Кафедра комп'ютерної інженерії та електроніки

Назва дисципліни – Програмування

Викладач: Дзундза Богдан Степанович

1. Айра Пол Объектно-ориентированное программирование с использованием C++. Киев: ДиаСофт Лтд. – 1995. – 476 с. Глава 1.
2. Брайан Керниган, Деннис Ритчи. Язык программирования C. — Москва: Вильямс, 2015. — 304 с.
3. Зербіно Д., Цимбал Ю. Тенденції невизначеності у мовах програмування // Вфіник Національний університет “Львівська політехніка”, 2008, с. 183-186.
4. Устілкін В.В., Люта М.В., Розломій І.О. Дослідження мов програмування Java та C# для серверних платформ та робочих станцій // Журнал науковий огляд № 9 (30), 2016. – с. 1-10.
5. І. Дронюк, М. Назаркевич, Ю. Розроблення програмного забезпечення для захисту документів фоновими сітками // Вфіник Національний університет “Львівська політехніка”, 2009, с. 245-250.
6. М.Ю. Шабатура Спеціалізоване програмне забезпечення інтерактивної комп’ютеризованої системи // Національний університет «Львівська політехніка». 2012. С. 185-189.
7. І.Р. Пітух Матричні моделі архітектур розподілених комп’ютерних систем та методологія побудови алгоритму діагностування руху даних центральним сервером // «Искусственный интеллект» № 1. 2009. – С. 286-292.

Глава 1

Зачем нужно объектно-ориентированное программирование на C++?

В этой главе дается обзор языка программирования C++. Она также служит введением в использование C++ в качестве объектно-ориентированного языка и представляет ряд программ, которые демонстрируют объектно-ориентированные возможности. Сложность программ постепенно увеличивается, последние разделы иллюстрируют некоторые концепции объектно-ориентированного программирования. Такой подход должен дать вам ощущение того, как работает язык. Будучи обзорной, эта глава содержит достаточно сложный материал, который может быть просмотрен бегло или пропущен теми читателями, которые хотят начать с простых основ программирования. Они могут сразу перейти к следующей главе.

Объектно-ориентированное программирование (ООП) — основная методология программирования 90-х годов. Она является результатом тридцатилетнего опыта и практики, которые берут начало в языке Simula 67 и продолжаются в языках Smalltalk, LISP, Clu и в более поздних — Actor, Eiffel, Objective C, Java и C++. ООП — это стиль программирования, который фиксирует поведение реального мира так, что детали разработки скрыты, а это позволяет тому, кто решает задачу, мыслить в терминах, присущих этой задаче, а не программированию.

C++ был создан в начале 80-х Бьерном Страуструпом. Страуструп имел перед собой две цели: (1) оставить C++ совместимым с обычным C и (2) расширить C конструкциями ООП, основанными на понятии класса в Simula 67. Язык C был разработан Деннисом Ричи в начале 70-х для создания UNIX и задумывался как язык системного программирования. Постепенно он приобрел популярность не только как системный язык, но и в качестве языка общего назначения.

Конечная цель создания C++ — предоставить профессиональному программисту язык, который можно использовать при создании объектно-ориентированного программного обеспечения, не жертвуя эффективностью или переносимостью C. Первые

шаги на этом пути были сделаны Деннисом Ричи, а продолжили его Бьерн Страуструпп и растущее сообщество современных практикующих программистов.

Подчеркнем два аспекта, связанных с C++. Во-первых, он превосходен как язык общего назначения, благодаря своим новыми свойствам. Во-вторых, он удачен и как объектно-ориентированный язык программирования.

1.1. Объектно-ориентированное программирование

Объектно-ориентированное программирование — это программирование, сфокусированное на данных, причем данные и поведение неразрывно связаны. Вместе данные и поведение представляют собой класс, а объекты являются экземплярами класса. Например, многочлен имеет область значений, и она может изменяться такими операциями, как сложение и умножение многочленов.

ООП рассматривает вычисления как моделирование поведения. То, что моделируется, является объектами, представленными вычислительной абстракцией. Допустим, мы хотим улучшить наши навыки игры в покер; для этого мы должны научиться вычислять вероятность выпадения различных карточных комбинаций. Нам надо смоделировать перетасовку колоды, кроме того следует найти подходящий способ для оперирования мастями и достоинствами карт. Мы можем открыто[1] использовать названия мастей: пики, черви, бубны и трефы, но технически масти представлены целыми числами. Это внутреннее представление скрыто и, поэтому, не может повлиять на наши расчеты. Также как материальные колоды карт могут иметь разные физические свойства, но при этом ведут себя одинаковым ожидаемым от них образом, так и различные моделируемые колоды должны вести себя одинаково.

Мы будем применять термин *абстрактный тип данных, АТД* (abstract data type, ADT) для обозначения определяемого пользователем расширения исходных типов языка. АТД состоит из набора значений и операций, которые могут влиять на эти значения. Например, в C нет типа данных для комплексных чисел, а C++ позволяет добавить такой тип и интегрировать его с существующими.

Объекты (objects) являются переменными класса. Объектно-ориентированное программирование позволяет легко создавать и использовать АТД. Объектно-ориентированное программирование использует механизм *наследования* (inheritance). Наследование выгодно тем, что позволяет получать производные типы из уже определенных пользователем типов данных. Этот механизм сродни биологической таксономии. И грызуны, и кошки — млекопитающие. Если категория «млекопитающие» несет в себе информацию о свойствах и поведении, истинную для каждого из объектов соответствующего биологического класса, то создание категорий «кошачьи» и «грызуны» из категории «млекопитающие» чрезвычайно экономично.

