**6. Структури**

*Структура -* це набір з однієї або більше змінних, можливо різних типів, згрупованих під одним ім'ям для зручності обробки. (В деяких мовах, найвідоміша з яких паскаль, структури називаються "записами").

Традиційним прикладом структури є облікова картка працює: "службовець" описується набором атрибутів таких, як прізвище, ім'я, по батькові (прізвище та ініціали), адреса, код соціального забезпечення, зарплата і т.д. Деякі з цих атрибутів самі можуть опинитися структурами: П.І.Б. Має кілька компонент, як і адреса, і навіть зарплата.

Структури виявляються корисними при організації складних даних особливо в великих програмах, оскільки в багатьох ситуаціях вони дозволяють згрупувати пов'язані дані таким чином, що до них можна звертатися, як до одного цілого, а не як з окремими об'єктами. У цій лекції ми постараємося продемонструвати те, як використовуються структури. Програми, які ми для цього будемо використовувати, більше, ніж багато інших в цій книзі, але все ж досить помірних розмірів.

**6.1. Основні відомості**

Давайте знову звернемося до процедур перетворення дати з "лекції №5". Дата складається з кількох частин таких, як день, місяць, і рік, і, можливо, день року і ім'я місяці. Ці п'ять змінних можна об'єднати в одну структуру виду:

struct date {

int day;

int month;

int year;

int yearday;

char mon\_name[4];

};

опис структури, що складається з укладеного у фігурні дужки списку описів, починається з ключового слова struct. За словом struct може слідувати необов'язкове ім'я, зване ярликом структури (тут це date). Такий ярлик іменує структури цього виду і може використовуватися в подальшому як скорочений запис докладного опису.

Елементи або змінні, згадані в структурі, називаються членами. Ярлики і члени структур можуть мати такі ж імена, що і звичайні змінні (тобто не є членами структур), оскільки їх імена завжди можна розрізнити за контекстом. Звичайно, зазвичай однакові імена привласнюють тільки тісно зв'язаних об'єктів.

Точно так же, як в разі будь-якого іншого базисного типу, за правою фігурною дужкою, що закриває список членів, може слідувати список змінних. *Оператор*

struct { ... } x,y,z;

синтаксично аналогічний

int x,y,z;

в тому сенсі, що кожен з операторів описує x, y і z в якості змінних відповідних типів і призводить до виділення для них пам'яті.

Опис структури, за яким не слід списку змінних, не приводить до виділення будь-якої пам'яті; воно тільки визначає шаблон або форму структури. Однак, якщо такий опис забезпечено ярликом, то цей ярлик може бути використаний пізніше при визначенні фактичних екземплярів структур. Наприклад, якщо дано наведене вище описdate, то

struct date d;

визначає змінну d як структури типу date. Зовнішню або статичну структуру можна ініціалізувати, помістивши слідом за її визначенням список ініціалізаторів для її компонент:

struct date d={ 4, 7, 1776, 186, "jul"};

Член певної структури може бути вказаний в вираженні за допомогою конструкції виду

имя структуры . Член

--------------------

Операція вказівки члена структури "." пов'язує ім'я структури і ім'я члена. Як приклад визначимо leap (ознака високосного року) на основі дати, що знаходиться в структурі d,

leap = d.year % 4 == 0 && d.year % 100 != 0

|| d.year % 400 == 0;

або перевіримо ім'я місяців

if (strcmp(d.mon\_name, "aug") == 0) ...

Або перетворимо перший символ імені місяця так, щоб воно починалося з малої літери

d.mon\_name[0] = lower(d.mon\_name[0]);

структури можуть бути вкладеними; облікова картка службовця може фактично виглядати так:

struct person {

char name[namesize];

char address[adrsize];

long zipcode; /\* почтовый индекс \*/

long ss\_number; /\* код соц. Обеспечения \*/

double salary; /\* зарплата \*/

struct date birthdate; /\* дата рождения \*/

struct date hiredate; /\* дата поступления

на работу \*/

};

структура person містить дві структури типу date. Якщо ми визначимо emp як

struct person emp;

то

emp.birthdate.month

буде посилатися на місяць народження. Операція вказівки члена структури "." асоціюється зліва направо.

**6.2. Структури і функції**

У мові "C" існує ряд обмежень на використання структур. Обов'язкові правила полягають в тому, що єдині операції, які ви можете проводити зі структурами, полягають у визначенні її адреси за допомогою операції & і доступі до одного з її членів. Це тягне за собою те, що структури не можна присвоювати або копіювати як ціле, і що вони не можуть бути передані функціям або повернуті ними.

(В наступних версіях ці обмеження будуть зняті). На покажчики структур ці обмеження однак не накладаються, так що структури і функції все ж можуть із зручністю працювати спільно. І нарешті, автоматичні структури, як і автоматичні масиви, не можуть бути ініційовані; ініціалізація можлива тільки в разі зовнішніх або статичних структур.

