Лекции по C/C++: динамические структуры данных

Мы знаем, что элементами структур могут быть скалярные величины, массивы или другие структуры. При этом, также как для простых типов данных, возможна организация массивов из структур. Сделав эти массивы динамическими, мы можем реализовать любые динамические структуры данных. В таком случае, пользуясь средствами динамического распределения памяти из "кучи", программа может изменять текущую размерность набора структур.

При современной организации адресации оперативной памяти, теоретически можно выделить три основных класса динамических структур, отличающихся между собой двумя характеристиками:

- Имеет ли значение для доступа к данным физический порядок расположения элементов в оперативной памяти?
- Возможны ли произвольная вставка и удаление элементов в любом месте последовательности?

Название класса динамических структур	Значим ли физический порядок элементов в ОП	Возможны ли произвольная вставка/ удаление элементов	Примеры реализации структуры
Упорядоченные совокупности	Да	Нет	Динамические массивы
Неупорядоченные совокупности (коллекции)	Нет	Да	Списки, Деревья, Хэши
Последовательности	Нет	Нет	Стеки, очереди

Комбинация характеристик "Да"-"Да" в данном случае противоречила бы "линейной" адресации памяти – в конце концов, чтобы физически добавить элемент в середину массива, мы должны будем сдвинуть в оперативной памяти все его элементы, начиная с точки вставки.

Возможны 2 основных подхода к реализации динамических структур:

- Использовать готовые библиотеки среды программирования или стандартные средства языка программирования, например, библиотеку стандартных шаблонов stl из стандарта C++ или библиотеку classlib из старых реализаций Borland C++ для DOS;
- "Вручную" организовывать динамические наборы структур, связанных между собой *указателями на свой структурный тип*.

Именно второй подход рассматривается в этой теме, а первый будет изучаться отдельно.

Опишем обычную структуру "студент", подобную тем, что рассмотрены в лекции по структурам:

```
#define MAX 30 /* максимальная длина фамилии */
#define ST 4 /* максимальное количество студентов */
struct student {
unsigned char f[MAX]; /* фамилия */
int ekz[4]; /* 4 оценки за сессию */
};
//...
student group[ST];      //Размер списка фиксирован заранее!
```

Недостаток такого подхода очевиден – у нас нет средств добавить в список (ST+1)-го студента. Мы могли бы просто описать указатель `student *group`, сделав наш массив студентов динамическим и выделяя под него память так же, как для скалярного массива. Но

вся проблема состоит в том, что когда наши структуры становятся достаточно громоздкими, а отношения между ними – сложнее простого следования друг за другом, становится крайне невыгодно обходиться одними массивами. Например, при сортировке элементов массива требуется физическая перестановка значений его элементов в памяти (а если каждый элемент – большая структура?). Время поиска в массиве нужного элемента всегда растёт линейно по отношению к количеству элементов в нём, а для более сложных структур данных это может быть не так. Кроме того, отношения между моделируемыми объектами обычно таковы, что не укладываются в простую упорядоченную совокупность (вспомните примеры из теории баз данных).

Основным механизмом моделирования коллекций и последовательностей служит включение в состав описывающей объект структуры указателя на свой же структурный тип, который будет хранить адреса одного или нескольких объектов, связанных с текущим. Конкретное количество этих указателей программист определяет, исходя из возможного количества связей любого конкретного объекта с остальными. Так, для простейшего односвязного списка, каждый элемент которого связан с одним следующим за ним элементом, нам достаточно было бы добавить в структуру один-единственный указатель:

```
struct student {
 unsigned char f[MAX];
 int ekz[4];
 student *next;
};
```

Указатель `next` будет просто хранить адрес того места в ОП, где находятся данные о следующем студенте.

Таким образом, один указатель может моделировать одну связь. Для двусвязного списка, каждый элемент которого связан с одним последующим и одним предыдущим элементом, нам понадобилось бы включить в структуру 2 указателя, на предыдущего и следующего студента:

```
student *prev, *next;
```

В случае бинарного дерева, каждый узел которого имеет "левого" и "правого" потомков, добавление к структуре было бы точно таким же:

```
student *left, *right;
```

Наконец, захоти мы по каким-то причинам организовывать записи о студентах в произвольное дерево, похожее на дерево папок Windows (каждый узел может иметь произвольное количество узлов-потомков), нам пришлось бы использовать указатель на указатель, хранящий адрес динамического массива адресов узлов-потомков:

```
student **childs;
```

Вернёмся к моделированию односвязного списка. Примем, что глобальный указатель `student *head`, описанный в программе, всегда показывает на начало списка, а у последнего элемента поле `next==NULL`. Тогда метод просмотра всего списка мог бы выглядеть так:

```
int show (student *head) {
 int count=0;
 while (1) {
  //вывести данные очередного узла
  printf ("\n%s", head->f);
  for (int i=0; i<4; i++)  printf (" %d",head->ekz[i]);
  count++;
  //проверка на конец списка
  if (head->next == NULL) break;
  //переход к следующему узлу
  head = head->next;
 }
 printf ("\nAll=%d",count);
```

```
 return count;
}
```