В ООП объекты отвечают за свое поведение. Например, многочлены, комплексные числа, целые числа, числа с плавающей точкой — все это объекты, «понимающие» сложение. Каждый из этих типов включает в себя код для выполнения сложения. Компилятор предоставляет надлежащий код для целых и чисел с плавающей точкой. АТД «многочлен» содержит функцию, определяющую сложение, в соответствии с особенностями своей реализации. Поставщик АТД должен включать в него код для описания любого поведения, которое обычно можно ожидать от соответству-

[1] Под открытостью здесь понимается открытый (public) доступ к данным. — *Примеч. перев.*

ющих объектов. То, что объект сам отвечает за свое поведение, значительно упрощает задачу программирования для пользователя этого объекта.

Представим себе класс объектов под названием «фигуры». Если мы хотим нарисовать на экране какую-нибудь фигуру, нам надо знать, где будет находиться ее центр и как ее рисовать. Некоторые фигуры, например многоугольники, относительно легко нарисовать. В общем случае, процедура рисования фигуры может быть очень трудоемкой, так как, возможно, потребуется хранить большое число отдельных граничных точек. Напротив, вариант с многоугольником несомненно удобен. Если отдельная фигура прекрасно понимает, как себя нарисовать, программист при использовании такой фигуры должен лишь передать объекту сообщение «нарисовать(ся)».

В C++ новое понятие классов предоставляет механизм *инкапсуляции* (encapsulation) для реализации АТД. Инкапсуляция сочетает в себе, с одной стороны, внутренние детали реализации конкретного типа и, с другой, доступные извне операции и функции, которые могут действовать на объекты этого типа. Детали реализации могут быть недоступны для программы, которая использует данный тип. Например, стек может быть реализован как массив фиксированной длины, а доступные всем операции должны включать в себя функции `push` (поместить в стек) и `pop` (извлечь из стека). Изменение внутренней реализации на связный список не должно повлиять на то, как `push` и `pop` используются снаружи класса. Код, который использует АТД, называется *клиентом АТД*. Реализация стека скрыта от его клиентов.

К понятию ООП имеет отношение целый набор концепций, включая следующие:

Концепции ООП

- Моделирование действий из реального мира
- Наличие типов данных, определяемых пользователем
- Сокрытие деталей реализации
- Возможность многократного использование кода благодаря наследованию
- Интерпретация вызовов функций на этапе выполнения

Некоторые из этих понятий довольно расплывчаты, некоторые — абстрактны, другие носят общий характер. Чтобы смягчить возможные разочарования и смятение, мы постараемся проиллюстрировать ООП примерами, которые демонстрируют конкретные преимущества для программиста.

1.2. Пример программы на C++

C++ — это тесный союз программирования на низком и высоком уровнях. C был разработан как системный язык, близкий к машинному. C++ дополнен объектно-ориентированными свойствами, которые позволяют программисту создавать или импортировать библиотеки, присущие конкретной задаче. Пользователь может написать код на проблемном уровне, в то же время поддерживая контакт с машинным уровнем реализации деталей.

В файле hello.cpp

```
//Здороваемся с миром на C++

#include <iostream.h>      //библиотека ввода-вывода
#include <string>          //строковый тип
```

```
using namespace std; //пространство имен стандартных библиотек
inline void pr_message(string s = "Hello world!")
                  //Hello world! - Здравствуй, мир!
  { cout << s << endl; }
int main()
{
  pr_message();
}
```

При выполнении эта программа напечатает сообщение:

```
Hello world!
```

Программа на C++ — это набор объявлений и функций; выполнение начинается с функции `main()`. Когда программа компилируется, сначала выполняются все директивы препроцессора, такие, как директива `#include`. Она импортирует необходимый файл, обычно определения библиотеки. В нашем случае, библиотека ввода-вывода находится в файле *iostream* или *iostream.h.*

Библиотека *string* является частью стандартной библиотеки C++ и должна быть включена для использования объявления `string`. Пространство имен `std` зарезервировано для использования со стандартными библиотеками. Пространства имен были введены в ANSI C++ для предоставления области видимости, которая позволяет различным поставщикам кода избежать конфликта глобальных имен. Объявление `using` позволяет использовать идентификаторы из стандартной библиотеки без уточнения полного имени. При отсутствии такого объявления программа должна была бы использовать запись `std::string`.

Символ // (двойная косая черта) используется для добавления комментария в конце строки. Текст программы может располагаться в любом месте страницы, пустое пространство между строками игнорируется. Вообще, пробелы, комментарии, абзацные отступы в тексте программы, — все это используется лишь для создания хорошо документированной программы и не влияет на ее семантику.

Модификатор `inline` функции `pr_message()` сообщает компилятору, что ее надо компилировать, по возможности отказавшись от генерации инструкций вызова и возврата. Это позволяет обеспечить высокую эффективность. Как видно из текста, функция `pr_message()` имеет строковый параметр `s` со значением по умолчанию `"Hello world!"`. Это значит, что когда передается пустой список параметров, выполняется `pr_message("Hello world!")`.

Идентификатор `cout` определен в *iostream* как стандартный выходной поток, в большинстве систем направляющий вывод на экран. Идентификатор `endl` является стандартным *манипулятором* (manipulator), который очищает выходной буфер, так что печатается все содержимое буфера и осуществляется переход на новую строку. Оператор << направляет в `cout` (в данном случае) все, что за ним следует.

В C++ функция может возвращать неопределенный тип `void`. Это значит, что функция не возвращает никакого значения, как в случае с `pr_message()`. Специальная функция `main()` возвращает выполняющей системе целое значение; в большинстве случаев неявно возвращается ноль, что означает нормальное завершение. Другие значения `main()` должны возвращаться явно инструкцией `return`.

