Работа с массивами (продолжение)

Перестановки

Преобразование массивов часто заключается в некоторой перестановке их элементов. Можно этот вопрос рассмотреть в общей постановке. Пусть дан массив p, содержащий все целые числа из диапазона $0, \ldots, n-1$, которые задают перестановку. Нужно применить эту перестановку для элементов массива a.

Задание перестановки можно понимать в двух вариантах:

1. Значение `p[i]` есть индекс, куда должен быть переставлен элемент `a[i]`.

2. Значение `p[i]` есть индекс элемента, который должен быть поставлен на место `i`.

По сути эти два варианта задают прямую или обратную перестановки.

Реализация таких перестановок не составляет труда (например, для массива целых чисел)

```
void PermDirect (int *a, int n, int *p)
{
    int i = 0, b, c;
    c = a[0];
    do {
        b = a[p[i]];
        a[p[i]] = c;
        i = p[i];
        c = b;
    } while (i != 0);
    a[i] = c;
}
void PermReverse (int *a, int n, int *p)
{
    int i = 0, b, c;
    c = a[0];
    do {
        a[i] = a[p[i]];
        i = p[i];
    } while (i != 0);
    a[i] = c
}
```

Однако такие реализации будут работать неправильно. Перестановка может раскладываться в набор циклов, и данные варианты обработывают только цикл, содержащий индекс 0. Как обработать другие циклы, какие индексы в них входят? Это зависит от конкретной перестановки. Для некоторых перестановок это можно определить заранее. Для других явное указание начала других циклов может оказаться непосильной задачей.

С практической точки зрения мы можем помечать пройденные индексы в массиве перестановки специальным значением, или, например, просто менять им знак. Тогда при завершении очередного цикла мы можем найти в мессиве p следующий положительный элемент, и начать с него новый цикл, и т.д. пока не исчерпаются все индексы.

Подправьте приведенный выше код так, чтобы он смог обрабатывать всю перестановку.

Иногда перестановку можно задавать явным образом по формулам. Пример рассматривался ранее — циклический сдвиг массива на k позиций. В этом последнем случае перестановка задается простой формулой `p[i] = (i + k)%n` . С циклами тут тоже все просто. Если они существуют, то будут начинаться последовательно в позициях $0, 1, 2, \ldots$. Таким образом, для выполнения подобной перестановки не нужен массив p, а можно просто записать значения `p[i]` по указанным формулам.

В других случаях такой фокус не проходит, и не смотря на формулы для задания перестановки, определение циклов явным образом может оказаться тоже непростым и даже невозможным делом.

В заключение еще одна задача про перестановки. Нужно последовательно перебрать все перестановки, т.е. для данной перестановки уметь "вычислять" следующую.

Идея решения. Перестановка рассматривается как "число", записанное в n-ричной системе счисления. Таким образом, элементы перестановки — это "цифры" такого числа. Для данного числа надо найти чледующее по порядку, но состояшее из тех же цифр. Надо переставить цифры так, чтобы число увеличилось, но на минимально возможное значение. Число увеличится, если бóльшая цифра будут переставлена с более старший разряд. Поэтому, если имеем убывающий набор чисел, то это "наибольшая" перестановка, поскольку сдвиг облого элемента влево ее только "уменьшит". Значит, надо заменять первый же элемент, который примыкает слева к убывающему фрагменту в конце массива, и для замены надо брать наименьший подходящий элемент (от должен быть больше заменяемого, чтобы "число увеличилось"). После перестановки надо обеспечить "минимальность" фрагмента после измененной "цифры". Для этого достаточно его просто упорядочить по возрастанию (так как это будет "миниальная" перестановка).