Давайте розберемо деякі з цих питань, переписавши з цією метою функції перетворення дати з попередньої лекції так, щоб вони використовували структури. Так як правила забороняють безпосередню передачу структури функції, то ми повинні або передавати окремо компоненти, або передати покажчик всієї структури. Перша можливість демонструється на прикладі функції day\_of\_year, як ми її написали в "лекції №5" :

d.yearday = day\_of\_year(d.year, d.month, d.day);

інший спосіб полягає в передачі покажчика. Якщо ми опишемо hiredate як

struct date hiredate;

і перепишемо day\_of\_year потрібним чином, ми зможемо тоді написаи

hiredate.yearday = day\_of\_year(&hiredate);

передаючи покажчик на hiredate функції day\_of\_year. Функція повинна бути модифікована, тому що її аргумент тепер є покажчиком, а не списком змінних.

day\_of\_year(pd) /\* set day of year from month, day \*/

struct date \*pd;

{

int i, day, leap;

day = pd->day;

leap = pd->year % 4 == 0 && pd->year % 100 != 0

|| pd->year % 400 == 0;

for (i =1; i < pd->month; i++)

day += day\_tab[leap][i];

return(day);

}

*опис*

struct date \*pd;

каже, що pd є покажчиком структури типу date. Запис, показана на прикладі

pd->year

є новим. Якщо p - покажчик на структуру, то

p-> член структури

------------------

звертається до конкретного члена. (Операція -> - це знак мінус, за яким слідує знак ">".)

Так як pd вказує на структуру, то до члена year можна звернутися і в такий спосіб

(\*pd).year

але покажчики структур використовуються настільки часто, що запис -> виявляється зручним скороченням. круглі дужки в (\* pd) .year необхідні, тому що операція вказівки члена структури старше, ніж \*. Обидві операції, "->" і ".", асоціюються зліва направо, так що конструкції зліва і справа еквівалентні

p->q->memb (p->q)->memb

emp.birthdate.month (emp.birthdate).month

Для повноти нижче наводиться інша функція, month\_day, переписана з використанням *структур*.

month\_day(pd) /\* set month and day from day of year \*/

struct date \*pd;

{

int i, leap;

leap = pd->year % 4 == 0 && pd->year % 100 != 0

|| pd->year % 400 == 0;

pd->day = pd->yearday;

for (i = 1; pd->day > day\_tab[leap][i]; i++)

pd->day -= day\_tab[leap][i];

pd->month = i;

}

Операції роботи зі структурами "->" і "." поряд зі () для списку аргументів і [] для індексів перебувають на самому верху ієрархії старшинства операцій і, отже, зв'язуються дуже міцно. Якщо, наприклад, є опис

struct {

int x;

int \*y;

} \*p;

то вираз

++p->x

збільшує x, а не p, так як воно еквівалентно висловом ++ (p-> х). Для зміни порядку виконання операцій можна використовувати круглі дужки: (++ p) -> x збільшує p до доступу до x, а (p ++) -> x збільшує p після. (Круглі дужки в останньому випадку не є обов'язковими. Чому?)

Цілком аналогічно \* p-> y витягує те, на що вказує y; \* P-> y ++ збільшує y після обробки того, на що він вказує (точно так само, як і \* s ++); (\* P-> y) ++ збільшує те, на що вказує y; \* P ++ -> y збільшує p після вибірки того, на що вказує y.

6.3. Масиви структур

структури особливо підходять для управління масивами пов'язаних змінних. Розглянемо, наприклад, програму підрахунку числа входжень кожного ключового слова мови "C". Нам потрібен масив символьних рядків для зберігання імен та масив цілих для підрахунку. одна з можливостей полягає в використанні двох паралельних масивів keyword і keycount:

char \*keyword [nkeys]; int keycount [nkeys];

Але сам факт, що масиви паралельні, вказує на можливість іншої організації. Кожне ключове слово тут по суті є парою:

char \*keyword; int keycount;

і, отже, є масив пар. Опис структури

struct key {

char \*keyword;

int keycount;

} keytab [nkeys];

визначає масив keytab структур такого типу і відводить для них пам'ять. Кожен елемент масиву є структурою. Це можна було б записати і так:

struct key {

char \*keyword;

int keycount;

};

struct key keytab [nkeys];

Так як структура keytab фактично містить постійний набір імен, то найлегше форматувати її один раз і для всіх членів при визначенні. Ініціалізація структур цілком аналогічна попереднім ініціалізаціям - за визначенням слідує укладений у фігурні дужки список ініціалізаторів:

struct key {

char \*keyword;

int keycount;

} keytab[] = {

"break", 0,

"case", 0,

"char", 0,

"continue", 0,

"default", 0,

/\* ... \*/

"unsigned", 0,

"while", 0

};

Ініціалізатори перераховуються парами відповідно членам структури. Було б більш точно укладати в фігурні дужки ініціалізатор для кожної "рядки" або структури в такий спосіб:

{ "break", 0 },

{ "case", 0 },

. . .

Але коли ініціалізатори є простими змінними або символьними рядками і всі вони присутні, то у внутрішніх фігурних дужках немає необхідності. Як завжди, компілятор сам вирахує число елементів масиву keytab, якщо ініціалізатор присутні, а дужки [] залишені порожніми.

Програма підрахунку ключових слів починається з визначення масиву keytab. Провідна програма читає свій файл введення, послідовно звертаючись до функції getword, яка витягує з введення по одному слову за звернення. Кожне слово шукається в масиві keytab за допомогою варіанту функції бінарного пошуку, написаної нами в "лекції №3". (Звичайно, щоб ця функція працювала, список ключових слів повинен бути розташований в порядку зростання).