Вот еще один вариант `main()`:

В файле dinner.cpp

```
int main()
{
   pr_message();
   pr_message("Лаура Пол");
   pr_message("Пора обедать!");
}
```

При выполнении программа напечатает:

```
Hello world!
Лаура Пол
Пора обедать!
```

1.3. Инкапсуляция и расширяемость типов

ООП — это сбалансированный подход к написанию программного обеспечения. Данные и поведение упакованы вместе. Такая инкапсуляция создает определяемые пользователем типы данных, расширяющие собственные типы языка и взаимодействующие с ними. *Расширяемость типов* — это возможность добавлять к языку определяемые пользователем типы данных, которые так же легко использовать, как и собственные типы.

Абстрактный тип данных, например строка, является описанием идеального, всем известного поведения типа. Пользователь строки знает, что операции, такие как конкатенация или печать, имеют определенное поведение. Операции конкатенации и печати называются *методами*. Конкретная реализация АТД может иметь ограничения; например, строки могут быть ограничены по длине. Эти ограничения влияют на открытое всем поведение. В то же время, внутренние или закрытые детали реализации не влияют прямо на то, как пользователь видит объект. Например строка часто реализуется как массив; при этом внутренний базовый адрес элементов этого массива и его имя не существенны для пользователя.

На терминологию ООП сильно повлиял язык Smalltalk. Создатели Smalltalk хотели, чтобы программисты порвали со своими старыми привычками и приняли новую методологию программирования. Они изобрели термины, такие как *сообщение* и *метод*, взамен традиционных понятий *вызов функции* и *функция-член*.

Инкапсуляция — это способность скрывать внутренние детали при предоставлении открытого интерфейса к определяемому пользователем типу. В C++ для обеспечения инкапсуляции используются объявления класса и структуры (`class` и `struct`) в сочетании с ключевыми словами доступа `private` (закрытый), `protected` (защищенный) и `public` (открытый).

Ключевое слово `public` показывает, что доступ к членам, которые стоят за ним, является открытым безо всяких ограничений. Без этого ключевого слова члены класса по умолчанию закрыты. Закрытые члены доступны только функциям-членам своего класса. Открытые члены доступны любой функции внутри области видимости объявления класса. Закрытость позволяет спрятать часть реализации класса, предотвращая тем самым непредвиденные изменения структуры данных. Ограничение доступа или сокрытие данных является особенностью объектно-ориентированного про-

граммирования. Давайте напишем класс под названием my_string, в котором реализована ограниченная форма строки.

В файле string1.cpp

```
//Простейшая реализация типа my_string

const int max_len = 255;

class my_string{
public:          //всеобщий доступ к интерфейсу
   void assign(const char* st);
   int length() const { return len; }
   void print() const
      { cout << s << "\nДлина: " << len << endl;}
private:         //ограниченный доступ к реализации
   char s[max_len];
   int len;
};
```

Скрытое представление — это массив из max_len символов и переменная len, в которой хранится длина строки. Объявление функций-членов позволяет АТД иметь отдельные функции, влияющие на его закрытое представление. Например, функция-член length() возвращает длину строки. Функция-член print() выводит строку и ее длину. Функция-член assign() помещает символьную строку в скрытую переменную s, затем вычисляет ее длину и сохраняет ее в скрытой переменной len. Функции-члены, которые не изменяют значения переменных, объявлены как const.

Теперь мы можем использовать тип данных my_string, как если бы это был основной тип языка. Новый тип удовлетворяет стандартным правилам C. Код, который будет использовать этот тип, называется *клиентом* типа; он может обращаться только к открытым членам для того, чтобы воздействовать на переменные my_string.

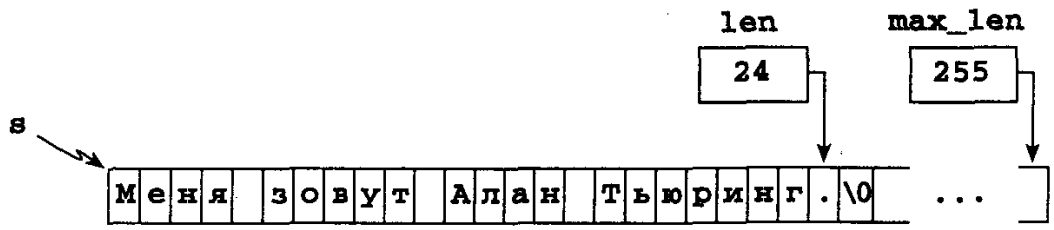
В файле string1.cpp

```
//Проверка класса my_string

int main()
{
   my_string one, two;
   char three[40] = {"Меня зовут Чарльз Бэббидж."};

   one.assign("Меня зовут Алан Тьюринг.");
   two.assign(three);
   cout << three;
   cout << "\nДлина: " << strlen(three) << endl;
        //Печать наиболее короткой из one и two
   if (one.length() <= two.length())
     one.print();
   else
     two.print();
}
```

Переменные one и two — типа my_string (см. рисунок). Переменная three имеет тип указателя на символ и не совместима с my_string.



