**Лабораторная работа №4**

Индексирование записей. Простой/ сложный индекс.

**Цель:** изучить индексирование структур данных, научиться использовать указатели.

**ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**

Единственно возможные операции над структурами - это их копирование, присваивание, взятие адреса с помощью & и осуществление доступа к ее элементам. Копирование и присваивание также включают в себя передачу функциям аргументов и возврат ими значений. Структуры нельзя сравнивать. Инициализировать структуру можно списком константных значений ее элементов; автоматическую структуру также можно инициализировать присваиванием.

Чтобы лучше познакомиться со структурами, напишем несколько функций, манипулирующих точками и прямоугольниками. Возникает вопрос: а как передавать функциям названные объекты? Существует по крайней мере три подхода: передавать компоненты по отдельности, передавать всю структуру целиком и передавать указатель на структуру. Каждый подход имеет свои плюсы и минусы.

Первая функция, *makepoint*, получает два целых значения и возвращает структуру *point*.

/\* makepoint: формирует точку по компонентам x и y \*/

struct point makepoint(int х, int у)

{

struct point temp;

temp.x = х;

temp.у = у;

return temp;

}

Заметим: никакого конфликта между именем аргумента и именем элемента структуры не возникает; более того, сходство подчеркивает родство обозначаемых им объектов.

Теперь с помощью *makepoint* можно выполнять динамическую инициализацию любой структуры или формировать структурные аргументы для той или иной функции:

struct rect screen;

struct point middle;

struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);

screen.pt2 = makepoint(XMAX, YMAX);

middle = makepoint((screen.pt1.x + screen.pt2.x)/2,

(screen.pt1.y + screen.pt2.y)/2);

Следующий шаг состоит в определении ряда функций, реализующих различные операции над точками. В качестве примера рассмотрим следующую функцию:

/\* addpoint: сложение двух точек \*/

struct point addpoint(struct point p1, struct point p2)

{

p1.x += p2.x;

p1.y += p2.y;

return p1;

}

Здесь оба аргумента и возвращаемое значение - структуры. Мы увеличиваем компоненты прямо в р1 и не используем для этого временной переменной, чтобы подчеркнуть, что структурные параметры передаются по значению так же, как и любые другие.

В качестве другого примера приведем функцию *ptinrect*, которая проверяет: находится ли точка внутри прямоугольника, относительно которого мы принимаем соглашение, что в него входят его левая и нижняя стороны, но не входят верхняя и правая.

/\* ptinrect: возвращает 1, если p в r, и 0 в противном случае \*/

int ptinrect(struct point р, struct rect r)

{

return p.x >= r.pt1.x && p.x < r.pt2.x

&& p.y >= r.pt1.y && p.y < r.pt2.y;

}

Здесь предполагается, что прямоугольник представлен в стандартном виде, т.е. координаты точки *pt1* меньше соответствующих координат точки *pt2*. Следующая функция гарантирует получение прямоугольника в каноническом виде.

#define min(a, b) ((a) < (b) ? (a) : (b))

#define max(a, b) ((a) > (b) ? (a) : (b))

/\* canonrect: канонизация координат прямоугольника \*/

struct rect canonrect(struct rect r)

{

struct rect temp;

temp.pt1.x = min(r.pt1.x, r.pt2.x);

temp.ptl.y = min(r.pt1.y, r.pt2.у);

temp.pt2.x = max(r.pt1.x, r.pt2.x);

temp.pt2.y = max(r.pt1.y, r.pt2.y);

return temp;

}

Если функции передается большая структура, то, чем копировать ее целиком, эффективнее передать указатель на нее. Указатели на структуры ничем не отличаются от указателей на обычные переменные. Объявление

struct point \*pp;

сообщает, что *pp* - это указатель на структуру типа *struct point*. Если *pp* указывает на структуру *point*, то *\*pp* - это сама структура, а *(\*pp).x* и *(\*pp).y* - ее элементы. Используя указатель *pp*, мы могли бы написать

struct point origin, \*pp;

pp = &origin;

printf("origin: (%d,%d)\n", (\*pp).x, (\*pp).y);

Скобки в *(\*pp).x* необходимы, поскольку приоритет оператора **.** выше, чем приоритет **\***. Выражение *\*pp.x* будет проинтерпретировано как *\*(pp.x)*, что неверно, поскольку *pp.x* не является указателем.