#define maxword 20

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

main() /\* count "c" keywords \*/

{

int n, t;

char word[maxword];

while ((t = getword(word,maxword)) != EOF)

if (t == letter)

if((n = binary(word,keytab,nkeys)) >= 0)

keytab[n].keycount++;

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

if (keytab[n].keycount > 0)

printf("%4d %s\n",

keytab[n].keycount, keytab[n].keyword);

}

binary(word, tab, n) /\* find word in tab[0]...tab[n-1] \*/

char \*word;

struct key tab[];

int n;

{

int low, high, mid, cond;

low = 0;

high = n - 1;

while (low <= high) {

mid = (low+high) / 2;

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

high = mid - 1;

else if (cond > 0)

low = mid + 1;

else

return (mid);

}

return(-1);

}

Ми незабаром наведемо функцію getword; поки досить сказати, що вона повертає letter кожен раз, як вона знаходить слово, і копіює це слово в свій перший аргумент.

Величина nkeys - це кількість ключових слів в масиві keytab. Хоча ми можемо порахувати це число вручну, набагато легше і надійніше доручити це машині, особливо в тому випадку, якщо список ключових слів зазнає змін. Однією з можливостей було б закінчити список ініціалізаторів зазначенням на нуль і потім пройти у циклі крізь масив keytab, поки не знайдеться кінець.

Але, оскільки розмір цього масиву повністю визначений до моменту компіляції, тут є більш проста можливість. Число елементів просто є

size of keytab / size of struct key

справа в тому, що в мові "C" передбачена унарна операція sizeof, яка виконується під час компіляції, яка дозволяє обчислити розмір будь-якого об'єкта.

Вираз

sizeof(object)

видає ціле, рівне розміру зазначеного об'єкта. (Розмір визначається в неспеціфіцированних одиницях, званих "байтами", які мають той же розмір, що і змінні типу char). Об'єкт може бути фактичною змінною, масивом і структурою, або ім'ям основного типу, як int або double, або іменем похідного типу, як структура. У нашому випадку число ключових слів дорівнює розміру масиву, поділеній на розмір одного елемента масиву. Це обчислення використовується в утвердженні #define для встановлення значення nkeys:

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

Тепер перейдемо до функції getword. Ми фактично написали більш загальний варіант функції getword, ніж необхідно для цієї програми, але він не на багато складніший. Функція getword повертає наступне "слово" з введення, де словом вважається або рядок букв і цифр, що починаються з літери, або окремий символ. тип об'єкту повертається в якості значення функції; це - letter, якщо знайдено слово, EOF для кінця файлу і сам символ, якщо він не буквений.

getword(w, lim) /\* get next word from input \*/

char \*w;

int lim;

{

int c, t;

if (type(c=\*w++=getch()) !=letter) {

\*w='\0';

return(c);

}

while (--lim > 0) {

t = type(c = \*w++ = getch());

if (t ! = letter && t ! = digit) {

ungetch(c);

break;

}

}

\*(w-1) = '\0';

return(letter);

}

Функція getword використовує функції getch і ungetch, які ми написали в "лекції №4": коли набір алфавітних символів переривається, функція getword отримує один зайвий символ. В результаті виклику ungetch цей символ поміщається назад у введення для наступного звернення.

Функція getword звертається до функції type для визначення типу кожного окремого символу з файлу введення. Ось варіант, справедливий тільки для алфавіту ASCII.

type(c) /\* return type of ascii character \*/

int c;

{

if (c>= 'a' && c<= 'z' || c>= 'A' && c<= 'Z')

return(letter);

else if (c>= '0' && c<= '9')

return(digit);

else

return(c);

}

Символічні константи letter і digit можуть мати будь-які значення, аби вони не вступали в конфлікт з символами, відмінними від букв і цифр, і з EOF; очевидно можливий наступний вибір

#define letter 'a'

#define digit '0'

функція getword могла б працювати швидше, якби звернення до функції type були замінені зверненнями до відповідного масиву type []. У стандартній бібліотеці мови "C" передбачені макроси isalpha і isdigit, діючі необхідним чином.

**Вправа 6-1**

Зробіть таку модифікацію функції getword та оцініть, як зміниться швидкість роботи програми.

**Вправа 6-2**

Напишіть варіант функції type, що не залежить від конкретного набору символів.

**Вправа 6-3**

Напишіть варіант програми підрахунку ключових слів, який би не брав до уваги появи цих слів у в'язнів в лапки рядках.

**6.4. Покажчики на структури**

Щоб проілюструвати деякі міркування, пов'язані з використанням покажчиків і масивів структур, давайте знову складемо програму підрахунку ключових рядків, використовуючи на цей раз покажчики, а не індекси масивів.