Функции-члены вызываются с использованием *оператора «точка»* (*оператора доступа к члену структуры*). Как видно из их определений, эти функции-члены действуют на скрытые члены данных соответствующих переменных. Вот результат работы программы:

```
Меня зовут Чарльз Бэббидж.
Длина: 26
Меня зовут Алан Тьюринг.
Длина: 24
```

1.4. Конструкторы и деструкторы

В терминологии ООП переменная называется *объектом*. Функция-член, единственная работа которой заключается в инициализации объекта класса, называется *конструктором* (constructor). Во многих случаях инициализация предполагает динамическое распределение памяти. Конструкторы вызываются всякий раз, когда создается объект данного класса. Конструктор с одним аргументом может производить преобразование типов, если только при объявлении такого конструктора не используется ключевое слово explicit (явный). *Деструктор* (destructor) — это функция-член, задача которой состоит в том, чтобы завершить существование переменной класса. Этот процесс часто предполагает динамическое освобождение памяти. Деструктор вызывается неявно, когда автоматический объект выходит за пределы своей области видимости.

Давайте изменим наш пример my_string. Теперь память для каждой переменной класса будет выделяться динамически. Мы заменим закрытый член данных (массив) на указатель. Переделанный класс будет использовать конструктор для динамического выделения нужного объема памяти. Для этого применим оператор new.

В файле string2.cpp

```
//Реализация динамической my_string

class my_string {
public:        //конструктор
   explicit my_string(int n) { s = new char[n + 1]; len = n; }
   void assign(const char* st);
   int length() const { return len; }
   void print() const
      { cout << s << "\nДлина: " << len << endl; }
```

```
private:
   char*  s;
   int    len;
};
```

Для такого представления нам потребуется вариант функции-члена `assign()`, динамически выделяющий память:

```
void my_string::assign(const char* str)
{
   delete [] s;
   len = strlen(str);
   s = new char[len + 1];
   strcpy(s, str);
}
```

Имя конструктора совпадает с именем класса. При выделении памяти для инициализации объектов конструктор часто использует оператор `new`. Это унарный оператор, который получает в качестве аргумента тип данных, в частности, размер массива определенного типа. Оператор `new` выделяет необходимый объем памяти для хранения данного типа и возвращает указатель на адрес выделенной памяти. В предыдущем примере из свободной памяти должно быть выделено `n + 1` байт. Так, при объявлении:

```
my_string a(40), b(100);
```

для переменной `a` потребуется 41 байт , на которые укажет `a.s`, и 101 байт будет выделен для переменной `b`, а укажет на эти байты `b.s`. Мы добавили один байт для символа конца строки `\0`. Оператор `new` выделяет память на постоянной основе, и она не освобождается автоматически при выходе из блока. Если требуется освободить память, в класс должна быть включена функция-деструктор. Деструктор записывается, как обычная функция-член, имя которой совпадает с именем класса, но начинается с символа ~ (тильда). Для уничтожения объекта, распределенного оператором `new`, деструктор использует унарный оператор `delete`, — еще одно дополнение к языку, — чтобы автоматически освободить память, на которую направлено соответствующее выражение-указатель.

```
//Добавлено как функция-член к классу my_string
~my_string() { delete []s; }     //деструктор
```

Конструкторы часто перегружают, задавая несколько функций-конструкторов, чтобы предоставить несколько способов инициализации объекта. Рассмотрим, например, инициализацию `my_srting` указателем на символьное значение. Такой конструктор будет выглядеть так:

```
my_string(const char* p)
{
   len = strlen(p);
   s = new char[len + 1];
   strcpy(s, p);
}
```

Типичное объявление, вызывающее эту версию конструктора:

```
char* str = "Я пришел пешком.";
my_string a("Я приехал на автобусе."), b(str);
```

Желательно также иметь конструктор без аргументов:

```
my_string() { len = 0; s = new char[1]; }
```

Здесь объявление производится без аргументов, и по умолчанию будет выделен один байт памяти. А следующее объявление вызовет все три конструктора:

```
my_string a, b(20), c("Я приехал верхом.");
```

Перегруженный конструктор выбирается в зависимости от формы каждого из объявлений. Переменная a не имеет параметров, поэтому под нее будет выделен один байт. У переменной b есть целый параметр, и для нее отведен 21 байт. Переменная c получает в качестве параметра указатель на символьную строку "Я приехал верхом.", и для нее выделено 18 байт, причем заданная строка копируется в закрытый член s.

1.5. Перегрузка

Перегрузкой (overloading) называется практика придания нескольких значений оператору или функции. Выбор конкретного значения зависит от типов аргументов, полученных оператором или функцией. Давайте перегрузим функцию `print()` из предыдущего примера. Это будет второе определение функции `print()`.

В файле string3.cpp

```
class my_string {

public:     //общий доступ
   .....
   void  print() const
     { cout << s << "\nДлина: " << len << endl; }
   void  print(int n) const
     { for (int i = 0; i < n; ++i)
       cout << s << endl; }
   .....
}
```

Новая версия функции `print()` принимает целый аргумент. Она напечатает строку n раз.

```
three.print(2);     //печать строки three дважды
three.print(-1);    //строка three не печатается
```

Большинство операторов C++ может быть перегружено. Мы, например, перегрузим оператор +, чтобы представить конкатенацию двух строк. Для этого нам понадобятся два новых ключевых слова: `friend` и `operator`. Ключевое слово `operator` располагается перед изображающим оператор значком и вместе с этим значком заменяет то, что было бы именем функции в объявлении обычной функции. Ключевое слово `friend` предоставляет функции доступ к закрытым членам переменной класса. Дру-

жественная (friend) функция не является членом класса, но обладает привилегиями функции-члена того класса, другом которого она объявлена.