Указатели на структуры используются весьма часто, поэтому для доступа к ее элементам была придумана еще одна, более короткая форма записи. Если *p* — указатель на структуру, то

р->*элемент-структуры*

есть ее отдельный элемент. (Оператор **->** состоит из знака -, за которым сразу следует знак >.) Поэтому *printf* можно переписать в виде

printf("origin: (%d,%d)\n", pp->х, pp->y);

Операторы **.** и **->** выполняются слева направо. Таким образом, при наличии объявления

struct rect r, \*rp = &r;

следующие четыре выражения будут эквивалентны:

r.pt1.x

rp->pt1.x

(r.pt1).x

(rp->pt1).x

Операторы доступа к элементам структуры **.** и **->** вместе с операторами вызова функции **()** и индексации массива **[]** занимают самое высокое положение в иерархии приоритетов и выполняются раньше любых других операторов. Например, если задано объявление

struct {

int len;

char \*str;

} \*p;

то

++p->len

увеличит на 1 значение элемента структуры *len*, а не указатель *p*, поскольку в этом выражении как бы неявно присутствуют скобки: *++(p->len)*. Чтобы изменить порядок выполнения операций, нужны явные скобки. Так, в *(++р)->len*, прежде чем взять значение *len*, программа прирастит указатель *p*. В *(р++)->len* указатель *p* увеличится после того, как будет взято значение *len* (в последнем случае скобки не обязательны).

По тем же правилам *\*p->str* обозначает содержимое объекта, на который указывает *str*; *\*p->str++* прирастит указатель *str* после получения значения объекта, на который он указывал (как и в выражении *\*s++*), *(\*p->str)++* увеличит значение объекта, на который указывает *str*; *\*p++->str* увеличит *p* после получения того, на что указывает *str*.

**Массивы структур**

Рассмотрим программу, определяющую число вхождений каждого ключевого слова в текст Си-программы. Нам нужно уметь хранить ключевые слова в виде массива строк и счетчики ключевых слов в виде массива целых. Один из возможных вариантов - это иметь два параллельных массива:

char \*keyword[NKEYS];

int keycount[NKEYS];

Однако именно тот факт, что они параллельны, подсказывает нам другую организацию хранения - через массив структур. Каждое ключевое слово можно описать парой характеристик

char \*word;

int count;

Такие пары составляют массив. Объявление

struct key {

char \*word;

int count;

} keytab[NKEYS];

объявляет структуру типа *key* и определяет массив *keytab*, каждый элемент которого является структурой этого типа и которому где-то будет выделена память. Это же можно записать и по-другому:

struct key {

char \*word;

int count;

};

struct key keytab[NKEYS];

Так как *keytab* содержит постоянный набор имен, его легче всего сделать внешним массивом и инициализировать один раз в момент определения. Инициализация структур аналогична ранее демонстрировавшимся инициализациям - за определением следует список инициализаторов, заключенный в фигурные скобки:

struct key {

char \*word;

int count;

} keytab[] = {

"auto", 0,

"break", 0,

"case", 0,

"char", 0,

"const", 0,

"continue", 0,

"default", 0,

/\*...\*/

"unsigned", 0,

"void", 0,

"volatile", 0,

"while", 0

};

Инициализаторы задаются парами, чтобы соответствовать конфигурации структуры. Строго говоря, пару инициализаторов для каждой отдельной структуры следовало бы заключить в фигурные скобки, как, например, в

{ "auto", 0 },

{ "break", 0 },

{ "case", 0 },

...

Однако когда инициализаторы - простые константы или строки символов и все они имеются в наличии, во внутренних скобках нет необходимости. Число элементов массива *keytab* будет вычислено по количеству инициализаторов, поскольку они представлены полностью, а внутри квадратных скобок "[]" ничего не задано.

Программа подсчета ключевых слов начинается с определения *keytab*. Программа *main* читает ввод, многократно обращаясь к функции *getword* и получая на каждом ее вызове очередное слово. Каждое слово ищется в *keytab*. Для этого используется функция бинарного поиска. Список ключевых слов должен быть упорядочен в алфавитном порядке.