Зовнішнє опис масиву keytab не потрібно змінювати, але функції main і binary вимагають модифікації.

main() /\* count c keyword; pointer version \*/

{

int t;

char word[maxword];

struct key \*binary(), \*p;

while ((t = getword(word, maxword;) !=EOF)

if (t==letter)

if ((p=binary(word,keytab,nkeys)) !=null)

p->keycount++;

for (p=keytab; p>keytab + nkeys; p++)

if (p->keycount > 0)

printf("%4d %s/n", p->keycount, p->keyword);

}

struct key \*binary(word, tab, n) /\* find word \*/

char \*word /\* in tab[0]...tab[n-1] \*/

struct key tab [];

int n;

{

int cond;

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

struct key \*high = &tab[n-1];

struct key \*mid;

while (low <= high) {

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

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

high = mid - 1;

else if (cond > 0)

low = mid + 1;

else

return(mid);

}

return(null);

}

Тут є кілька моментів, які варто відзначити. По-перше, опис функції binary має вказувати, що вона повертає покажчик на структуру типу key, а не на ціле; це оголошується як у функції main, так і в binary. Якщо функція binary знаходить слово, то вона повертає покажчик на нього; якщо ж ні, вона повертає null.

По-друге, всі звернення до елементів масиву keytab здійснюються через покажчики. Це тягне за собою одна істотна зміна в функції binary: середній елемент більше не можна обчислювати просто по формулі

mid = (low + high) / 2

тому що складання двох покажчиків не дає якогось корисного результату (навіть після поділу на 2) і в дійсності є незаконним. Цю формулу треба замінити на

mid = low + (high-low) / 2

в результаті якої mid стає покажчиком на елемент, розташований посередині між low і high.

Вам також слід розібратися в ініціалізації low і high. Покажчик можна форматувати адресою раніше визначеного об'єкта; саме як ми тут і надійшли.

У функції main ми написали

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

Якщо p є покажчиком структури, то будь-яка арифметика з p враховує фактичний розмір цієї структури, так що p ++ збільшує p на потрібну величину, в результаті чого p вказує на наступний елемент масиву структур. Але не вважайте, що розмір структури дорівнює сумі розмірів її членів, - через вимоги вирівнювання для різних об'єктів в структурі можуть виникати "дірки".

І, нарешті, дещо другорядне питання про форму запису програми. Якщо повертається функцією величина має тип, як, наприклад, в

struct key \*binary(word, tab, n)

Tо може виявитися, що ім'я функції важко виділити серед тексту. У зв'язку з цим іноді використовується інший стиль запису:

struct key \*

binary(word, tab, n)

Це головним чином як кому подобається; виберіть ту форму, яка вам подобається, і дотримуйтеся її.

6.5. Структури, що посилаються на себе

Припустимо, що нам треба впоратися з більш загальним завданням, що складається в підрахунку числа появ всіх слів в деякому файлі введення. Так як список слів заздалегідь не відомий, ми не можемо їх впорядкувати зручним чином і використовувати бінарний пошук. Ми навіть не можемо здійснювати послідовний перегляд під час вступу кожного слова, з тим щоб встановити, чи не зустрічалося його раніше; така програма буде працювати вічно. (Більш точно, очікуваний час роботи зростає як квадрат числа вводяться слів). Як же нам організувати програму, щоб впоратися зі списком довільних слів?

Одне з рішень полягає в тому, щоб весь час зберігати масив надходячих до сих пір слів в упорядкованому вигляді, поміщаючи кожне слово в потрібне місце в міру їх надходження. Oднако це не слід робити, переміщаючи слова в лінійному масиві, - це також вимагатиме занадто багато часу. Замість цього ми використовуємо структуру даних, звану двійковим деревом.

Кожному новому слову відповідає один "вузол" дерева; кожен вузол містить:

покажчик тексту слова

----------------------

лічильник числа появ

-----------------------

покажчик вузла лівого нащадка

-----------------------------

покажчик вузла правого нащадка

------------------------------

Ніякої вузол не може мати більше двох нащадків; можливо відсутність нащадків або наявність тільки одного нащадка.

Вузли створюються таким чином, що ліве піддерево кожного вузла містить тільки ті слова, які менше слова в цьому вузлі, а праве піддерево тільки ті слова, які більше. Щоб визначити, чи знаходиться нове слово вже в дереві, починають з кореня і порівнюють нове слово зі словом, що зберігаються в цьому вузлі. Якщо слова збігаються, то питання вирішується ствердно. Якщо нове слово менше слова в дереві, то переходять до розгляду лівого нащадка; в іншому випадку досліджується правий нащадок. Якщо в потрібному напрямку нащадок відсутній, то значить нове слово не перебуває у дереві і місце цього відсутнього нащадка якраз і є місцем, куди слід ввести нове слово. Оскільки пошук з будь-якого вузла призводить до пошуку одного з його нащадків, то сам процес пошуку по суті є рекурсивним. Відповідно до цього найприродніше використовувати рекурсивні процедури введення і виведення.

Повертаючись назад до опису вузла, ясно, що це буде структура з чотирма компонентами:

struct tnode { /\* the basic node \*/

char \*word; /\* points to the text \*/

int count; /\* number of occurrences \*/

struct tnode \*left; /\* left child \*/

struct tnode \*right; /\* right child \*/

};

Цей "рекурсивний" опис вузла може здатися ризикованим, але насправді він цілком коректний. Структура не має права містити посилання на саму себе, але

struct tnode \*left;

описує left як покажчик на вузол, а не як сам вузол.

Текст самої програми виявляється на диво маленьким, якщо, звичайно, мати в розпорядженні набір написаних нами раніше процедур, що забезпечують потрібні дії. Ми маємо на увазі функцію getword для вилучення кожного слова з файлу введення і функцію alloc для виділення місця для зберігання слів.