Так как длина списка в нашей реализации нигде не хранится, функция `show` будет её возвращать, что можно использовать в будущем. Разумеется, это лишь один из множества возможных вариантов реализации, но суть дела всегда одинакова – получив доступ к полям текущего объекта через указатель `head`, мы затем переставляем его локальную копию на адрес следующего объекта, пользуясь как раз тем, что в структуру включёно поле-указатель соответствующего типа:

```
head = head->next;
```

Напишем небольшую демо-программу, которая позволит нам прочитать данные списка и показать его на экране. Записи для списка будут браться из файла `data.txt`, расположенного в текущей папке и имеющего простой формат "одна строка – одна запись", например, такого:

```
Ivanov 5 4 4 5
Petrov 3 4 4 3
Sidorov 3 4 4 5
```

Текст функции `main`:

```
int main (void) {
 FILE *f=fopen("data.txt","rt");
 if (f==NULL) {
  printf ("\nFile data.txt not found"); exit(1);
 }
 char buf[40]; //буфер
 student *current = NULL, //текущий элемент
  *prev = NULL, //есть ли предыдущий?
  *head; //начало списка
 do {
  fgets (buf, 40, f);
  if (strlen(buf)<10) continue; //Убираем слишком короткие строки
  if (current!=NULL) prev=current; //если был тек.эл-т, запомнили его
  current = (student *) malloc(sizeof(student));
  if (current==NULL) {
   printf ("\nNo memory"); exit(2);
  }
  current->next = NULL; //добавляемый эл-т пометили как последний
  if (prev!=NULL) prev->next = current;
    //Если добавляемый эл-т current не первый,
    //предыдущий эл-т заставили показывать на него
  else head = current;
    //Иначе запомнили, что он – начало списка
  sscanf (buf,"%s %d %d %d %d",current->f,
   &current->ekz[0], &current->ekz[1],
   &current->ekz[2], &current->ekz[3]);
 } while (!feof(f));
 show (head);
 fclose (f);

 fflush (stdin); getchar (); return 0;
}
```

К программе должны быть добавлены обычные заголовки:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
using namespace std;
#define MAX 30 /* максимальная длина фамилии */
```

Здесь мы добавляли записи в список последовательно по мере чтения их из файла, но существенно то, что цикл прохождения по списку, реализованный в функции `show`, вовсе не предполагает физически последовательного расположения данных в ОП! Например, мы можем реализовать функции для добавления записей в начало, конец списка или произвольно выбранное место списка. Оперативная память при этом будет выделяться последовательно, но истинный порядок следования элементов, "зашитый" в цепочке указателей, по которой мы перемещаемся избавляет нас от необходимости заботиться о том, в какой именно ячейке ОП "лежит" какой элемент (в отличие от массива).

Добавим к приложению метод, который находит в списке фамилию, содержащую в себе указанную строку `st`. Логично, что он вернёт указатель на найденную запись или `NULL`, если поиск невозможен или искомая фамилия не найдена:

```
student *search (student *head, char *st) {
 if (head==NULL) return NULL; //список пуст
 student *next = head;
 st=strlwr((char *)st); //чтобы поиск не зависел от регистра символов
 do {
  char *find = strstr (strlwr((char *)next->f), st);
  if (find!=NULL) return next;
  if (next->next==NULL) break;
  next = next->next;
 } while (1);
 return NULL; //не найдено
}
```

Следует помнить, что стандартная реализация функции `strlwr` преобразования строки в нижний регистр может "не справиться" с не-латиницей, тогда нужна собственная аналогичная функция или выполнение соответствующих настроек компилятора и/или локали.

Тест функции мог бы быть таким:

```
char *st="rov";
 student *ptr=head;
 do { //возможность неоднократного поиска
  ptr=search(ptr,st);
  if (ptr) {
   printf ("\n%s",ptr->f);
   ptr = ptr->next; //следующий поиск - со след.эл.-та
  }
 } while (ptr);
```

Напишем функцию `add1` для добавления элемента `st` в начало списка, заданного указателем `head`, и функцию `add2` для аналогичного добавления элемента в конец списка. Так как в функции будут передаваться адреса уже размещённых в ОП структур типа `student`, нам понадобится также служебная функция `copy0`, способная переписать элемент списка из исходного адреса расположения `from` в указанный адрес `to`:

```
void copy0(student *to, student *from) {
 strcpy ((char *)to->f,(char *)from->f);
 for (int i=0; i<4; i++) to->ekz[i]=from->ekz[i];
}

student *add1 (student *head,student *st) {
 student *current = (student *) malloc (sizeof(student));
 copy0 (current,st);
 if (head==NULL) current->next=NULL;
 else {
```