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAXWORD 100

int getword(char \*, int);

int binsearch(char \*, struct key \*, int);

/\* подсчет ключевых слов Си \*/

main()

{

int n;

char word[MAXWORD];

while(getword(word, MAXWORD) != EOF)

if (isalpha(word[0]))

if ((n = binsearch(word, keytab, NKEYS)) >= 0)

keytab[n].count++;

for (n = 0; n < NKEYS; n++)

if (keytab[n].count > 0)

printf("%4d %s\n", keytab[n].count, keytab[n].word);

return 0;

}

/\* binsearch: найти слово в tab[0]...tab[n-1] \*/

int binsearch(char \*word, struct key tab[], int n)

{

int cond;

int low, high, mid;

low = 0;

high = n-1;

while (low <= high) {

mid = (low + high)/2;

if ((cond = strcmp(word, tab[mid].word)) < 0)

high = mid - 1;

else if (cond > 0)

low = mid + 1;

else

return mid;

}

return -1;

}

Чуть позже мы рассмотрим функцию *getword*, а сейчас нам достаточно знать, что при каждом ее вызове получается очередное слово, которое запоминается в массиве, заданном первым аргументом.

NKEYS - количество ключевых слов в *keytab*. Хотя мы могли бы подсчитать число таких слов вручную, гораздо легче и безопасней сделать это с помощью машины, особенно если список ключевых слов может быть изменен. Одно из возможных решений — поместить в конец списка инициализаторов пустой указатель (NULL) и затем перебирать в цикле элементы *keytab*, пока не встретится концевой элемент.

Но возможно и более простое решение. Поскольку размер массива полностью определен во время компиляции и равен произведению количества элементов массива на размер его отдельного элемента, число элементов массива можно вычислить по формуле

*размер* keytab / *размер* struct key

В Си имеется унарный оператор **sizeof**, который работает во время компиляции. Его можно применять для вычисления размера любого объекта. Выражения

sizeof *объект*

и

sizeof (*имя типа*)

выдают целые значения, равные размеру указанного объекта или типа в байтах. (Строго говоря, **sizeof** выдает беззнаковое целое, тип которого **size\_t** определена заголовочном файле **<stddef.h>**.) Что касается объекта, то это может быть переменная, массив или структура. В качестве имени типа может выступать имя базового типа (*int*, *double* ...) или имя производного типа, например структуры или указателя.

В нашем случае, чтобы вычислить количество ключевых слов, размер массива надо поделить на размер одного элемента. Указанное вычисление используется в инструкции *#define* для установки значения NKEYS:

#define NKEYS (sizeof keytab / sizeof(struct key))

Этот же результат можно получить другим способом - поделить размер массива на размер какого-то его конкретного элемента:

#define NKEYS (sizeof keytab / sizeof keytab[0])

Преимущество такого рода записей в том, что их не надо коppектировать при изменении типа.

Поскольку препроцессор не обращает внимания на имена типов, оператор **sizeof** нельзя применять в **#if**. Но в **#define** выражение препроцессором не вычисляется, так что предложенная нами запись допустима.

Теперь поговорим о функции *getword*. Мы написали *getword* в несколько более общем виде, чем требуется для нашей программы, но она от этого не стала заметно сложнее. Функция *getword* берет из входного потока следующее "слово". Под словом понимается цепочка букв-цифр, начинающаяся с буквы, или отдельный символ, отличный от символа-разделителя. В случае конца файла функция возвращает EOF, в остальных случаях ее значением является код первого символа слова или сам символ, если это не буква.

/\* getword: принимает следующее слово или символ из ввода \*/

int getword (char \*word, int lim)

{

int c, getch(void);

void ungetch(int);

char \*w = word;

while (isspace(c = getch()))

;

if (c != EOF)

\*w++ = c;

if (!isalpha(c)) {

\*w = '\0';

return c;

}

for ( ; --lim > 0; w++)

if (!isalnum(\*w = getch())) {

ungetch(\*w);

break;

}

\*w = '\0';

return word[0];

}

Функция *getword* обращается к *getch* и *ungetch*. По завершении набора букв-цифр оказывается, что *getword* взяла лишний символ. Обращение к *ungetch* позволяет вернуть его назад во входной поток. В *getword* используются также **isspace** - для пропуска символов-разделителей, **isalpha** - для идентификации букв и **isalnum** - для распознавания букв-цифр. Все они описаны в стандартном заголовочном файле **<ctype.h>**.