Провідна програма просто зчитує слова за допомогою функції getword і поміщає їх в дерево, використовуючи функцію tree.

#define maxword 20

main() /\* word freguency count \*/

{

struct tnode \*root, \*tree();

char word[maxword];

int t;

root = null;

while ((t = getword(word, maxword)) != EOF)

if (t == letter)

root = tree(root, word);

treeprint(root);

}

функція tree сама по собі проста. Слово передається функцією main до верхнього рівня (корені) дерева. На кожному етапі це слово порівнюється зі словом, вже зберігаються в цьому вузлі, і за допомогою рекурсивного звернення до tree просочується вниз або до лівого, або до правого піддерева. Зрештою це слово або збігається з якимось словом, що вже знаходяться в дереві (в цьому випадку лічильник збільшується на одиницю), або програма натрапить на нульовий покажчик, який свідчить про необхідність створення і додавання до дерева нового вузла. У разі створення нового вузла функція tree повертає покажчик цього вузла, який поміщається в батьківський вузол.

struct tnode \*tree(p, w)

/\* install w at or below p \*/

struct tnode \*p;

char \*w;

{

struct tnode \*talloc();

char \*strsave();

int cond;

if (p == null) { /\* a new word

has arrived \*/

p == talloc(); /\* make a new node \*/

p->word = strsave(w);

p->count = 1;

p->left = p->right = null;

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

p->count++; /\* repeated word \*/

else if (cond < 0)/\* lower goes into left subtree \*/

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

else /\* greater into right subtree \*/

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

return(p);

}

Пам'ять для нового вузла виділяється функцією talloc, що є адаптацією для даного випадку функції alloc, написаної нами раніше. Вона повертає покажчик вільного простору, придатного для зберігання нового вузла дерева. (Ми незабаром обговоримо це докладніше). Нове слово копіюється функцією strsave в приховане місце, лічильник ініціалізується одиницею, і покажчики обох нащадків покладаються рівними нулю. Ця частина програми виконується тільки при додаванні нового вузла до ребру дерева. Ми тут опустили перевірку на помилки повернених функцій strsave і talloc значень (що нерозумно для практично працюючої програми).

Функція treeprint друкує дерево, починаючи з лівого піддерева; в кожному вузлі спочатку друкується ліве піддерево (всі слова, які молодше цього слова), потім саме слово, а потім праве піддерево (всі слова, які старше). Якщо ви невпевнено оперуєте з рекурсією, намалюйте дерево самі і надрукуйте його за допомогою функції treeprint; це одна з найяскравіших рекурсивних процедур, яку можна знайти.

treeprint (p) /\* print tree p recursively \*/

struct tnode \*p;

{

if (p != null) {

treeprint (p->left);

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

treeprint (p->right);

}

}

Практичне зауваження: якщо дерево стає "незбалансованим" через те, що слова надходять не в довільному порядку, то час роботи програми може рости занадто швидко. У гіршому випадку, коли надходять слова вже впорядковані, справжня програма здійснює дорогу імітацію лінійного пошуку. Існують різні узагальнення двійкового дерева, особливо 2-3 дерева і avl дерева, які не ведуть себе так "в гірших випадках", але ми не будемо тут на них зупинятися.

Перш ніж розлучитися з цим прикладом, доречно зробити невеличкий відступ в зв'язку з питанням про розподіл пам'яті. Ясно, що в програмі бажано мати тільки один розподільник пам'яті, навіть якщо йому доводиться розміщувати різні види об'єктів. Але якщо ми хочемо використовувати один розподільник пам'яті для обробки запитів на виділення пам'яті для покажчиків на змінні типу char і для покажчиків на struct tnode, то при цьому виникають два питання.

Перше: як виконати те існуюче на більшості реальних машин обмеження, що об'єкти певних типів повинні задовольняти вимогам вирівнювання (наприклад, часто цілі повинні розміщуватися в парних адресах)? Друге: як організувати опис, щоб впоратися з тим, що функція alloc повинна повертати різні види покажчиків?

Взагалі, вимоги вирівнювання легко виконати за рахунок виділення деякого зайвого простору, просто забезпечивши те, щоб розподільник пам'яті завжди повертав покажчик, що задовольняє всім обмеженням вирівнювання. Наприклад, на PDP-11 досить, щоб функція alloc завжди повертала парний покажчик, оскільки в парну адресу можна помістити будь-який тип об'єкта. Єдина витрата при цьому - зайвий символ при запиті на непарну довжину. Аналогічні дії робляться на інших машинах.

Таким чином, реалізація alloc може не виявити ту переносимість, але її використання буде стерпним. функція alloc з "лекції №5" не передбачає ніякого певного вирівнювання; в "лекції №8" ми продемонструємо, як правильно виконати це завдання.