```
  current->next = head;
  head = current;
 }
 return current;
}

student *add2 (student *head,student *st) {
 student *last=NULL;
 if (head) {
  last=head;
  while (last->next) last=last->next;
 }
 student *current = (student *) malloc (sizeof(student));
 copy0 (current,st);
 current->next=NULL;
 if (last) last->next = current;
 return current;
}
```

Протестировать функции из main можно так:

```
 student test1 = {"Popova",4,4,4,4};
 head=add1 (head,&test1);
 student test2 = {"Vasilenko",5,5,5,4};
 add2 (head,&test2);
 show (head);
```

Обратите внимание, что `add1` возвращает новую "голову" списка, поэтому мы сохраняем её в переменной `head`, всё время работы программы служащей для этой цели.

Проверку на наличие записи `st` в списке, начинающемся с `head`, можно реализовать так:

```
int check (student *head, char *s) {
 while (head) {
  if (stricmp((char *)head->f,s)==0) return 1;
  head = head->next;
 }
 return 0;
}
```

Здесь проверяется только совпадение фамилии, с той же оговоркой для `stricmp`, что и для `strlwr` выше. Разумеется, эта функция может сканировать и часть списка, если передать вместо `head` указатель на какой-то другой элемент. Тест функции:

```
 printf ("\n%d",check(head,(char *)test1.f)); //1
 printf ("\n%d",check(head,"NeponyatnoKto")); //0
```

Реализуем сортировку списка методом вставок. Важно то, что записи не меняются местами в памяти, а только переставляются указатели. Для больших объемов данных это может быть существенно экономичней. Сортировка будет упорядочивать текущий список по алфавиту (фамилии, полю `f`) и возвращать указатель на начало измененного списка. Код этой функции нелегок для восприятия, но его тщательное изучение поможет вам разобраться с перестановкой указателей. Указатель `q` в функции обозначает элемент, с которого мы начинаем очередной шаг, `p` – очередной элемент, с которым сравниваем `q`, `pr` – предыдущий элемент, указатель `ph` служит для контроля на конец списка (по-прежнему предполагаем, что у последнего элемента ссылка на следующий должна быть равна `NULL`). Стандартная функция `stricmp` вернёт значение больше нуля, если 1-я строка-аргумент "больше" 2-й, то есть, следует за ней по алфавиту и их нужно переставить.

```
student *sort (student *ph) {
 student *q,*out=NULL,*p,*pr; //out - выход - сначала пуст
```

```
 while (ph !=NULL) { //пока не конец входного списка
  q = ph; ph = ph->next; //исключить очередной элемент
  for (p=out,pr=NULL;
       p!=NULL && stricmp((char *)q->f,(char *)p->f)>0;
       pr=p,p=p->next) ;
   //ищем, куда включить очередной элемент - тут stricmp
   //задает критерий сравнения элементов, в вашей задаче м.б. другой
  if (pr==NULL) { q->next=out; out=q; } //включение в начало
  else { q->next=p; pr->next=q; } //или после предыдущего
 }
 return out;
}
```

Так как функция возвращает указатель на начало отсортированного списка, не забудем при вызове сохранить его в `head`:

```
head = sort (head);
show (head);
```

Удаление из списка будем выполнять по фамилии студента (в реальности следует по некоторому первичному ключу записи). Метод удаления будет возвращать указатель на "голову" списка, учитывая случай удаления первого элемента.

```
student *exclude (student *head, char *f) {
 if (head==NULL) return NULL;
 student *current = head, //текущая запись
         *start = head, //начало списка
         *prev = NULL; //предыдущая запись
 while (current) {
  if (stricmp((char *)current->f,f)==0) {
   if (prev) prev->next = current->next;
   else start=current->next;
   free(current); //1
   break; //2
  }
  prev = current;
  current = current->next;
 }
 return start;
}
```

Обратим внимание на операторы, помеченные комментариями 1 и 2.

1. Записи для студентов Popova и Vasilenko были добавлены как локальные переменные стека `test1` и `test2`, а не динамические переменные "кучи", которые получаются при распределении памяти оператором `new` или си-функциями `malloc`/`calloc`. Попытка удаления этих переменных оператором `delete` или функцией `free` является, в общем случае, некорректным действием и может привести к краху программы. Соблюдайте следующие правила при работе с динамическими данными:

- не пытайтесь освободить память, которую вы не выделяли явно, то есть, не применяйте `delete` или `free` к данным, для которых явно не выполнялся оператор `new` или функции `malloc`/`calloc`;
- не смешивайте в одной программе разные способы выделения/освобождения динамической памяти, то есть, `new`/`delete` и `malloc`/`free`;
- помните, что наиболее потенциально опасное действие при работе с динамическими данными – не выделение, а именно освобождение ОП, проверяйте в первую очередь `delete` и `free`.

2. После удаления одной подходящей записи функция не продолжает поиск следующей записи, а завершает работу.

Тест метода:

