со скобками и бинарными операциями.

3.9. Постройте реализацию стека, элементом которого является структура вида

```
struct User{
  char FistName[10];
  char LastName[10];
  unsigned int  UId;
};
```

Проведите тестирование работоспособности программы, подготовив файлы с командами и входными данными.

3.10. Постройте реализацию стека, элементами которого являются указатели на строки, содержащие не более ста символов.

Идеи реализации. Заведем следующую структуру

```
struct Stack{
  char ** mem;    /* начало области элементов стека */
  int size;             /* текущее количество элементов */
  int maxsize;       /* размер стека */
  char buf[100];
};
```

Очередную строку будем считывать в `buf[]`, далее вычислять ее реальную длину, выделять для хранения с помощью функции `malloc()` память требуемого размера, сохраняя указатель в стеке, копировать из буфера элемент в выделенную память. При удалении элемента выделенная для его хранения память высвобождается функцией `free`.

3.11. Пусть имеется некоторое строковое выражение, содержащее корректную математическую формулу с бинарными операциями, записанную в обратной польской нотации (опп). По определению, выражение состоит из чисел и знаков операций и обрабатывается слева направо. Если встречается знак операции, то указанное действие выполняется для двух идущих перед ним чисел в порядке их записи. Результат записывается вместо участвовавших чисел и оператора. Процесс повторяется до тех пор, пока не останется последнее число, являющееся ответом. На основе задачи 3.10 реализуйте процедуру считывания готовой опн в построенный стек, и процедуру вычисления соответствующего результата.

3.12. Напишите функцию формирования опн для корректной математической формулы со скобками и бинарными операторами.

3.13. Реализуйте стек строк на базе одного большого блока памяти.

Идеи реализации. Выделим достаточно большой блок памяти и будем укладывать строки — последовательность символов, заканчивающихся символом с кодом 0x0A, одну за другой в этом блоке, сохраняя в стеке целые числа, равные смещению от начала блока до начала строки. Для добавления достаточно поддерживать указатель на начало свободной области в блоке

и сравнивать длину этой области с длиной добавляемой строки. Для удаления можно отыскивать в занятой части блока конец предыдущей строки и передвигать указатель свободной позиции на следующий за этим концом символ.

3.14. Добавьте к решению задачи 3.13 возможность эффективного сохранения стека в файл и считывания настроенного стека из такого файла, используя функции $fread, fwrite$.

3.15. Реализация стека строк из задачи 3.14 требует последовательного поиска конца предыдущей строки при удалении элемента. Модифицируйте реализацию, помещая вслед за каждой добавленной строкой ее длину. Это позволит сразу вычислить значение адреса начала последней добавленной строки по адресу начала свободной области в стеке.

3.16. Реализуйте стек строк (см. задачу 3.13) при условии, что строки хранятся в файле, причем начало файла соответствует дну стека, а вершиной стека является последняя строка в файле. При добавлении строк к файлу его размер должен увеличиваться, а при удалении строк — уменьшаться (следует использовать функции fread, fwrite, и т.п.). Реализуйте следующие функции

```
int     OpenFileStack (char *filename);
int     PushString (char *str);
int     PopString (char *str);
int     CloseFileStack (void);
```

Каждая из этих функций возвращает 0 в случае успешного выполнения требуемой операции и -1, если выполнение операции по каким-либо причинам невозможно.

Модифицируйте реализацию так, чтобы уменьшение длины файла при удалении строк происходило не каждый раз, а только при накоплении достаточно большой суммарной длины удаленных строк, например, 1024 байт). Проверьте как повлияет такая модификация на время работы со стеком.

3.17. Реализуйте (см. задачу 3.16) файловый стек, элементами которого будут байтовые записи произвольной длины, хранимые в следующем формате:

```
<запись 1>
<длина записи 1>
<запись 2>
<длина записи 2>
........
........
<запись k>
<длина записи k>
```

где `<длина записи n>` есть четырехбайтовое поле, содержащее значение числа типа `unsigned long` — длины соответствующей записи. Последняя запись в файле (т.е. `<запись k>`) соответствует вершине стека.

Реализуйте функции со следующими прототипами

```
int     OpenFileStack (char *filename);
int     PushRecord (void *record, size_t length);
          // функция помещает в стек length байтов,
          // размещенных начиная с адреса record
int     PopRecord  (void *record, size_t *length);
          // функция берет вершину стека и помещает ее
          // в области памяти по адресу record,
          // значение длины записи помещается по адресу length
size_t  TopLength  (void);
          // функция возвращает длину записи
          // на вершине стека
int     CloseFileStack (void);
```

Функции возвращают 0 в случае успешного выполнения и -1 при отказе.

3.18. Постройте реализацию файлового стека строк на основе объединения идей задач 3.16, 3.17 Строки частично хранятся в памяти, частично — на диске. В памяти выделяется два блока — первый и второй. Строки накапливаются сначала в первом блоке, а когда он заполнится — во втором. При заполнении второго блока первый блок записывается в файл (по стековому принципу) и блоки обмениваются своими номерами: второй (занятый) блок становится первым, а первый (теперь свободный) становится вторым. При удалении процедура обращается: пока существуют строки в блоках, работа идет в памяти, а когда оба блока становятся пустыми, в первый считываются данные из файлового стека блоков.

Дек. Эту структуру можно рассматривать как обобщение стека за счет возможности добавлять и удалять элементы "с обоих концов" набора данных, т.е. имеется два текущих элемента, к которым разрешен доступ и которые называются "начало" и "конец" дека.

Дисциплина доступа к началу и концу дека аналогична дисциплине работы с вершиной стека: разрешено чтение, добавление и извлечение элементов.

3.19. Постройте реализацию дека элементов абстрактного типа Type.

Идеи реализации. Будем размещать элементы в выделенном участке памяти вплотную друг к другу и хранить два указателя — на начало и конец дека. При добавлении или удалении элементов эти указатели будем перемещать на позицию следующего или предыдущего элементов. В этом случае для корректной работы дека нужно формально "замкнуть" левую и правую границы выделенной области: при необходимости новая позиция "перебрасывается" с позиции за концом области в начало, или с позиции перед началом — в конец. В итоге массив оказывается логически замкнутым в кольцо. Если принять, что начало, конец выделенной области памяти, голова, хвост и размер дека задаются переменными

```
Type  *begmem, *endmem;
Type  *head, *tail;
int    size;
```

то процедуры определения следующего и предыдущего элементов могут выглядеть, например, так