Питання опису типу функції alloc є болісним для будь-якої мови, який серйозно ставиться до перевірки типів. Кращий спосіб в мові "C" - оголосити, що alloc повертає покажчик на змінну типу char, а потім явно перетворити цей покажчик до бажаного типу за допомогою операції переказу типів (на даний момент в таких ситуаціях відповідно до найбільш поширеним стандартом слід повертати void \* , тобто покажчик на void). Таким чином, якщо описати p в вигляді

char \*p;

то

(struct tnode \*) p

перетворює його в виразах в покажчик на структуру типу tnode. Отже, функцію talloc можна записати у вигляді:

struct tnode \*talloc()

{

char \*alloc();

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

}

це більш ніж достатньо для працюючих в даний час компіляторів, але це і найбезпечніший шлях з урахуванням майбутнього.

**Вправа 6-4**

Напишіть програму, яка читає "C" - програму і друкує в алфавітному порядку кожну групу імен змінних, які збігаються в перших семи символах, але відрізняються десь далі. (Зробіть так, щоб 7 було параметром).

**Вправа 6-5**

Напишіть програму видачі перехресних посилань, тобто програму, яка друкує список всіх слів документа і для кожного з цих слів друкує список номерів рядків, в які це слово входить.

**Вправа 6-6**

Напишіть програму, яка друкує слова зі свого файлу введення, розташовані в порядку убування частоти їх появи. Перед кожним словом надрукуйте число його появ.

#### 6.6. Пошук в таблиці

Для ілюстрації подальших аспектів використання структур в цьому розділі ми напишемо програму, яка являє собою вміст пакета пошуку в таблиці. Ця програма є типовим представником підпрограм управління символьними таблицями макропроцесора або компілятора. Розглянемо, наприклад, оператор #define мови "C". Коли зустрічається рядок виду

#define yes 1

то ім'я yes і замінюючий текст 1 поміщаються в таблицю. Пізніше, коли ім'я yes з'являється в операторі вигляду

inword = yes;

Oно має бути заміщено на 1.

Є дві основні процедури, які керують іменами і замінюють їх текстами. функція install (s, t) записує ім'я s і замінює текст t в таблицю; тут s і t просто символьні рядки. функція lookup (s) шукає ім'я s в таблиці і повертає або покажчик того місця, де це ім'я знайдено, або null, якщо цього імені в таблиці не виявилося.

При цьому використовується пошук по алгоритму хешування - надходить ім'я перетворюється в маленьке позитивне число, яке потім використовується для індексації масиву покажчиків. Елемент масиву вказує на початок ланцюгових блоків, що описують імена, які мають це значення хешування. Якщо ніякі імена при хешуванні не отримують за це значення, то елементом масиву буде null.

Блоком ланцюга є структура, яка містить покажчики на відповідне ім'я, на який замінює текст і на наступний блок в ланцюзі. Нульовий покажчик наступного блоку є ознакою кінця цього ланцюга.

struct nlist { /\* basic table entry \*/

char \*name;

char \*def;

struct nlist \*next; /\* next entry in chain \*/

};

*Масив покажчиків це просто*

#define hashsize 100

static struct nlist \*hashtab[hashsize] /\* pointer table \*/

Значення функції хешування, використовуваної обома функціями lookup і install, виходить просто як залишок від ділення суми символьних значень рядка на розмір масиву. (Це не найкращий можливий алгоритм, але його перевага полягає у виключній простоті).

hash(s) /\* form hash value for string \*/

char \*s;

{

int hashval;

for (hashval = 0; \*s != '\0'; )

hashval += \*s++;

return(hashval % hashsize);

}

В результаті процесу хешування видається початковий індекс в масиві hashtab; якщо даний рядок може бути десь знайдена, то саме в ланцюзі блоків, початок якій зазначено там. Пошук здійснюється функцією lookup. Якщо функція lookup знаходить, що даний об'єкт буде відразу присутній, то вона повертає покажчик на нього; якщо немає, то вона повертає null.

struct nlist \*lookup(s) /\* look for s in hashtab \*/

char \*s;

{

struct nlist \*np;

for (np = hashtab[hash(s)]; np != null;np=np->next)

if (strcmp(s, np->name) == 0)

return(np); /\* found it \*/

}

return(null); /\* not found \*/

функція install використовує функцію lookup для визначення, чи не присутній вже вводиться в даний момент ім'я; якщо це так, то нове визначення має витіснити старе. В іншому випадку створюється абсолютно новий елемент. Якщо з якої-небудь причини для нового елемента більше немає місця, то функція install повертає null.

struct nlist \*install(name, def) /\* put (name, def) \*/

char \*name, \*def;

{

struct nlist \*np, \*lookup();

char \*strsave(), \*alloc();

int hashval;

if((np = lookup(name)) == null) \( /\* not found \*/

np = (struct nlist \*) alloc(sizeof(\*np));

if (np == null)

return(null);

if ((np->name = strsave(name)) == null)

return(null);

hashval = hash(np->name);

np->next = hashtab[hashval];

hashtab[hashval] = np;

} else /\* already there \*/

free((np->def);/\* free previous definition \*/

if ((np->def = strsave(def)) == null)

return (null);

return(np);

}

функція strsave просто копіює рядок, зазначений в якості аргументу, в місце зберігання, отримане в результаті звернення до функції alloc. Ми вже привели цю функцію в "лекції №5". Так як звернення до функції alloc і free можуть відбуватися в будь-якому порядку і в зв'язку з проблемою вирівнювання, простий варіант функції alloc з "лекції №5" нам більше не підходить; дивіться "лекції №7" і "лекції №8".