# **Указатели на структуру**

Для иллюстрации некоторых моментов, касающихся указателей на структуры и массивов структур, перепишем программу подсчета ключевых слов, пользуясь для получения элементов массива вместо индексов указателями.

Внешнее объявление массива *keytab* остается без изменения, a *main* и *binsearch* нужно модифицировать.

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAXWORD 100

int getword(char \*, int);

struct key \*binsearch(char \*, struct key \*, int);

/\* подсчет ключевых слов Си: версия с указателями \*/

main()

{

char word[MAXWORD];

struct key \*p;

while (getword(word, MAXWORD) != EOF)

if (isalpha(word[0]))

if ((p = binsearch(word, keytab, NKEYS)) != NULL)

p->count++;

for (p = keytab; p < keytab + NKEYS; p++)

if (p->count > 0)

printf("%4d %s\n", p->count, p->word);

return 0;

}

/\* binsearch: найти слово word в tab[0]...tab[n-1] \*/

struct key \*binsearch(char \*word, struct key \*tab, int n)

{

int cond;

struct key \*low = &tab[0];

struct key \*high = &tab[n];

struct key \*mid;

while (low < high) {

mid = low + (high - low) / 2;

if ((cond = strcmp(word, mid->word)) < 0)

high = mid;

else if (cond > 0)

low = mid + 1;

else

return mid;

}

return NULL;

}

Некоторые детали этой программы требуют пояснений. Во-первых, описание функции *binsearch* должно отражать тот факт, что она возвращает указатель на *struct key*, а не целое, это объявлено как в прототипе функции, так и в функции *binsearch*. Если *binsearch* находит слово, то она выдает указатель на него, в противном случае она возвращает NULL. Во-вторых, к элементам *keytab* доступ в нашей программе осуществляется через указатели. Это потребовало значительных изменений в *binsearch*. Инициализаторами для *low* и *high* теперь служат указатели на начало и на место сразу после конца массива. Вычисление положения среднего элемента с помощью формулы

mid = (low + high) / 2 /\* НЕВЕРНО \*/

не годится, поскольку указатели нельзя складывать. Однако к ним можно применить операцию вычитания, и так как *high-low* есть число элементов, присваивание

mid = low + (high-low) / 2

превратит *mid* в указатель на элемент, лежащий посередине между *low* и *high*.

Самое важное при переходе на новый вариант программы - сделать так, чтобы не генерировались неправильные указатели и не было попыток обратиться за пределы массива. Проблема в том, что и *&tab[-1]*, и *&tab[n]* находятся вне границ массива. Первый адрес определенно неверен, нельзя также осуществить доступ и по второму адресу. По правилам языка, однако, гарантируется, что адрес ячейки памяти, следующей сразу за концом массива (т. е. *&tab[n]*), в арифметике с указателями воспринимается правильно.

В главной программе *main* мы написали

for (р = keytab; р < keytab + NKEYS; р++)

Если *p* - это указатель на структуру, то при выполнении операций с *р* учитывается размер структуры. Поэтому *р++* увеличит р на такую величину, чтобы выйти на следующий структурный элемент массива, а проверка условия вовремя остановит цикл.

Не следует, однако, полагать, что размер структуры равен сумме размеров ее элементов. Вследствие выравнивания объектов разной длины в структуре могут появляться безымянные "дыры". Например, если переменная типа *char* занимает один байт, а *int* - четыре байта, то для структуры

struct {

char с;

int i;

};

может потребоваться восемь байтов, а не пять. Оператор *sizeof* возвращает правильное значение.

Наконец, несколько слов относительно формата программы. Если функция возвращает значение сложного типа, как, например, в нашем случае она возвращает указатель на структуру:

struct key \*binsearch(char \*word, struct key \*tab, int n)

то "высмотреть" имя функции оказывается совсем не просто. В подобных случаях иногда пишут так:

struct key \*

binsearch(char \*word, struct key \*tab, int n)

Какой форме отдать предпочтение - дело вкуса. Выберите ту, которая больше всего вам нравится, и придерживайтесь ее.