Например, пусть перестановка имеет вид 1,2,5,4,6,3. Упорядоченность нарушается на паре 4, 6. Обмениваем и получаем 1,2,5,6,4,3, но это не самы маленький вариант, упорядочиваем конец и получаем 1,2,5,6,3,4. Далее аналогично получим 1,2,5,6,4,3; 1,2,6,3,4,5 и т.д. Вот программа

```
void nextPermutation(int * p, int n) {
        int i,j,k;
        for (i = n - 1; i > 0; --i) { // ищем начало убывающего участка с конца
            if (nums[i-1] < nums[i]) break;
        }
        --i;  // позиция перед участком
        if (i == -1) {  // весь массив убывающий - это максимальная перестановка
            sort(a, n);   // переходим к минимальной перестановке
            return;
        }
        for (j = n - 1; j > i; --j) { // ищем число для обмена
            if (a[j] > a[i]) break;
        }
        // обмениваем
        k = nums[i]; nums[i] = nums[j]; nums[j] = k;
        // упорядочиваем конец массива
        sort(a+i+1, n-i-1);
    }
```

Здесь использовалось обозначение для функции сортировки мессива по возрастанию, которую нам еще предстоит написать.

Набор из всевозможных перестановок может пригодиться при тестировании различных алгоритмов.

Отладка и организация работы.

Примитивные приемы отладки

Программа редко начинает работать правильно сразу, практически никогда такого не случается. После того, как исправлены все синтаксические ошибки и компиляция прошла успешно, программу запускают на выполнение, и тут возможно несколько различных исходов. Перечислим их в порядке “тяжести” или ”очевидности“.

1. Программа не завершается (“повисла”).

2. Выполнение программы было остановлено операционной системой по причине какого-то фатального системного отказа.

3. Программа завершилось, но результата не выдала или выдала “странный” результат.

4. Программа нормально завершилась и выдала какой-то результат.

Все подобные ситуации требуют дополнительных действий по проверке правильности работы вашей программы, обнаружению ошибок и их устранению. Даже п.4 еще не свидетельствует о правильности, так как выданный “нормальный” результат может не соответствовать постановке задачи.

Для первичной и простейшей отладки можно применить технику явной трассировки, т.е. заставить программу отчитываться о том, что с ней происходит по ходу выполнения. Простейший вариант — вставить по ходу выполнения программы печать различных сообщений или значений переменных из текущего контекста с тем, чтобы понять до какого места дошло выполнение вашей программы, а какие участки программы вдруг оказались не задействованы.

```
...
printf("call fun with %d %d %d\n", a, b, c);
x = fun(a, b, c);
printf("fun:  %d\n", x);

printf("if:  %d %d\n", b, c);
if (b < c) {
    .....
}

for (i = 0; i < x; i++) {
    printf("loop  i  %d\n", i);
    .....
}
printf("end loop\n");
...
```

По ходу выполнения подобной программы мы увидим с какими значениями параметров вызывается функция, что получено в результате вызова, какие значения были перед проверкой условия, для каких i выполняется цикл и т.п. Конечно тут нужна разумная достаточность, так как большое количество данных при выводе будет непросто анализировать, поэтому такие печати надо вставлять по мере необходимости и потом отключать, например, комментируя соответствующие строчки.

В простейшем варианте может оказаться достаточно просто печатать сообщения о том, что выполнение вашей программы дошло до конкретного места в коде.

```
.... some code A
```

```
printf("A:\n");
.... some code B
printf("B:\n");
.... some code C
printf("C:\n");
.... some code D
printf("D:\n");
```

Увидев, что на выходе появились метки A: и C:, а D: уже не появилось, можно предположить, что фрагмент some code C работает некорректно и провести более подробную трассировку на это участке.

ВАЖНО!

