Структуры данных

Структура — это совокупность элементов произвольных типов. Например:

```
struct Phone
{
    char* name;
    long phoneNumber;
};
```

Структура задаёт тип данных. Обратите внимание на ; в конце определения, здесь она необходима в силу убогости синтаксиса C++.

Использование:

```
Phone somePhone;
somePhone.name = "Иванов Иван";
somePhone.phoneNumber= 1234567;
```

Можно инициализировать структуры как массивы:

```
Phone somePhone = {
    "Иванов Иван",
    1234567
};
```

Структуры можно передавать как параметры в функции, возвращать как значения из функций. А вот операторы, в т.ч. оператор сравнения, для структур не определён, его нужно определять самостоятельно (например, сравнением поэлементно, но про это как-нибудь потом расскажу).

Полезны указатели на структуры, есть специальный синтаксис обращения к полям структуры через указатель:

```
Phone *somePhone = new Phone;
somePhone->name = "Иванов Иван";
somePhone->phoneNumber= 1234567;
```

Тип структуры можно использовать сразу после того, как он появится в описании, если нам не нужен его размер:

```
struct ListElement
{
    int value;
    ListElement *next;
};
```

Если написать

```
struct ListElement
{
    int value;
    ListElement next;
};
```

Такое компилироваться не будет — размер структуры ListElement ещё не известен в момент описания поля next, так что компилятор не знает, сколько места под это поле выделить.

Ещё бывают предварительные объявления (forward declarations):

```
struct List;

struct Link {
    Link* pre;
    Link* suc;
    List* memberOf;
};

struct List {
    Link* head;
};
```

Зачем всё это. Структуры могут применяться для реализации так называемых абстрактных типов данных. АТД — это важная концепция программирования, являющаяся обобщением понятия "тип". Что такое тип — это множество значений и связанный с ним набор операций. АТД — это некоторая математическая модель и набор операций, определённый в рамках этой модели. Так же, как функция является неким обобщением встроенных операторов языка, АТД является неким обобщением встроенных (или составных) типов. АТД обычно состоит из какого-то типа данных и из операций, которые работают со значениями этого типа, при этом использующие АТД программы не должны ничего знать о внутреннем представлении данных АТД, работая со значениями АТД только с помощью его процедур (это называется принципом сокрытия деталей реализации). Процедуры и тип данных АТД должны быть описаны в каком-то одном месте программы (желательно, в отдельном модуле), чтобы если надо что-то поменять в реализации АТД, можно было быстро найти нужное место и быть уверенным, что больше нигде ничего не сломалось (это называется принципом инкапсуляции). Обратите внимание, что инкапсуляцию и сокрытие деталей реализации путают очень часто, это два разных и, по сути, независимых принципа! Следующим обобщением АТД, кстати, являются классы, но мы их будем рассматривать только в следующем семестре.

Начнём с простого примера АТД — типа данных "список". Список — это последовательность элементов определённого типа (типа элемента списка), которые можно линейно упорядочить в соответствии с их позицией в списке. Со списком непосредственно связан тип данных "позиция" — как способ эффективно указать какой-либо элемент в списке. АТД "Список" может иметь, например, такие операции:

- *end(l)* — вернуть позицию, следующую за последним элементом списка l (типа как feof() для файла).

- *insert(x, p, l)* — добавить элемент x в позицию p списка l. Например, если есть список a1, ..., a_p, a_p+1, ..., a_l, то будет a1, ..., x, a_p, ..., a_l
- *locate(x, l)* — возвращает позицию элемента x в списке l.
- *retrieve(p, l)* — возвращает значение элемента в позиции p списка l.
- *delete(p, l)* — удаляет элемент в позиции p
- *next(p, l)* и *previous(p, l), makeNull(l), first(l), printList(l)* и т.п.

И ещё тысячи других, в зависимости от потребностей приложения. Но нужно помнить про принцип модульности и соблюдать баланс между богатством возможностей и простотой интерфейса.

Если есть такие операции, со списком можно работать непосредственно через них, не заботясь о том, как список представляется в памяти и как эти операции реализованы.

Например, программа удаления совпадающих элементов из списка:

```
void purge(List l)
{
    Position p = first(l);
    while (p != end(l))
    {
        q = next(p, l);
        while (q != end(l))
        {
            if (isEqual(retrieve(p, l), retrieve(q, l))
                delete(q, l);
            else
                q = next(q, l);
        }
        p = next(p, l);
    }
}
```

Заметьте, что очень легко писать неэффективные программы, забывая о трудоёмкости операций АТД, что просто сделать, так как трудоёмкость зависит от реализации, которая скрыта от пользователя. Поэтому обычно в документации к АТД указывается гарантированная трудоёмкость операций.

Реализация: самый простой способ — через массив. Мы на нём подробно останавливаться не будем, потому что он тупой и неэффективный. Список можно объявить так:

```
struct List
{
    ElementType a[n];
    int last;
};
```

Для вставки и удаления требуется сдвигать все элементы массива за вставляемым/удаляемым, зато предыдущий/следующий элемент находится без проблем. К несчастью, для хранения списка в массиве надо выделять память, которой

достаточно для хранения списка максимальной длины, так что это вообще не подходит, если максимальная длина неизвестна. makeNull() устанавливает last в -1, end возвращает last + 1.