**Вправа 6-7**

Напишіть процедуру, яка буде видаляти ім'я і визначення з таблиці, керованої функціями lookup і install.

**Вправа 6-8**

Розробіть просту, засновану на функціях цього розділу, версію процесора для обробки конструкцій #define, придатну для використання з "C" - програмами. Вам можуть також виявитися корисними функції getchar і ungetch.

#### 6.7. Поля

Коли питання економії пам'яті стає дуже істотним, то може виявитися необхідним поміщати в одне машинне слово кілька різних об'єктів; одне з особливо поширених вживань - набір однобітових ознак в застосуваннях, подібних символьним таблицями компілятора. Зовні обумовлені формати даних, такі як інтерфейси апаратних засобів також часто припускають можливість отримання слова по частинах.

Уявіть собі фрагмент компілятора, який працює з символьного таблицею. З кожним ідентифікатором програми пов'язана певна інформація, наприклад, є він чи ні ключовим словом, чи є він чи ні зовнішнім і / або статичним і т.д. Найкомпактніший спосіб закодувати таку інформацію - помістити набір однобітових ознак в окрему змінну типу char або int.

Звичайний спосіб, яким це робиться, полягає у визначенні набору "масок", що відповідають відповідним двійкового позиціях, як в

#define keyword 01

#define external 02

#define static 04

(Числа повинні бути ступенями двійки). Тоді обробка бітів зведеться до "жонглювання бітами" за допомогою операцій зсуву, маскування і доповнення, описаних нами в "лекції №2" .

Деякі часто зустрічаються ідіоми:

flags |= external | static;

включає біти external і static в flags, в той час як

flags &= ~(еxternal | static);

їх виключає, а

if ((flags & (external | static)) == 0) ...

істинно, якщо обидва біта вимкнені.

Хоча цими ідіомами легко опанувати, мова "C" в якості альтернативи пропонує можливість визначення і обробки полів всередині слова безпосередньо, а не за допомогою побітових логічних операцій. Поле - це набір суміжних бітів всередині однієї змінної типу int. Синтаксис визначення і обробки полів ґрунтується на структурах. Наприклад, символьну таблицю конструкції #define, наведену вище, можна б було замінити визначенням трьох полів:

struct {

unsigned is\_keyword : 1;

unsigned is\_extern : 1;

unsigned is\_static : 1;

} flags;

Тут визначається змінна з ім'ям flags, яка містить три 1-бітових поля. Наступне за двокрапкою число задає ширину поля в бітах. Поля описані як unsigned, щоб підкреслити, що вони дійсно будуть величинами без знака.

На окремі поля можна посилатися, як flags.is\_static, flags.is\_extern, flags.is\_keyword і т.д., тобто точно так же, як на інші члени структури. Поля поводяться подібно невеликим цілим без знаку і можуть брати участь в арифметичних виразах точно так же, як і інші цілі. Таким чином, попередні приклади більш природно переписати так:

flags.is\_extern = flags.is\_static = 1;

для включення бітів;

flags.is\_extern = flags.is\_static = 0;

для виключення бітів;

if (flags.is\_extern == 0 && flags.is\_static == 0)...

для їх перевірки.

Поле не може перекривати кордон int; якщо зазначена ширина така, що це має статися, то поле вирівнюється по межі наступного int. Полям можна й не присвоювати імена; неіменовані поля (тільки двокрапка і ширина) використовуються для заповнення вільного місця. Щоб змусити вирівнювання на кордон наступного int, можна використовувати спеціальну ширину 0.

При роботі з полями є ряд моментів, на які слід звернути увагу. Мабуть найбільш істотним є те, що відображаючи природу різних апаратних засобів, розподіл полів на деяких машинах здійснюється зліва направо, а на деяких справа наліво. Це означає, що хоча поля дуже корисні для роботи з внутрішньо певних структур даних, при поділі зовні визначених даних слід ретельно розглядати питання про те, який кінець надходить першим.

Інші обмеження, які слід мати на увазі: поля не мають знака; вони можуть зберігатися тільки в змінних типу int (або, що еквівалентно, типу unsigned); вони не є масивами; вони не мають адрес, так що до них не може бути застосована операція &.

#### 6.8. Об’єднання

Об’єднання - це змінна, яка в різні моменти часу може містити об'єкти різних типів і розмірів, причому компілятор бере на себе відстеження розміру і вимог вирівнювання. Об'єднання представляють можливість працювати з різними видами даних в одній області пам'яті, не вводячи в програму ніякої машинно-залежною інформації.

Як приклад, знову із символьної таблиці компілятора, припустимо, що константи можуть бути типу int, float або бути покажчиками на символи. Значення кожної конкретної константи повинно зберігатися у змінній відповідного типу, але все ж для управління таблицею найзручнішим було б, якщо це значення займало б один і той же обсяг пам'яті і зберігалося в тому ж самому місці незалежно від його типу. Це і є призначенням об'єднання - виділити окрему змінну, в якій можна законно зберігати будь-яку одну з змінних декількох типів. Як і в випадку полів, синтаксис грунтується на структурах.

union u\_tag {

int ival;

float fval;

char \*pval;

} uval;

змінна uval матиме чималий розмір, щоб зберігати найбільший із трьох типів, незалежно від машини, на якій здійснюється компіляція, - програма не буде залежати від характеристик апаратних засобів. Будь-який з цих трьох типів може бути присвоєно uval і потім використаний у виразах, поки таке використання сумісно: витягнутий тип повинен збігатися з останнім поміщеним типом. Справа програміста - стежити за тим, який тип зберігаються в об'єднанні у даний момент; якщо що-небудь зберігається як один тип, а витягується як інший, то результати будуть залежати від використовуваної машини.