```
head = exclude (head,"Petrov");
show (head);
```

Вообще говоря, для односвязного списка, в отличие от стека, "физическое" удаление данных из памяти – трудоёмкая задача, такая же, как для массива. Часто физическое удаление заменяют перестановкой указателей, аналог которой – "пометка записи на удаление" в СУБД, физически ничего не удаляющая. На практике время от времени для таких данных, в которых из-за множественных удалений образуются "дырки", следует выполнять физическую "упаковку" файлов с данными.

Приложение. Процедурно-ориентированные коды для реализации динамических структур данных: "Односвязный список", "Двусвязный список", "Очередь", "Стек", "Бинарное дерево".

Это 5 разных программ, обратите внимание на горизонтальные разделители. Проверено в Visual Studio 2010, должно работать и в 2015.

```
=================================================================
/* Процедурно-ориентированная реализация односвязного списка */
#define _CRT_SECURE_NO_WARNINGS
struct list {
 struct list * next;
 int   info;
};

#include <locale.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <malloc.h>

/* INPUT_info() - ввод с клавиатуры информации */
int input_info(struct list * ptr) {
 fflush(stdin);
 scanf("%d", &(ptr -> info));
 return ptr->info;
}

/* PRINTON() - Вывод информации на экран. Возвращает
   указатель на следующий элемент списка. */
struct list * printOn(struct list * ptr) {
  printf("%d ", ptr -> info);
  return (ptr -> next);
}

/* DISPLAY() - Выводит информацию во всех элементах списка.
   Сканирует список от начала в конец. Возвращает указатель на
   последний элемент. */
struct list * display(struct list * head) {
 struct list * ret;
 if (head == NULL) { /* если список пуст */
  puts("Список пуст");
  return NULL;
 }
 do {
  printOn(head);
```

```
  ret = head;
  head = head -> next;
 } while(head  != NULL);
 return (ret);
}

/* DESTROY() - Освобождает память, выделенную под элемент списка. */
void destroy(struct list * element) {
 free(element);
}

/* HASMEMBER() - Проверяет, есть ли в списке, элемент с
   идентичными полями.  Список сканируется от начала в конец. */
int hasMember(struct list * head, struct list * work) {
 while(head != NULL) { /* цикл сканирования списка */
  if (head->info  == work -> info) return 1;
  head = head -> next;
 }
 return(0);
}

/* ADD() - Помещает элемент new_ptr в список head в порядке
   возрастания чисел (поля info). */
int add(struct list ** head, struct list * new_ptr) {
  struct list * first, * second;
  if ((*head) == NULL) { /* если список пуст */
   (* head) = new_ptr;
   new_ptr -> next = NULL;
   return 0;
  }
  if ((*head) -> next == NULL) { /* если в нем всего один элемент */
   if( (*head) -> info >  new_ptr -> info) {
    /* новый элемент становится первым в списке */
    second = (*head); /* сохраним указатель на второй */
    (*head) = new_ptr;
    new_ptr -> next = second;
    second -> next = NULL;
   }
   else { /* новый элемент становится в конец списка */
    (*head) -> next = new_ptr;
    new_ptr -> next = NULL;
   }
   return 1;
  }
  else {
   if ( (*head) -> info > new_ptr -> info) {
    /* новый эл-т ставится первым в списке */
    second = (*head); /* сохраним указатель на второй */
    (*head) = new_ptr;
    new_ptr -> next = second;
    return 4;
   }
   first  = (* head);
   second = first -> next;
   while (first -> next != NULL) { /* цикл поиска места в списке  */
    if ( first  -> info <= new_ptr -> info &&
```

```
         second -> info >= new_ptr -> info) {
     /* вставляем элемент между first и second */
     first -> next   = new_ptr;
     new_ptr -> next = second;
     return 2;
    }
    first = second;
    second = first -> next;
   }
   /* если добрались сюда, элемент ставим в конец списка */
   first -> next    = new_ptr;
   new_ptr  -> next = NULL;
   return 3;
  }
}

/* DETACH() - Удаляет элемент element из списка head */
void detach(struct list ** head, struct list * element) {
 struct list * prev;
 if(* head == NULL) return; /* если список пуст */
 if((*head) == element) { /* если удаляемый эл-т - первый       */
  (* head) = element -> next;
  destroy(element);
  return;
 }
 /* Цикл поиска предыдущего для element элемента. */
 prev = (* head);
 while((prev -> next) != element)
  prev = prev -> next; /* на выходе из цикла - адрес предыдущ*/
 prev -> next = element -> next;
 destroy(element);
 return;
}