Любую отладочную печать надо заканчивать символом перевода строки `\n`

Напоминаем, ввод/вывод в системе буферизуется, т.е. функции вывода (printf и подобные) только переписывают выводимые символы в специальный участок памяти (буфер) а непосредственный вывод этих символов на устройство (монитор, диск) выполняется операционной системой в тот момент, когда она сочтет, что можно этим заняться без ущерба выполняющимся в данный момент программам. Поэтому, если вдруг ваша программа вывела что-то, а потом "упала", то может быть, что к моменту падения система еще не вывела ваше сообщение, и оно не появилось на экране. Тогда вы будете ошибочно считать, что функция вывода еще не отработала в вашей программе и отказ произощел до ее вызова, хотя на самом деле это совсем не так. Вывод перевода строки `\n` на экран монитора принуждает систему вывести имеющийся буфер именно в этот момент. Таким образом, вы можете быть уверены, что программа доработала до этого места.

Трассировка часто помогает решить проблемы с пунктами 1 и 2.

Если программа выдается странные результаты, то первым делом надо проверить правильность ввода и вывода данных. Распечатайте данные сразу после того как они введены. Ошибки в форматах (типах переменных) при вводе будут с большой вероятностью обнаружены.

Иногда полезно распечатывать возвращаемое значение непосредственно из функции перед оператором `return` и там, где эта функция вызывается. Это позволяет отлавливать ошибки несоответствия типов переменных внутри функции и при приеме ее результата.

Типичными ошибками являются отказы системы по причине segmentation fault и arithmetic (или floating point) exception.

Первая ошибка формально вызвана вашей попыткой обратиться к закрытому от вас или неопределенному участку памяти.

```
int x[10];
char * p = nullptr;


....
y = x[100];  // возможно  segmentation fault
c = *p;      // segmentation fault
```

Типичные причины — выход индекса за границы массива или разименовывание некорректного указателя (например, нулевого). Помогает печать индексов или указателей (адресов) в подозрительных местах вашего кода.

Arithmetic exception — попытка выполнения некорректной арифметической операции — деление на 0, корень или логарифм отрицательного числа и т.п, а также исползование неициализированной переменной.

```
z = 2 + x[1];  // возможно arithmetic exception
z = 0;
w = 2 / z;   // arithmetic  exception
```

ВАЖНО!

Обнаружение системой подобных ошибок зависит от настроек этой системы. Например, Windows системы, как правило игнорируют подобные ошибки и просто продолжают выполнение вашей программы. Например, при делении на 0 результат полагается равным Inf, при использовании неинициализированной переменной результат может оказаться NaN, что далее будет как-то влиять на окончательный итог. Кроме того, свободная память также по-разному предоставляется пользовательской программе. Она может чиститься и запоняться нулями или NaN, может не чиститься и содержать непредсказуемые старые значения. Самый плохой итог — когда неинициализированные переменнае и настройки системы позволят вашей программе вычислить правдоподобный, но неверный ответ. Поэтому, пункт 4 — это еще не вывод о полной корректности вашей программы.

Наиболее неприятный случай — это когда программа вычисляет и печатает (вроде бы) правильный ответ, но после этого завершается с системной ошибкой. Это скорее всего означает, что с алгоритмом “все в порядке”, но есть ошибки в использовании памяти, и система, завершая вашу программу, обнаружила, что вы где-то некорректно эту память использовали. То есть, обращение по некорректному адресу не вызвало краха сразу (segmentation fault) так как вы обратились формально к разрешенной вам памяти, но эта память была использована системой для своих нужд, и вы ее испортили. Главная проблема здесь в том, то реакция системы на вашу ошибку происходит гораздо позже, чем эта ошибка сделана. Возможный варант отладки в этом случае — временное “отключение” отдельных часте вашего кода (например, комментированием их) и проверка сохраняется или исчезает при этом ошибка. Конечно, исходный алгоритм уже не будет сохранен, но зато так можно примерно обнаружить участок, вызывающий ощибку. Если программа правильно разбита на отдельные функции, то такого рода поиск выполняется достаточно легко комментированием вызовов отдельных функции либо их тела целиком.

Отладчики

Любая система программирования предоставляет пользователю специальные средства и инструменты для отладки программ. Подобные отладчики весьма разнообразны по функциям и возможностям, и их следует осваивать для тех систем, с которыми вы работаете. Однако все отладчики имеют общие принципы.