**Структуры со ссылками на себя**

Предположим, что мы хотим решить более общую задачу - написать программу, подсчитывающую частоту встречаемости для любых слов входного потока. Так как список слов заранее не известен, мы не можем предварительно упорядочить его и применить бинарный поиск. Было бы неразумно пользоваться и линейным поиском каждого полученного слова, чтобы определять, встречалось оно ранее или нет - в этом случае программа работала бы слишком медленно. (Более точная оценка: время работы такой программы пропорционально квадрату количества слов.) Как можно организовать данные, чтобы эффективно справиться со списком произвольных слов?

Один из способов - постоянно поддерживать упорядоченность уже полученных слов, помещая каждое новое слово в такое место, чтобы не нарушалась имеющаяся упорядоченность. Делать это передвижкой слов в линейном массиве не следует, - хотя бы потому, что указанная процедура тоже слишком долгая. Вместо этого мы воспользуемся структурой данных, называемой бинарным деревом.

В дереве на каждое отдельное слово предусмотрен "узел", который содержит:

- указатель на текст слова;

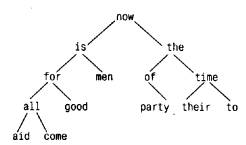
- счетчик числа встречаемости;

- указатель на левый сыновний узел;

- указатель на правый сыновний узел.

У каждого узла может быть один или два сына, или узел вообще может не иметь сыновей.

Узлы в дереве располагаются так, что по отношению к любому узлу левое поддерево содержит только те слова, которые лексикографически меньше, чем слово данного узла, а правое - слова, которые больше него. Вот как выглядит дерево, построенное для фразы "*now is the time for all good men to come to the aid of their party*" ("настало время всем добрым людям помочь своей партии"), по завершении процесса, в котором для каждого нового слова в него добавлялся новый узел:



Чтобы определить, помещено ли уже в дерево вновь поступившее слово, начинают с корня, сравнивая это слово со словом из корневого узла. Если они совпали, то ответ на вопрос — положительный. Если новое слово меньше слова из дерева, то поиск продолжается в левом поддереве, если больше, то — в правом. Если же в выбранном направлении поддерева не оказалось, то этого слова в дереве нет, а пустующая позиция, говорящая об отсутствии поддерева, как раз и есть то место, куда нужно "подвесить" узел с новым словом. Описанный процесс по сути рекурсивен, так как поиск в любом узле использует результат поиска в одном из своих сыновних узлов. В соответствии с этим для добавления узла и печати дерева здесь наиболее естественно применить рекурсивные функции.

Вернемся к описанию узла, которое удобно представить в виде структуры с четырьмя компонентами:

struct tnode { /\* узел дерева \*/

char \*word; /\* указатель на текст \*/

int count; /\* число вхождений \*/

struct tnode \*left; /\* левый сын \*/

struct tnode \*right; /\* правый сын \*/

};

Приведенное рекурсивное определение узла может показаться рискованным, но оно правильное. Структура не может включать саму себя, но ведь

struct tnode \*left;

объявляет *left* как указатель на *tnode*, а не сам *tnode*.

Иногда возникает потребность во взаимоссылающихся структурах: двух структурах, ссылающихся друг на друга. Прием, позволяющий справиться с этой задачей, демонстрируется следующим фрагментом:

struct t {

...

struct s \*p; /\* р указывает на s \*/

};

struct s {

...

struct t \*q; /\* q указывает на t \*/

}

Вся программа удивительно мала - правда, она использует вспомогательные программы вроде *getword*, уже написанные нами. Главная программа читает слова с помощью *getword* и вставляет их в дерево посредством *addtree*.

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAXWORD 100

struct tnode \*addtree(struct tnode \*, char \*);

void treeprint(struct tnode \*);

int getword(char \*, int);

/\* подсчет частоты встречаемости слов \*/

main()

{

struct tnode \*root;

char word[MAXWORD];

root = NULL;

while (getword (word, MAXWORD) != EOF)

if (isalpha(word[0]))

root = addtree(root, word);

treeprint(root);

return 0;