int main(void) {
 char ch;
 struct list * head;   /* указатель на начало списка          */
 struct list * new_ptr;/* указатель на новый элемент списка   */
 struct list * cur;    /* указатель на текущий элемент списка */
 struct list work;
 head = NULL;          /* вначале список пуст                 */
 setlocale(LC_ALL,"Rus"); SetConsoleCP(1251); SetConsoleOutputCP(1251);
 puts ("Вводите целые числа для списка, 0 - завершение ввода");
 while(input_info(&work) != 0) { /* цикл ввода описаний */
  if (hasMember(head, &work) != 1) {
   if((new_ptr = (struct list *)malloc(sizeof(struct list))) == NULL) {
    puts("Прием информации завершен: нет памяти");
    break;
   }
   new_ptr -> info = work.info;/* копируем введен. информацию */
   add(&head, new_ptr);   /* добавляем элемент в список  */
  }
 }
 if (head == NULL)              /* список остался пустым ?     */
  return 1;                   /* да, завершение программы    */
 puts("*** СВЯЗАННЫЙ СПИСОК ПОСЛЕ ЗАВЕРШЕНИЯ ВВОДА ***");
```

```
 display(head);                 /* вывод всех элем-тов списка  */
 /* Цикл выборочного удаления элементов, начиная с первого. */
 cur = head;
 puts("*** ВЫБОРОЧНОЕ УДАЛЕНИЕ ЭЛЕМЕНТОВ СПИСКА ***");
 do {
  new_ptr = printOn(cur);  /* вывод элемента списка       */
  printf("Удаляете элемент ? (Y/N) ");
  if ((ch = getch()) == 'Y' || ch == 'y') {
   puts("Yes");
   detach(&head, cur);   /* удаление элемента из спсика */
  }
  else puts("No");
  cur = new_ptr;
 }
 while(new_ptr != NULL);
  puts("*** СВЯЗАННЫЙ СПИСОК ПОСЛЕ ВЫБОРОЧНОГО УДАЛЕНИЯ ***");
 display(head);                 /* вывод всех элем-тов списка  */
 fflush (stdin); getchar();
 return 0;
}

=====================================================================
/* Процедурно-ориентированная реализация двусвязных списков
   Элемент списка - динамическая строковая переменная
*/
#define _CRT_SECURE_NO_WARNINGS
#include <locale.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <malloc.h>

typedef struct list {
 list *next, *prev;
 char *info;
};
char buffer[80];

char *input_info(list *ptr) {
 fflush(stdin);
 scanf("%s", &buffer[0]);
 if (buffer[0]=='0') return NULL;
 ptr->info=(char *)malloc((strlen(buffer)+1)*sizeof(char));
 if (ptr->info == NULL) {
  puts("Прием информации завершен: нет памяти");
  return NULL;
 }
 strcpy (ptr->info,buffer);
 return ptr->info;
}

list * printOn(list * ptr) {
  printf("\n%s", ptr -> info);
  return (ptr -> next);
}
```

```
list * display(list * head) {
 list * ret;
 printf ("\n");
 if (head == NULL) { /* если список пуст */
  puts("Список пуст");
  return NULL;
 }
 do {
  printOn(head);
  ret = head;
  head = head -> next;
 } while(head  != NULL);
 printf ("\n");
 return (ret);
}

void destroy (list *element) {
 free (element->info);
 free (element);
}

int hasMember (list * head, list * work) {
 while(head != NULL) {
  if (!strcmpi (head->info,work->info)) return 1;
  head = head -> next;
 }
 return 0;
}

int add(list ** head, list * new_ptr) {
  list * first, * second;
  if ((*head) == NULL) { /* если список пуст */
   (* head) = new_ptr;
   new_ptr->next = NULL;
   new_ptr->prev = NULL;
   return 0;
  }
  if ((*head) -> next == NULL) { /* если в нем всего один элемент */
   if (strcmpi((*head)->info,new_ptr->info)>0) {
    /* новый элемент становится первым в списке */
    second = (*head); /* сохраним указатель на второй */
    (*head) = new_ptr;
    new_ptr -> next = second;
    new_ptr -> prev = NULL;
    second -> next = NULL;
    second -> prev = new_ptr;
   }
   else { /* новый элемент становится в конец списка */
    (*head) -> next = new_ptr;
    (*head) -> prev = NULL;
    new_ptr -> next = NULL;
    new_ptr -> prev = new_ptr;
   }
   return 1;
  }
```

```
  else {
   if ( strcmpi((*head)->info,new_ptr->info)>0) { /* новый эл-т ставится первым в списке *
    second = (*head);
    (*head) = new_ptr;
    new_ptr -> next = second;
    new_ptr -> prev = NULL;
    second -> prev = new_ptr;
    return 4;
   }
   first  = (* head);
   second = first -> next;
   while (first -> next != NULL) { /* цикл поиска места в списке  */
    if ( (strcmpi(first->info,new_ptr->info)<0) &&
         (strcmpi(second->info,new_ptr->info)>0) ) {
     /* вставляем элемент между first и second */
     first->next   = new_ptr;
     second->prev= new_ptr;
     new_ptr->next = second;
     new_ptr->prev = first;
     return 2;
    }
    first = second;
    second = first->next;
   }
   /* если добрались сюда, элемент ставим в конец списка */
    first->next=new_ptr;
   first->prev=second;
    new_ptr->next = NULL;
   new_ptr->prev=first;
   return 3;
  }
}