— для работы с отладчиком программа должна быть подготовлена (откомпилирована и собрана) в режиме отладки (debug)

— в коде программы можно установить точки останова (breakpoint)

— после остановки выполнения программы в точке останова, вы можете выполнять программу “по шагам”, обычно строка кода за строкой.

— если в строке вашего кода стоит вызов функции, вы можете либо выполнить этот вызов “целиком”, либо “провалиться” внутрь функции и продолжить пошаговое выполнение программы уже внутри этой функции.

— вы можете запустить программу на выполнение дальше до следующей точки останова.

— при останове вы можете распечатать значения переменных из текущей области видимости.

— вы можете посмотреть текущий стек вызовов функций, т.е. какие функции были последовательно вызваны при проходе до вашей текущей позиции в коде и с какими

параметрами. Можно перемещаться по стеку вызовов вверх и потом обратно вниз, переходя на строчки с вызовами соответствующих функций.

Таким образом, вы можете детально проследить как именно выполняется ваша программа и какие значения переменных она в эти моменты получает и обрабатывает.

Кроме этого, если при выполнении вашей программы происходит системный отказ, то отладчик останавливается на строке вашей программы, вызвавшей этот отказ, и далее можно смотреть стек вызовов и текущие значение переменных.

Отладчик GDB (GNU Debugger)

В unix системах в рамках проекта GNU широко применяется отладчик gdb. Это консольное приложение и поэтому не всегда удобен для сложной отладки, но в простейших случаях учебных программ, он помогает быстро найти многие ошибки.

Для отладки программы с помощью gdb, ее надо скомпилировать с параметром `-g` и желательно без оптимизации `-O0` и далее запустить из под отладчика

```
gcc myprog.c myfun.c -o prog -g -O0
gdb ./prog
```

Отладчик напечатает заголовок на пол-экрана и далее будет ждать от вас команд с приглашением (gdb).

Команд довольно много, но нам потребуется всего несколько. Многие команды можно сокращать до 1-2 первых символов.

help — подсказка по командам, все понятно, если знаете чего хотите;
quit, q — выйти из отладчика;
run, r — запустить программу на выполнение;
backtrace, bt — показать стек вызовов при останове
up — подняться по стеку вызовов
down — спуститься по стеку вызовов
nexn, n — выполнить следующую строку (которая отображается на экране);
step, s — выполнить следующую строку с входом внутрь вызываемой функции;
next k, step k — выполнить сразу k строк кода
until k — выполнить до указанной строчки
finish — продолжить выполнение до выхода из функции
print, p — напечатать значения переменных
p a, p s[21] ... — напечатать значение переменной a, элемента массива p s[21] и т.п.
list — печать текущей части листинга программы
list 20, 35 — печать диапазона строк
list main.c:f1 — печать листинга функции f1() из файла main.c
continue — продолжить выполнение после останова
set variable a = 1234 — установить новое значение указанной переменной
break, b — установить точку останова
b file:function — b myprog.c:main — на функцию main в файле myprog.c
b file:line — b myfun.c:123 — на 123 строку в файле myfun.c
break ... if ... — условный останов
break 120 if i == 12 — на 120 строке при условии что i == 12
watch a — останов на месте изменения переменной a
rwatch a — останов на месте чтения переменной a
awatch a — одновременно считывание/изменение
info break — листинг имеющихся точек останова
del k — удаление точки останова по номеру
d — удаление всех точек останова

disable k — временное отключение точки останова

enable k — включение точки останова

Для управления точками останова предусмотрено еще множество команд, который можно увидеть по команде help break или help breakpoints.