}

Функция *addtree* рекурсивна. Первое слово функция *main* помещает на верхний уровень дерева (корень дерева). Каждое вновь поступившее слово сравнивается со словом узла и "погружается" или в левое, или в правое поддерево с помощью рекурсивного обращения к *addtree*. Через некоторое время это слово обязательно либо совпадет с каким-нибудь из имеющихся в дереве слов (в этом случае к счетчику будет добавлена 1), либо программа встретит пустую позицию, что послужит сигналом для создания нового узла и добавления его к дереву. Создание нового узла сопровождается тем, что *addtree* возвращает на него указатель, который вставляется в узел родителя.

struct tnode \*talloc(void);

char \*strdup(char \*);

/\* addtree: добавляет узел со словом w в р или ниже него \*/

struct tnode \*addtree(struct tnode \*p, char \*w)

{

int cond;

if (р == NULL) { /\* слово встречается впервые \*/

p = talloc(); /\* создается новый узел \*/

p->word = strdup(w);

p->count = 1;

p->left = p->right = NULL;

} else if ((cond = strcmp(w, p->word)) == 0)

p->count++; /\* это слово уже встречалось \*/

else if (cond < 0) /\* меньше корня левого поддерева \*/

p->left = addtree(p->left, w);

else /\* больше корня правого поддерева \*/

p->right = addtree(p->right, w);

return p;

}

Память для нового узла запрашивается с помощью программы *talloc*, которая возвращает указатель на свободное пространство, достаточное для хранения одного узла дерева, а копирование нового слова в отдельное место памяти осуществляется с помощью *strdup*. (Мы рассмотрим эти программы чуть позже.) В тот (и только в тот) момент, когда к дереву подвешивается новый узел, происходит инициализация счетчика и обнуление указателей на сыновей. Мы опустили (что неразумно) контроль ошибок, который должен выполняться при получении значений от *strdup* и *talloc*.

Функция *treeprint* печатает дерево в лексикографическом порядке: для каждого узла она печатает его левое поддерево (все слова, которые меньше слова данного узла), затем само слово и, наконец, правое поддерево (слова, которые больше слова данного узла).

/\* treeprint: упорядоченная печать дерева р \*/

void treeprint(struct tnode \*p)

{

if (p != NULL) {

treeprint (p->left);

printf("%4d %s\n", p->count, p->word);

treeprint(p->right);

}

}

Если вы не уверены, что досконально разобрались в том, как работает рекурсия, "проиграйте" действия *treeprint* на дереве, приведенном выше. Практическое замечание: если дерево "несбалансировано" (что бывает, когда слова поступают не в случайном порядке), то время работы программы может сильно возрасти. Худший вариант, когда слова уже упорядочены; в этом случае затраты на вычисления будут такими же, как при линейном поиске. Существуют обобщения бинарного дерева, которые не страдают этим недостатком, но здесь мы их не описываем.

Прежде чем завершить обсуждение этого примера, сделаем краткое отступление от темы и поговорим о механизме запроса памяти. Очевидно, хотелось бы иметь всего лишь одну функцию, выделяющую память, даже если эта память предназначается для разного рода объектов. Но если одна и та же функция обеспечивает память, скажем) и для указателей на *char*, и для указателей на *struct tnode*, то возникают два вопроса. Первый: как справиться с требованием большинства машин, в которых объекты определенного типа должны быть выровнены (например, *int* часто должны размещаться, начиная с четных адресов)? И второе: как объявить функцию-распределитель памяти, которая вынуждена в качестве результата возвращать указатели разных типов?

Вообще говоря, требования, касающиеся выравнивания, можно легко выполнить за счет некоторого перерасхода памяти. Однако для этого возвращаемый указатель должен быть таким, чтобы удовлетворялись любые ограничения, связанные с выравниванием. Функция *alloc* не гарантирует нам любое конкретное выравнивание, поэтому мы будем пользоваться стандартной библиотечной функцией **malloc**, которая это делает.