/* DETACH() - Удаляет элемент element из списка head */
void detach(list ** head, list * element) {
 list * prev;
 if(* head == NULL) return; /* если список пуст */
 if((*head) == element) { /* если удаляемый эл-т - первый       */
  (* head) = element -> next;
  destroy(element);
  return;
 }
 /* Цикл поиска предыдущего для element элемента. */
 prev = (* head);
 while((prev -> next) != element)
  prev = prev -> next; /* на выходе из цикла - адрес предыдущ*/
 prev -> next = element -> next;
 destroy(element);
 return;
}

int main(void) {
 char ch;
 list * head;   /* указатель на начало списка          */
 list * new_ptr;/* указатель на новый элемент списка   */
 list * cur;    /* указатель на текущий элемент списка */
```

```
 list work;
 head = NULL;          /* вначале список пуст                 */

 setlocale(LC_ALL,"Rus"); SetConsoleCP(1251); SetConsoleOutputCP(1251);
 puts ("Вводите строки для списка, 0 - завершение ввода");

 while(input_info(&work) != NULL) { /* цикл ввода описаний */
  if (hasMember(head, &work) != 1) {
   new_ptr = (list *)malloc(sizeof(list));
   if (new_ptr == NULL) {
    puts("Прием информации завершен: нет памяти");
    break;
   }
   new_ptr->info = work.info;/* копируем введен. информацию */
   add(&head, new_ptr);   /* добавляем элемент в список  */
   //display (head);
  }
  else {
   puts("Элемент уже есть в списке");
  }
 }
 if (head == NULL)              /* список остался пустым ?     */
  return 1;                   /* да, завершение программы    */
 puts("*** СВЯЗАННЫЙ СПИСОК ПОСЛЕ ЗАВЕРШЕНИЯ ВВОДА ***");
 display(head);                /* вывод всех элем-тов списка  */
 /* Цикл выборочного удаления элементов, начиная с первого. */
 cur = head;
 puts("\n*** ВЫБОРОЧНОЕ УДАЛЕНИЕ ЭЛЕМЕНТОВ СПИСКА ***");
 do {
  new_ptr = printOn(cur);  /* вывод элемента списка       */
  printf(" Удаляете элемент ? (Y/N) ");
  if ((ch = getch()) == 'Y' || ch == 'y') {
   puts("Yes");
   detach(&head, cur);   /* удаление элемента из спсика */
  }
  else puts("No");
  cur = new_ptr;
 }
 while(new_ptr != NULL);
  puts("*** СВЯЗАННЫЙ СПИСОК ПОСЛЕ ВЫБОРОЧНОГО УДАЛЕНИЯ ***");
 display(head);                /* вывод всех элем-тов списка  */
 fflush (stdin); getchar();
 return 0;
}

=====================================================================
/*
   Процедурно-ориентированная реализация стека
*/
#define _CRT_SECURE_NO_WARNINGS
#include <locale.h>
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>
```

```
typedef struct STACK {
 STACK * next;
 int info;
};

/* Из реализации списка в list1.cpp */
int input_info(STACK * ptr) {
 fflush(stdin);
 scanf("%d", &(ptr -> info));
 return ptr->info;
}

STACK * printOn(STACK * ptr) {
  printf("%d ", ptr -> info);
  return (ptr -> next);
}

STACK * display (STACK * head) {
 STACK * ret;
 if (head == NULL) { /* если список пуст */
  puts("Список пуст");
  return NULL;
 }
 do {
  printOn(head);
  ret = head;
  head = head -> next;
 } while(head  != NULL);
 return (ret);
}

/* Извлечь элемент с вершины стека */
STACK * pop(STACK ** head) {
 STACK * current;
 if ((*head) == NULL) /* если список пуст                        */
  return NULL;
 current = (* head);
 if (current -> next == NULL) /* если всего один элемент      */
  (* head) = NULL;
 else
  (*head)  = current -> next;
 return current;
}

/* Взять элемент с вершины, не извлекая его */
STACK * peek(STACK ** head) {
 if ((*head) == NULL) /* если список пуст                        */
  return NULL;
 else return (* head);
}

/* Добавить элемент на вершину стека */
void push (STACK ** head, STACK * newitem) {
 STACK * old_head;
 if ((* head) == NULL) { /* список пуст                            */
  (* head) = newitem ;
```