В типичном варианте возникновения отказов типа segmentation fault или arithmetic exception сценарий отладки может выглядеть примерно так:

— компилируем с -g

— запускаем через gdb

— получаем останов на некоторой строке и анализируем эту строку

например, строка

```
a[i] = a[i+2] - b[j-1];
```

Печатаем значения индексов и обнаруживаем, что они выходят за границы массива. Либо, если индексы правильные, печатаем значения элементов массива с этими индексами и обнаруживаем, что они не инициализированы.

Может оказаться, что строка останова не ваша, а принадледит программам из системных библиотек. Например, так часто бывает, если при работе с файлом он не открылся, и вы передали нулевой указатель в функцию ввода или вывода. В этом случае надо посмотреть стек вызовов командой bt и подняться вверх (up) до своих функций. В упомянутом выше случае вы увилите свою функцию типа fscanf и, напечатав указатель на файл, увидите, что он есть 0. Аналогично действуем и в других случаях.

При отладке других ошибок ставим точки останова и выполняем части кода по шагамЮ отслеживая что и как делает ваш алгоритм.

Процедура make

При большом количестве файлов в проекте приходится их часто исправлять, перекомпилировать и собирать готовую программу. Этот процесс носит технический характер и в больших проектах требует указатия многих дополнительных параметров. Кроме того, в больших проектах компиляция всех исходных файлов может занимать достаточно много времени (сборка обычно происходит быстро). Однако понятно, что нет необходимости перекомпилировать файл, если внесенные изменения его не затронули. Для ускорения и автоматизации этого процесса в интергрированных системых программирования вволится понятие проекта с описанием всех опций, а в unix системах традиционно используется аналог — процедура make, которая управляет подготовкой программы при помощи описанийЮ указанных в специальном файле makefile.

Суть makefile — указать какие части вашего проекта от чего зависят и какие действия нужно предпринять, чтобы переделать эту часть при внесении изменений в файлы, от которых эта часть зависит. Содержимое makefile состоит из возможных определений разнообразных сокращений для длинных строк и из пар строк, первая из которых задает цель и ее дависимости, а вторая определяет как получить цель из указанных зависимостей.

Здесь рассмотрим простейшие способы записи makefile.

Пусть в проекте участвуют файлы `main.c, fun.c, fun.h, def.h`, составляющие нашу программу, причем файл `def.h` подключается только в файле `main.c` а файл `fun.h` подключается в оба с-файла.

Создадим следующий makefile:

```
prog: main.o fun.o
        gcc main.o fun.o -o prog
```

```
main.o: main.c fun.h def.h
        gcc -c main.c -Wall -Wextra

fun.o: fun.c fun.h
        gcc -c fun.c -Wall -Wextra

clean:
        rm -f prog main.o fun.o
```

ВАЖНО! строка с правилом должна начинаться с символа табуляции (в не нескольких пробелов)!!!

В правиле компиляции как пример указаны опции компиляции -Wall -Wextra, можно было сократить запись, введя для них отдельное обозначение

```
CFLAGS = -c -Wall -Wextra

prog: main.o fun.o
        gcc main.o fun.o -o prog

main.o: main.c fun.h def.h
        gcc $(CFLAGS) main.c

fun.o: fun.c fun.h
        gcc  $(CFLAGS) fun.c

clean:
        rm -f prog *.o
```

Есть средства автоматически указывать группы файлов и применять правила к этим группам, но это пока нам не надо.

По умолчанию, процедура make использует makefile для работы, но можно также указать ключ -f для использования другого файла.

```
make                        - используется makefile
make -f othermakefile       - используется othermakefile
```

По умолчанию реализуется первая цель, указанная в файле. Для реализации любой другой цели надо ее указать при вызове, например

```
make clean
```

Алгоритм выбора целей и операций основан на анализе времени изменения указанных файлов. Если время цели раньше, чем время ее зависимостей (т.е. зависимости были изменены уже после создания цели), то цель надо переделать (выполнить указанную команду). Алгоритм проверяет все зависимости и выстраивает команты в том порядке, который обеспечивает корректный последовательный пересчет всех целей, причем только тех, которые действительно нужны.