Вопрос об объявлении типа таких функций, как **malloc**, является камнем преткновения в любом языке с жесткой проверкой типов. В Си вопрос решается естественным образом: **malloc** объявляется как функция, которая возвращает указатель на **void**. Полученный указатель затем явно приводится к желаемому типу (Замечание о приведении типа величины, возвращаемой функцией *malloc*, нужно переписать. Пример коpректен и работает, но совет является спорным в контексте стандартов ANSI/ISO 1988-1989 г. На самом деле это не обязательно (при условии что приведение *void\** к *ALMOSTANYTYPE\** выполняется автоматически) и возможно даже опасно, если *malloc* или ее заместитель не может быть объявлен как функция, возвращающая *void\**. Явное приведение типа может скрыть случайную ошибку. В другие времена (до появления стандарта ANSI) приведение считалось обязательным, что также справедливо и для C++. — Примеч. авт.). Описания *malloc* и связанных с ней функций находятся в стандартном заголовочном файле **<stdlib.h>**. Таким образом, функцию *talloc* можно записать так:

#include <stdlib.h>

/\* talloc: создает tnode \*/

struct tnode \*talloc(void)

{

return (struct tnode \*) malloc(sizeof(struct tnode));

}

Функция *strdup* просто копирует строку, указанную в аргументе, в место, полученное с помощью *malloc*:

char \*strdup(char \*s) /\* делает дубликат s \*/

{

char \*p;

p = (char \*) malloc(strlen(s)+1); /\* +1 для '\0' \*/

if (p != NULL)

strcpy(p, s);

return p;

}

Функция *malloc* возвращает NULL, если свободного пространства нет; *strdup* возвращает это же значение, оставляя заботу о выходе из ошибочной ситуации вызывающей программе.

Память, полученную с помощью *malloc*, можно освободить для повторного использования, обратившись к функции *free*.

**ПРИМЕР**

**Создание структуры Студенты.**

#include<stdio.h>

#include<string>

#include<iostream>

#include <windows.h>

using namespace std;

//объявление структуры

typedef struct student {

int rno;

char name[20]; //объявления члена структуры

char surname[20];

char second\_name[20];

struct subject { //структура внутри струкуры

int scode;

char name[20];

int mark;

}sub[3]; //объявление массива в структуре

int total;

float per;

}student;

**Пример функции сортировки**

void sort\_on\_screen() {

student\* s, s1;

FILE\* fp;

fp = fopen("mystudents1.txt", "r");

fseek(fp, 0, SEEK\_END);

int n = ftell(fp) / sizeof(student);

int i, j;

rewind(fp);

s = (student\*)calloc(n, sizeof(student));

for (i = 0; i < n; i++)

fread(&s[i], sizeof(student), 1, fp);

for (i = 0; i < n; i++) {

for (j = i + 1; j < n; j++) {

if (s[i].total < s[j].total) {

s1 = s[i];

s[i] = s[j];

s[j] = s1;

}

}

}

for (i = 0; i < n; i++) {

printf("\n%-5d%-20s%-20s%-20s", s[i].rno, s[i].name, s[i].surname, s[i].second\_name);

for (j = 0; j < 3; j++) {

printf("%4d", s[i].sub[j].mark);

}

printf("%5d%7.2f", s[i].total, s[i].per);

}

fclose(fp);

}

void sort\_in\_file() {

student\* s, s1;

FILE\* fp;

fp = fopen("mystudents1.txt", "r");

fseek(fp, 0, SEEK\_END);

int n = ftell(fp) / sizeof(student);

int i, j;

rewind(fp);

s = (student\*)calloc(n, sizeof(student));

for (i = 0; i < n; i++)

fread(&s[i], sizeof(student), 1, fp);

fclose(fp);

for (i = 0; i < n; i++) {

for (j = i + 1; j < n; j++) {

if (s[i].total < s[j].total) {

s1 = s[i];

s[i] = s[j];

s[j] = s1;

}

}

}

for (i = 0; i < n; i++) {

printf("\n%-5d%-20s%-20s%-20s", s[i].rno, s[i].name, s[i].surname, s[i].second\_name);

for (j = 0; j < 3; j++) {

printf("%4d", s[i].sub[j].mark);

}

printf("%5d%7.2f", s[i].total, s[i].per);

}

fp = fopen("mystudents1.txt", "w");

for (i = 0; i < n; i++) {

fwrite(&s[i], sizeof(student), 1, fp);

}

fclose(fp);

}

**ЗАДАНИЕ**

Создайте структуру Библиотека с полями Название, Автор, Дата издания, Издательство и присвойте каждой книге Индекс (не менее 10 объектов), чтение осуществлять из файла. Отсортируйте по индексу всю литературу и выведите на консоль самую старую по году издания книгу. (добавьте 2 функции – вывод самой новой и самой старой книги).