```
  newitem -> next = NULL;
 }
 else {
  old_head = (* head);
  (* head) = newitem;
  newitem -> next = old_head;
 }
}

int main(void) {
 STACK * head;  /* указатель на вершину стека                  */
 STACK * newitem;   /* элемент стека                               */
 STACK work;
 head = NULL;   /* вначале стек пуст                           */
 setlocale(LC_ALL,"Rus"); SetConsoleCP(1251); SetConsoleOutputCP(1251);
 puts ("Вводите целые числа в стек, 0 - завершение ввода");

 while (input_info (&work) != 0) { /* цикл ввода описаний */
  if ((newitem = (STACK *)malloc(sizeof(STACK))) == NULL) {
   puts("Прием информации завершен: нет памяти");
   break;
  }
  newitem -> info = work.info;      /* копируем введен. информацию */
  push(&head, newitem);         /* добавляем элемент */
 }
 if (head == NULL)              /* список остался пустым ?     */
  return 1;                  /* да, завершение программы    */
 puts("\n*** СТЕК ПОСЛЕ ЗАВЕРШЕНИЯ ВВОДА ***");
 display(head);                 /* вывод всех элем-тов списка  */
 /* Цикл чтения/удаления элементов очереди, начиная с вершины */
 puts("\n*** ПОЭЛЕМЕНТНОЕ ЧТЕНИЕ ЭЛЕМЕНТОВ СТЕКА PEEK+POP ***");
 while((newitem = peek(&head)) != NULL) {
  printOn(newitem);              /* вывод элемента очереди      */
  pop(&head);
  free(newitem);
 }
 puts("\n*** СТЕК ПОСЛЕ ЧТЕНИЯ ЭЛЕМЕНТОВ ***");
 display(head);                 /* вывод всех элем-тов списка  */

 STACK n,n2;
 n.info=111; n2.info=222; push(&head, &n); push(&head, &n2);
 puts("\n*** СТЕК ПОСЛЕ НОВОГО ДОБАВЛЕНИЯ 2 ЭЛЕМЕНТОВ ПРОГРАММНО ***");
 display(head);

 fflush (stdin); getchar();
 return 0;
}

=====================================================================
/*
   Процедурно-ориентированная реализация очереди
*/
#define _CRT_SECURE_NO_WARNINGS
#include <locale.h>
#include <windows.h>
#include <stdio.h>
```

```
#include <string.h>
#include <malloc.h>

typedef struct QUEUE {
 QUEUE * next;
 int info;
};

int input_info(QUEUE * ptr) {
 fflush(stdin);
 scanf("%d", &(ptr -> info));
 return ptr->info;
}

QUEUE * printOn(QUEUE * ptr) {
  printf("%d ", ptr -> info);
  return (ptr -> next);
}

QUEUE * display (QUEUE * head) {
 QUEUE * ret;
 if (head == NULL) { /* если список пуст */
  puts("Список пуст");
  return NULL;
 }
 do {
  printOn(head);
  ret = head;
  head = head -> next;
 } while(head  != NULL);
 return (ret);
}

/* получить элемент из очереди */
QUEUE * get(QUEUE ** head) {
 QUEUE * current, * previous;
 if((*head) == NULL)    /* если список пуст */
  return NULL;
 if ((* head) -> next == NULL) {        /* если всего один элемент  */
  current = (* head);
  (* head) = NULL;
  return current;
 }
 previous  = (* head);
 current = previous -> next;
 while (current -> next != NULL) { /* цикл поиска конца    */
  previous = current;
  current = previous -> next;
 }
 previous -> next  = NULL;
 return current;
}

/* поместить элемент в очередь */
void put(QUEUE ** head, QUEUE * newitem) {
 QUEUE * old_head;
```

```
 if ((* head) == NULL) { /* список пуст */
  (* head) = newitem;
  newitem -> next = NULL;
 }
 else {
  old_head = (* head);
  (* head) = newitem;
  newitem -> next = old_head;
 }
}

int main(void) {
 QUEUE * head;          /* указатель на начало списка       */
 QUEUE * newitem;       /* элемент очереди                  */
 QUEUE work;
 head = NULL;       /* вначале список пуст              */

 setlocale(LC_ALL,"Rus"); SetConsoleCP(1251); SetConsoleOutputCP(1251);
 puts ("Вводите целые числа в очередь, 0 - завершение ввода");

 while(input_info(&work) != 0)  { /* цикл ввода описаний              */
  if ((newitem = (struct QUEUE *)malloc(sizeof(struct QUEUE))) == NULL) {
   puts("Прием информации завершен: нет памяти");
   break;
  }
  newitem -> info = work.info;       /* копируем введен. информацию */
  put(&head, newitem);       /* добавляем элемент в очередь */
 }
 if(head == NULL)       /* список остался пустым ?  */
  return 1;             /* да, завершение программы */
 puts("\n*** ОЧЕРЕДЬ ПОСЛЕ ЗАВЕРШЕНИЯ ВВОДА ***");
 display(head); /* вывод всех элементов списка  */
 /* Цикл чтения/удаления элементов очереди, начиная с первого
    пришедшего в очередь.
 */
 puts("\n*** ПОЭЛЕМЕНТНОЕ ЧТЕНИЕ ЭЛЕМЕНТОВ ОЧЕРЕДИ ***");
 while((newitem = get(&head)) != NULL) {
  printOn(newitem);             /* вывод элемента очереди   */
  free(newitem);
 }
 puts("\n*** ОЧЕРЕДЬ ПОСЛЕ ЧТЕНИЯ ЭЛЕМЕНТОВ ***");
 display(head);                 /* вывод всех элем-тов списка  */