Более пригодный способ — однонаправленные списки, реализованные с помощью указателей. Нужно завести структуру, описывающую тип ячейки — пару (значение, указатель на следующий элемент), например, так:

```
struct ListElement {
    ElementType value;
    ListElement *next;
};
```

Тогда сам тип списка будет каким-то таким:

```
struct List {
    ListElement *head;
};
```

(здесь должна быть картинка)

Тогда сам список будет представляться указателем на головную ячейку. Позиция в списке представляется указателем на ячейку, причём на какую именно — зависит от реализации, тут возможна масса вариантов. Чтобы операции вставки и удаления работали за константное время, обычно позиция указывает на предыдущую ячейку. Тогда для единообразия операций имеет смысл завести в списке фиктивную головную ячейку, в которой не будет храниться значение. end() будет возвращать указатель на ячейку, у которой next равен NULL, first() — указатель на головную ячейку.

(тут должны быть картинки с реализацией insert и delete)

next() будет выполняться за константное время, а вот previous() и end() — за линейное. При удалении элемента из списка нельзя забывать освобождать память, а при удалении всего списка — удалять память, выделенную под все элементы. Делается это обычно рекурсивно:

```
void removeAll(ListElement *element)
{
    if (element != NULL) {
        removeAll(element->next);
        delete element;
    }
}

makeNull(List &l)
{
    removeAll(l.next);
    l.next = NULL;
```

}

Бывают ещё двусвязные списки, которые хороши тем, что по ним можно одинаково эффективно ходить и в прямом, и в обратном направлении. В этом случае можно сделать позицией указатель не на предыдущую, а прямо на текущую ячейку, а end() сделать возвращающей null.

(здесь должны быть картинки, показывающие добавление и удаление)

С указателями можно делать сколь угодно хитрые структуры данных, например, циклические списки — когда хвост указывает на голову, а голова — на хвост.

(тут должны быть картинки и пояснения про то, как это может быть устроено)

Со списками связана одна из эффективных (работающих за O(n log n)) сортировок, которая может быть использована в качестве внешней — сортировка слиянием. Работает она так: пусть у нас есть два отсортированных списка. Построим результирующий отсортированный список так: если элемент первого списка меньше, чем элемент второго, добавим его в результирующий список и удалим из первого. Если второго больше, чем первого — наоборот. Продолжим так делать, пока один из списков не кончится, оставшиеся в другом списке элементы добавим в конец результирующего. Как отсортировать неотсортированный список: рекурсивно. Делим его на два списка примерно одинакового размера, сортируем слиянием каждый, а потом сливаем два отсортированных списка. Вырожденный случай сортировки слиянием — сортировка вставками, когда список делится не на две одинаковые части, а на часть из 1 элемента и всё остальное.

Сравнение операций над статическими массивами, динамическими массивами и связными списками:

	Linked list	**Array**	**Dynamic array**
Indexing	Θ(*n*)	Θ(1)	Θ(1)
Insertion/deletion at end	Θ(1)	N/A	Θ(1)
Insertion/deletion in middle	Θ(1)	N/A	Θ(*n*)
Wasted space (average)	Θ(*n*)	0	Θ(*n*)

Достоинства списков:

- лёгкость добавления и удаления элементов
- размер ограничен только объёмом памяти компьютера и разрядностью указателей
- динамическое добавление и удаление элементов

Недостатки:

- сложность определения адреса элемента по его индексу (номеру) в списке
- на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в массивах, например, указатели не нужны)

- работа со списком медленнее, чем с массивами, так как к любому элементу списка можно обратиться, только пройдя все предшествующие ему элементы
- элементы списка могут быть расположены в памяти разреженно, что окажет негативный эффект на кэширование процессора
- над связными списками гораздо труднее (хотя и в принципе возможно) производить параллельные векторные операции, такие как вычисление суммы

Очередь — это такой список, в который можно добавлять только в конец, а удалять только из начала. Применяется для организации последовательной обработки, например, очередь событий в ОС.

Типичные операции:

```
bool empty(Queue &q)
T& front(Queue &q)
void dequeue(Queue &q)
void enqueue(Queue &q, const T& value)
int size(Queue &q)
```

Реализуется циклическим массивом или поверх списка.

Ещё бывает дэк — очередь, операции над которой можно совершать и с начала, и с конца.

Стек (или магазин) — это такой список, добавлять в который или удалять из которого можно только в голову (называемую вершиной стека). Весьма активно используется, например, для организации стека вызовов. Есть стековые языки программирования, например, Forth, байт-коды Java и .Net.

Типовые операции:

```
bool empty(Stack &s)
SizeType size(Stack &s)
ValueType& top(Stack &s)
void push(Stack &s, const ValueType &value)
void pop(Stack &s)
```

Реализуется поверх массива или списка.