В файле string4.cpp

```
//Перегруженный оператор +

class my_string {
public:
   my_string() { len = 0; s = new char[1]; }
   explicit my_string(int n){ s = new char[n + 1]; len = n; }
   void  assign(const char* st);
   int  length() const { return len; }
   void  print() const
      { cout << s << "\nДлина: " << len << endl; }
   my_string& operator=(const my_string& a);
   friend my_string& operator+
      (const my_string& a, const my_string& b);
private:
   char* s;
   int  len;
};

//Перегрузка +
my_string& operator+(const my_string& a, const my_string& b)
{
   my_string* temp = new my_string(a.len + b.len);

   strcpy(temp->s, a.s);
   strcat(temp->s, b.s);
   return *temp;
}

void print(const char* c) //print с областью видимости файла
                          //(не путать с print из my_string!)
{
   cout << c << "\nДлина: " << strlen(c) << endl;
}

int main()
{
   my_string one, two, both;
   char three[40] = {"Меня зовут Чарльз Бэббидж."};

   one.assign("Меня зовут Алан Тьюринг.");
   two.assign(three);
   print(three); //Вызов print с областью видимости файла
                 //Печать наболее короткой строки из one или two
   if (one.length() <= two.length())
     one.print();  //вызов функции-члена print
```

```
    else
      two.print();
    both = one + two;  //плюс перегружен как конкатенация
    both.print();
}
```

Разбор функции operator+()

- `my_string& operator+(const my_string& a, const my_string& b)`

Плюс перегружен. У него два аргумента, оба — `my_string`. Аргументы передаются по ссылке. Запись вида *тип*`&` *идентификатор* объявляет идентификатор как переменную-ссылку. Ключевое слово `const` показывает, что аргументы не могут быть изменены.

- `my_string* temp = new my_string(a.len + b.len);`

Функция должна возвратить значение типа `my_string`. Этот локальный указатель будет использован чтобы вернуть значение `my_string` после выполнения конкатенации. Для выделения достаточного объема памяти используется оператор `new`.

- `return *temp;`

разыменованный указатель `temp` ссылается на объединенную my_string.

1.6. Шаблоны и обобщенное программирование

Ключевое слово `template` используется в C++ для обеспечения *параметрического полиморфизма*. Параметрический полиморфизм позволяет использовать один и тот же код применительно к разным типам, причем тип является параметром кода. Код пишется обобщенно. Особенно важно применение такой техники при написании общих *контейнерных классов*. Контейнерный класс используется для хранения данных определенного типа. Стеки, векторы, деревья, списки — все это примеры стандартных контейнерных классов. Вот пример контейнерного класса `stack`, определенного как параметризованный тип:

В файле tstack.cpp

```
//Реализация шаблона stack

template <class TYPE>
class stack {
public:
  explicit stack(int size = 1000) : max_len(size)
    { s = new TYPE[size]; top = EMPTY; }
  ~stack() { delete []s; }
  void  reset() { top = EMPTY; }          //очистить стек
  void  push(TYPE c) { s[++top] = c; }  //поместить в стек
  TYPE  pop() { return s[top--]; }       //извлечь из стека
```

```
    TYPE  top_of() { return s[top]; }     //считать верхний
                                          //элемент
    bool  empty() { return (top == EMPTY); }  //стек пуст?
    bool  full() {return (top == max_len-1);} //стек заполнен?
  private:
    enum {EMPTY = -1};
    TYPE*  s;
    int   max_len;      //максимальная длина
    int   top;          //вершина
  };
```

Объявление класса выглядит так:

```
template <class идентификатор>
```

Идентификатор является аргументом шаблона, который на самом деле будет подставлен на место произвольного типа. Везде в тексте определения класса аргумент шаблона может использоваться в качестве имени типа. Сам аргумент подставляется при фактическом объявлении. Вот пример объявления `stack`, использующий вышесказанное:

```
stack<char>    stk_ch;          //стек из 1000 символов
stack<char*>   stk_str(200);    //стек из 200 указателей
stack<complex> stk_cmplx(100); //стек из 100 комплексных чисел
```

Этот механизм позволяет нам не переписывать каждый раз объявления классов, которые отличаются лишь типами. При работе с таким типом в качестве части объявления всегда должны использоваться угловые скобки < >. Вот две функции, использующие шаблон `stack`:

```
//Обращение последовательности указателей char*,
//представляющих строку

void reverse(char* str[], int n)
{
  stack<char*> stk(n); //в стеке хранятся char*

  for (int i = 0; i < n; ++i)
    stk.push(str[i]);
  for (int i = 0; i < n; ++i)
    str[i] = stk.pop();
}
```

В функции `reverse()` используется стек `stack<char*>`. Он принимает `n` строк, а затем строки извлекаются в обратном порядке.

```
//Инициализация стека комплексными числами из массива

void init(complex c[], stack<complex>& stk, n)
{
  for (int i = 0; i < n; ++i)
    stk.push(c[i]);
}
```

В функции `init()` переменная типа `stack<complex>` передается по ссылке; в стек помещаются n комплексных чисел.

1.7. Стандартная библиотека шаблонов (STL)

Стандартная библиотека шаблонов (Standard Template Library — STL) является стандартной библиотекой C++, которая позволяет использовать обобщенное программирование для множества распространенных структур данных и алгоритмов. Эта библиотека предлагает: контейнерные классы, такие как векторы, очереди, отображения; средства перебора элементов контейнера с помощью классов итераторов; а также алгоритмы для различного использования контейнеров — функции сортировки и поиска данных и т. п. Мы предлагаем краткое описание STL с акцентом на эти три основные составляющие, — контейнеры, итераторы и алгоритмы, — которые создают основу для обобщенного программирования.

Библиотека построена с использованием шаблонов, ее дизайн вполне ортогонален. Компоненты можно комбинировать друг с другом, используя в качестве параметров как «родные» типы C++, так и типы, определяемые пользователем. Необходимо только следить за правильностью инстанцирования элементов библиотеки STL.