 QUEUE n,n2;
 n.info=111; n2.info=222; put(&head, &n); put(&head, &n2);
 puts("\n*** ОЧЕРЕДЬ ПОСЛЕ НОВОГО ДОБАВЛЕНИЯ 2 ЭЛЕМЕНТОВ ПРОГРАММНО ***");
 display(head);

 fflush (stdin); getchar();
 return 0;
}

=====================================================================
/*
   Пример работы с бинарным деревом и рекурсией
*/
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <locale.h>
#include <windows.h>
#include <stdio.h>
#include <iostream>
#include <string.h>
using namespace std;

typedef struct btree {
 int val;
 btree *left,*right;
};

//----- Рекурсивный поиск в двоичном дереве---------------
// Возвращается указатель на найденную вершину
btree *Search(btree *p, int v) {
 if (p==NULL) return(NULL);                    // Ветка пустая
 if (p->val == v) return(p);                   // Вершина найдена
 if (p->val > v)                             // Сравнение с текущим
  return(Search(p->left,v));                   // Левое поддерево
 else
  return(Search(p->right,v));                  // Правое поддерево
}

//----- Включение значения в двоичное дерево--------------
// функция возвращает указатель на созданную вершину,
// либо на существующее поддерево
btree *Insert(btree *pp, btree *v) {
 if (pp == NULL) {           // Найдена свободная ветка
                             // Создать вершину дерева
  btree *q = new btree;      // и вернуть указатель
  q->val = v->val;
  q->left = q->right = NULL;
  return q;
 }
 if (pp->val == v->val) return pp;
 if (pp->val > v->val)                      // Перейти в левое или
  pp->left=Insert(pp->left,v);             // правое поддерево
 else
  pp->right=Insert(pp->right,v);
 return pp;
}

// Рекурсивный обход двоичного дерева с выводом
// значений вершин в порядке возрастания
void Scan(btree *p) {
 if (p==NULL) return;
 Scan(p->left);
 cout << p->val << endl;
 Scan(p->right);
}

// Рекурсивный обход двоичного дерева с нумерацией вершин
// снизу-вверх слева-направо, n - текущий номер вершины
int Scan2 (btree * p, int n) {
 if (p==NULL) return n;
```

```
 Scan2 (p->left,n);
 n++;
 cout << n << ") " << p->val << endl;
 n=Scan2(p->right,n) ;
 return n;
}

// Рекурсивный обход двоичного дерева с последовательной
// нумерацией вершин в возвратом указателя на вершину с заданным номером
// Глобальный счетчик вершин передается через указатель
btree *ScanNum(btree *p, int *n) {
 btree *q;
 if (p==NULL) return NULL;
 q=ScanNum(p->left,n);
 if (q!=NULL) return q;
 if ((*n)-- ==0) return p;
 return ScanNum(p->right,n);
}

int main (void) {
 int v=1;
 btree *root=NULL;
 int depth=1;
 setlocale(LC_ALL,"Rus"); SetConsoleCP(1251); SetConsoleOutputCP(1251);
 // ввод узлов
 while (1) {
  printf ("\n Введите целое число (узел дерева) или 0 для выхода:");
  fflush (stdin);
  scanf ("%d",&v);
  if (!v) break;
  btree node;
  node.val=v;
  root=Insert (root,&node);
 }
 // поиск значения
 printf ("\n введите узел для поиска:");
 fflush (stdin);
 scanf ("%d",&v);
 btree *found=Search (root,v);
 if (found) printf ("\n found %d\n",found->val);
 else printf ("\n not found %d\n",v);
 // нумерация вершин
 printf ("\nScan:\n");
 Scan (root);
 printf ("\n введите начальный номер вершины:");
 fflush (stdin);
 scanf ("%d",&v);
 // поиск по номеру вершины
 printf ("\nScan2:\n");
 Scan2 (root,v);
 printf ("\nScan&Num:\n");
 found=ScanNum (root,&v);
 if (found) printf ("\n found %d\n",found->val);
 else printf ("\n not found %d\n",v);
 fflush (stdin); getchar ();
```

```
 return 0;
}
```

Пример постановки и реализации задачи на динамические структуры данных можно также найти в этой заметке.

Оглавление всей серии

теги: программирование c++ учебное

Здесь можно оставить комментарий, обязательны только выделенные цветом поля. Не пишите лишнего, и всё будет хорошо

Ваше имя:

Пароль (если желаете запомнить имя):

Любимый URL (если указываете, то вставьте полностью):

Текст сообщения (до 1024 символов):

Введите код сообщения: 3983

Добавить Сброс

27.01.2016, 13:58; рейтинг: 22167