Синтаксично доступ до членів об'єднання здійснюється наступним чином:

ім'я об'єднання.член

--------------------

або

покажчик об’єднання ->член

----------------------------

тобто точно так же, як і в випадку структур. Якщо для відстеження типу, що зберігається в даний момент в uval, використовується змінна utype, то можна зустріти таку ділянку програми:

if (utype == int)

printf("%d\n", uval.ival);

else if (utype == float)

printf("%f\n", uval.fval);

else if (utype == string)

printf("%s\n", uval.pval);

else

printf("bad type %d in utype\n", utype);

Об'єднання можуть з'являтися всередині структур і масивів і навпаки. Запис для звернення до члена об'єднання в структурі (або навпаки) абсолютно ідентична тій, яка використовується у вкладених структурах. Наприклад, в масиві структур, визначеному таким чином

struct {

char \*name;

int flags;

int utype;

union {

int ival;

float fval;

char \*pval;

} uval;

} symtab[nsym];

на змінну ival можна посилатися як

symtab[i].uval.ival

а на перший символ рядка pval як

\*symtab[i].uval.pval

По суті об'єднання є структурою, в якій всі члени мають нульовий зсув. Сама структура досить велика, щоб зберігати "найширший" член, і вирівнювання придатне для всіх типів, що входять в об'єднання. Як і в випадку структур, єдиними операціями, які в даний час можна проводити з об'єднаннями, є доступ до члена і витяг адреси; об'єднання не можуть бути присвоєні, передані функціям або повернуті ними. покажчики об'єднань можна використовувати в точно такій же манері, як і покажчики структур.

Програма розподілу пам'яті, що приводиться в "лекції №8", показує, як можна використовувати об'єднання, щоб зробити деяку змінну вирівняною за певним видом кордону пам'яті.

#### 6.9. Визначення типу

У мові "C" передбачена можливість, звана typedef для введення нових імен для типів даних. Наприклад, опис

typedef int length;

робить ім'я length синонімом для int. "Тип" length може бути використаний в описах, перекладів типів і т.д. Точно таким же чином, як і тип int:

length len, maxlen;

length \*lengths[];

Аналогічний опис

typedef char \*string;

робить string синонімом для char \*, тобто для покажчика на символи, що потім можна використовувати в описах виду

string p, lineptr[lines], alloc();

Зверніть увагу, що оголошений в конструкції typedef тип з'являється в позиції імені змінної, а не відразу за словом typedef. Синтаксично конструкція typedef подібна описами класу пам'яті extern, static і т. Д. Ми також використовували великі літери, щоб ясніше виділити імена.

У якості більш складного прикладу ми використовуємо конструкцію typedef для опису вузлів дерева, розглянутих раніше в цій лекції:

typedef struct tnode { /\* the basic node \*/

char \*word; /\* points to the text \*/

int count; /\* number of occurrences \*/

struct tnode \*left; /\* left child \*/

struct tnode \*right; /\* right child \*/

} treenode, \*treeptr;

В результаті отримуємо два нових ключових слова: treenode (структура) і treeptr (покажчик на структуру). Тоді функцію talloc можна записати у вигляді

treeptr talloc()

{

char \*alloc();

return((treeptr) alloc(sizeof(treenode)));

}

Необхідно підкреслити, що опис typedef не приводить до створення нового в будь-якому сенсі типу; воно тільки додає нове ім'я для деякого існуючого типу. При цьому не виникає і ніякої нової семантики: описані таким способом змінні мають точно такі ж властивості, що і змінні, описані явно. По суті конструкція typedef подібна до #define за винятком того, що вона інтерпретується компілятором і тому може здійснювати підстановки тексту, які виходять за межі можливостей макропроцесора мови "C".

Наприклад,

typedef int (\*pfi) ();

створює тип pfi для "покажчика функції, що повертає значення типу int", який потім можна було б використовувати в програмі сортування з "лекції №5" в контексті виду

pfi strcmp, numcmp, swap;

Є дві основні причини застосування описів typedef. Перша причина пов'язана з параметризацією програми, щоб полегшити вирішення проблеми переносимості. Якщо для типів даних, які можуть бути машинно-залежними, використовувати опис typedef, то при перенесенні програми на іншу машину доведеться змінити тільки ці описи. Одна з типових ситуацій полягає в використанні визначених за допомогою typedef імен для різних цілих величин і в подальшому відповідному виборі типів short, int і long для кожної наявної машини. Друге призначення typedef полягає в забезпеченні кращої документації для програми - тип з ім'ям treeptr може виявитися більш зручним для сприйняття, ніж тип, який описаний тільки як покажчик складної структури. І нарешті, завжди існує ймовірність, що в майбутньому компілятор або деяка інша програма, така як lint, зможе використовувати міститься в описах typedef інформацію для проведення деякої додаткової перевірки програми.