В файле stl_list.cpp

```
//Использование контейнера list (список)

#include <iostream>
#include <list>       //контейнер списков
#include <numeric>    //нужно для функции accumulate()
using namespace std;

void print(const list<double> &lst)
{  //используем итератор для путешествия по lst
   list<double>::const_iterator where;

   for (where = lst.begin(); where !=lst.end(); ++where)
     cout << *where << '\t';
   cout << endl;
}

int main()
{
   double w[4] = {0.9, 0.8, 88, -99.99};
   list<double> z;

   for( int i = 0; i < 4; ++i)
     z.push_front(w[i]);
   print(z);
   z.sort();
   print(z);
   cout << "Сумма равна "<< accumulate(z.begin(), z.end(), 0.0)
        << endl;
}
```

В этом примере списочный контейнер должен хранить переменные с двойной точностью. Массив из таких переменных помещается в список. Функция `print()` использует итератор для печати по очереди всех элементов списка. Итераторы имеют стандартный интерфейс, частью которого являются функции-члены `begin()` и `end()` для определения начала и конца контейнера. Кроме того, интерфейс включает в себя алгоритм сортировки — функцию-член `sort()`. Функция `accumulate()` использует 0.0 в качестве начального значения и вычисляет сумму элементов списочного контейнера путем прохода от начальной позиции `z.begin()` до конечной `z.end()`.

1.8. Наследование

Особенностью ООП является поощрение повторного использования кода при помощи механизма наследования. Новый класс *производится* от существующего, называемого *базовым* классом. Производный класс использует члены базового класса, но может также изменять и дополнять их.

Многие типы представляют собой вариации на темы существующих. Часто бывает утомительно разрабатывать новый код для каждого из них. Кроме того, новый код — новые ошибки. Производный класс наследует описание базового класса, делая ненужными повторную разработку и тестирование кода. Отношения наследования иерархичны. Иерархия — это метод, позволяющий копировать элементы во всем их многообразии и сложности. Она вводит классификацию объектов. Например, в периодической системе элементов есть газы. Они обладают свойствами, присущими всем элементам системы. Инертные газы — следующий важный подкласс. Иерархия заключается в том, что инертный газ, например, аргон — это газ, а газ, в свою очередь является элементом системы. Такая иерархия позволяет легко толковать поведение инертных газов. Мы знаем, что их атомы содержат протоны и электроны, что верно и для прочих элементов. Мы знаем, что они пребывают в газообразном состоянии при комнатной температуре, как все газы. Мы знаем, что ни один газ из подкласса инертных газов не вступает в обычную химическую реакцию ни с одним из элементов, и это свойство всех инертных газов.

Методология объектно-ориентированного проектирования

1. Выбери надлежащую совокупность типов.
2. Спроектируй взаимосвязи типов в коде, используя наследование.

Предположим, нам надо разработать базу данных для университета. Студенческий отдел должен вести учет студентов разных категорий. Базовый класс, который нам необходимо разработать, будет фиксировать описание студента. Две основные категории студентов — это аспирант и просто студент (не аспирант).

Вот пример наследования:

В файле student1.cpp

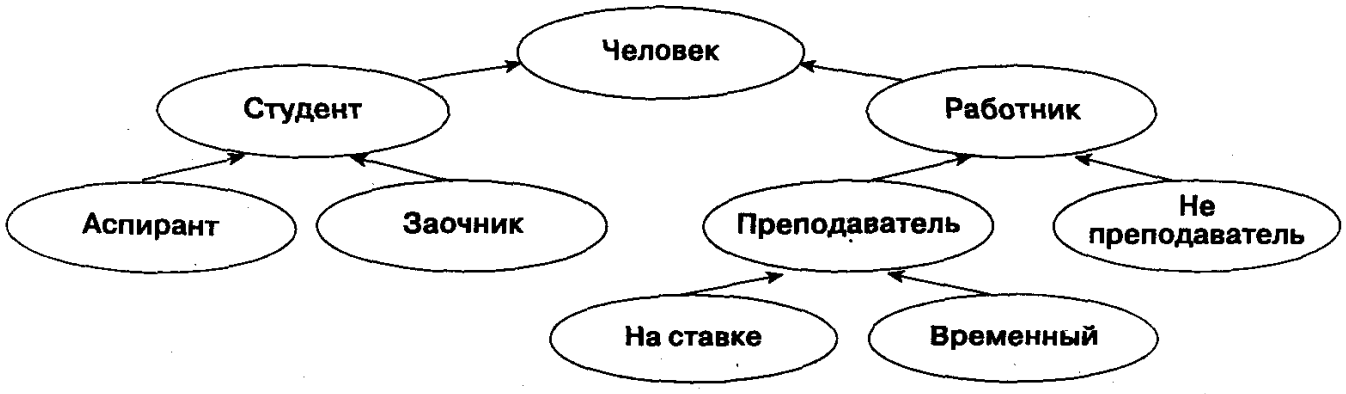
```
            //тип финансовой поддержки
enum support { ta, ra, fellowship, other };
            //курс (год обучения)
enum year { fresh, soph, junior, senior, grad };
```

```
class student {          //класс студентов
public:                  //конструктору передаются имя,
                         //номер, средний балл, курс
  student(char* nm, int id, double g, year x);
  void print() const;
private:
  int student_id;        //номер
  double gpa;            //средний балл
  year y;                //курс
  char name[30];         //имя
};

class grad_student : public student { //класс аспирантов
public:                  //конструктору дополнительно передаются:
                         //тип финансовой поддержки,
                         //название кафедры, тема диссертации
  grad_student(char* nm, int id, double g,
               year x, support t, char* d, char* th);
  void print() const;
private:
  support  s;            //финансовая поддержка
  char   dept[10];       //кафедра
  char   thesis[80];     //диссертация
};
```

В этом примере `grad_student` — производный класс, а `student` — базовый. Использование в заголовке производного класса ключевого слова `public`, следующего за двоеточием, означает, что открытые члены класса `student` должны наследоваться как открытые члены `grad_student`. Закрытые члены базового класса недоступны производному классу. Открытое наследование означает также, что производный класс `grad_student` является подтипом `student`.

Структуры наследования образуют каркас для построения достаточно общих систем. Например, база данных, содержащая информацию обо всех людях в университете, может быть унаследована от базового класса `person` (человек). Базовый класс `student` можно использовать для создания производного класса студентов-юристов, как следующей значимой категории объектов. Аналогично, `person` может служить базовым классом для различных категорий работников. Иерархическая структура наследования показана на рисунке.



1.9. Полиморфизм

Полиморфная функция или оператор имеет несколько вариантов. Например, в C++ оператор деления полиморфен. Если аргументы оператора деления целые, выполняется целочисленное деление. Если один или оба аргумента — с плавающей точкой, используется деление для чисел с плавающей точкой.

В C++ имя функции или оператора может быть перегружено. То, какая именно из одноименных функций будет вызвана, определяется *сигнатурой*, которая представляет собой перечисление типов аргументов в списке параметров функции.

Например, в выражении деления

```
a/b     //тип определяется естественными правилами приведения
```

результат зависит от аргументов, которые автоматически приводятся к более широкому типу. Так, если оба аргумента целые, результатом будет целое частное. Но если хотя бы один из аргументов — с плавающей точкой, результат тоже будет с плавающей точкой.

Другой пример — оператор вывода

```
cout << a;  //полиморфизм через перегруженную функцию
```

в котором оператор сдвига << вызывает функцию, которая умеет выводить объекты типа a. Таким образом, если a — целое, то и вывод будет целым. Если оно — с плавающей точкой, то и результат — с плавающей точкой.

Полиморфизм локализует ответственность за поведение. Клиентский код часто не требует пересмотра, когда к системе добавляется функциональность с помощью улучшений кода АТД.

Реализация набора процедур для задания типа геометрической фигуры может основываться на исчерпывающем описании произвольной фигуры. Например, структура

```
srtuct shape {   //фигура
                 //окружность, прямоугольник
   enum { CIRCLE, RECTANGLE, ..... } e_val;
   double center, radius; //центр, радиус
   .....
};
```

должна содержать все члены, необходимые для произвольной фигуры, которую на данном этапе может нарисовать система. Она должна включать переменную перечисляемого типа, с тем, чтобы фигуру можно было идентифицировать. Тогда процедура вычисления площади должна быть записана так:

```
duoble area(shape* s)
{
   switch(s -> e_val) {
   case CIRCLE: return(PI * s -> radius * s -> radius);
   case RECTANGLE: return(s -> heght * s -> width);
   .....
   }
}
```

Что означает пересмотр этого кода с целью включить новую фигуру? Понадобится дополнительная строка case в теле кода и дополнительные члены структуры. К сожалению, это повлечет за собой изменения во всем теле кода, поскольку каждая процедура построена так, что придется добавить дополнительный case, даже когда такой case выступает в роли еще одной метки в уже существующем case. Таким образом, локальное улучшение требует глобальных перемен.

Объектно-ориентированная техника программирования на C++ использует иерархию фигур для решения той же проблемы. Иерархия очевидна, когда круг и прямоугольник наследуются из фигуры. В процессе пересмотра кода возможные улучшения делаются в новом производном классе, так что дополнительные описания локализованы. Программист замещает смысл любой измененной процедуры. В нашем случае, необходимо задать новую формулу для вычисления площади. Клиентский код, не использующий новый тип, остается без изменений; код, который должен быть улучшен добавлением нового типа, обычно изменяется минимально.

Программа на C++, следующая изложенной схеме, использует shape как *абстрактный базовый класс*. Это класс, содержащий хотя бы одну чисто виртуальную функцию, как показано в следующем коде:

В файле shape1.cpp

```
//shape – абстрактный базовый класс (фигура)

class shape {
public:
   virtual double area() = 0;      //чисто виртуальная функция
                                   //(площадь)
};

class rectangle : public shape { //прямоугольник
public:
   rectangle(double h, double w) :
           height(h), width(w){}
   double area() {return (height * width);}
private:
   double height, width;           //высота, ширина
};

class circle : public shape {     //окружность
public:
   circle(double r) : radius(r) { }
   double area() {return(3.14159 * radius * radius);}
private:
   double radius;                  //радиус
};
```

Клиентский код для вычисления произвольной площади полиморфен. Надлежащая функция area() выбирается на этапе выполнения:

```
shape* ptr_shape;
   .....
```

```
cout << "Площадь = " << ptr_shape -> area();
.....
```

Теперь представим, что мы хотим дополнить нашу иерархию типов и разработать класс `square` (квадрат):

```
class square : public rectangle {
public:
   square(double h) : rectangle(h,h) { }
   double area() { return (rectangle::area()); }
};
```

Клиентский код остается без изменений, в отличие от не-ООП-кода, рассмотренного выше.

Иерархическая схема должна минимизировать интерфейс передачи параметров. Каждый уровень стремится сокрыть в себе (в пределах своей реализации) детали строения, на которые влияет вызов функции. В обычной схеме (не ООП) это иногда может сопровождаться внесением изменений в глобально определяемые данные. Такая практика почти всегда осуждается, потому что она ведет к неясному стилю программирования с побочными эффектами, а это затрудняет отладку, пересмотр и поддержку кода.

1.10. Исключения в C++

C++ вводит механизм обработки исключений, чувствительный к контексту. Контекстом для возбуждения исключения является блок `try`. Обработчики, объявляемые ключевым словом `catch`, находятся ниже блока `try`.

Исключение возбуждается с помощью ключевого слова `throw`. Исключение будет обработано вызовом соответствующего обработчика, выбранного из списка, который идет сразу за блоком `try`. Ниже приведен простой пример:

В файле stackex.cpp

```
//Конструктор стека с исключениями
stack::stack(int n)
{
   if (n < 1)
      throw (n);     //хотим положительное значение
   p = new char[n]; //создается символьный стек
   if (p == 0)      //new возвращает 0 при неудачном завершении
      throw ("СВОБОДНАЯ ПАМЯТЬ ИСЧЕРПАНА");
   top = EMPTY;
   max_len = n;
}

void g()
{
   try {
      stack a(n), b(n);
      .....
   }
```

```
    catch (int n) {.....}          //некорректный размер
    catch (char* error) {.....}    //свободная память исчерпана
}
```

Первый `throw()` имеет целый аргумент и соответствует сигнатуре `catch(int n)`. Этот обработчик должен выполнить соответствующие действия, когда конструктору в качестве аргумента передается некорректный размер массива. Например, вывести сообщение об ошибке и прервать программу. Второй `throw()` получает в качестве аргумента указатель на символ и соответствует сигнатуре `catch(char* error)`.

1.11. Преимущества объектно-ориентированного программирования

Центральным элементом ООП является инкапсуляция совокупности данных и соответствующих им операций. Понятие класса с его функциями-членами и членами данных предоставляет программисту подходящий для реализации инкапсуляции инструмент. Переменные класса являются объектами, которыми можно управлять.

Кроме того, классы обеспечивают сокрытие данных. Права доступа могут устанавливаться или ограничиваться для любой группы функций, которым необходим доступ к деталям реализации. Тем самым обеспечивается модульность и надежность.

Еще одной важной концепцией ООП является поощрение повторного использования кода с помощью механизма наследования. Суть этого механизма — получение нового производного класса из существующего, называемого базовым. При создания производного класса базовый класс может быть дополнен или изменен. Таким путем могут создаваться иерархии родственных типов данных, которые используют общий код.

Объектно-ориентированное программирование зачастую более сложно, чем обычное процедурное программирование, как оно выглядит на С. Существует по крайней мере один дополнительный шаг на этапе проектирования, перед алгоритмизацией и кодированием. Он состоит в разработке таких типов данных, которые соответствовали бы поставленной проблеме. Зачастую проблема решается «обобщеннее», чем это действительно необходимо.

Есть уверенность, что использование ООП принесет дивиденды в нескольких отношениях. Решение будет более модульным, следовательно, более понятным и простым для модификации и обслуживания. Кроме того, такое решение будет более пригодно для повторного использования. Например, если в программе нужен стек, то он легко заимствуется из существующего кода. В обычном процедурном языке программирования такие структуры данных часто «вмонтированы» в алгоритм и не могут экспортироваться.

ООП много значит для многих людей. Все попытки дать ему определение напоминают старания слепых мудрецов описать слона. Я бы предложил еще одно утверждение:

ООП = расширяемость типов + полиморфизм

```
LINE[I] = '\0'
RETURN(I)
```

2. Типы, операции и выражения

Переменные и константы являются основными объектами, с которыми оперирует программа. Описания перечисляют переменные, которые будут использоваться, указывают их тип и, возможно, их начальные значения. Операции определяют, что с ними будет сделано. выражения объединяют переменные и константы для получения новых значений. Все это — темы настоящей главы.

2.1. Имена переменных

Хотя мы этого сразу прямо не сказали, существуют некоторые ограничения на имена переменных и символических констант. Имена составляются из букв и цифр; первый символ должен быть буквой. Подчеркивание "_" тоже считается буквой; это полезно для удобочитаемости длинных имен переменных. Прописные и строчные буквы различаются; традиционная практика в "с" — использовать строчные буквы для имен переменных, а прописные — для символических констант.

Играют роль только первые восемь символов внутреннего имени, хотя использовать можно и больше. Для внешних имен, таких как имена функций и внешних переменных, это число может оказаться меньше восьми, так как внешние имена используются различными ассемблерами и загрузчиками. Детали приводятся в приложении а. Кроме того, такие ключевые слова как IF, ELSE, INT, FLOAT и т.д., зарезервированы: вы не можете использовать их в качестве имен переменных. (Они пишутся строчными буквами).

Конечно, разумно выбирать имена переменных таким образом, чтобы они означали нечто, относящееся к назначению переменных, и чтобы было менее вероятно спутать их при написании.

2.2. Типы и размеры данных

Языке "С" имеется только несколько основных типов данных:

CHAR один байт, в котором может находиться один символ из внутреннего набора символов.

INT Целое, обычно соответствующее естественному размеру целых в используемой машине.

FLOAT С плавающей точкой одинарной точности.

DOUBLE С плавающей точкой двойной точности.

Кроме того имеется ряд квалификаторов, которые можно использовать с типом INT: SHORT (короткое), LONG (длинное) и UNSIGNED (без знака). Квалификаторы SHORT и LONG указывают на различные размеры целых. Числа без знака подчиняются законам арифметики по модулю 2 в степени N, где N — число битов в INT; числа без знаков всегда положительны. Описания с квалификаторами имеют вид:

```
SHORT INT X;
```

```
LONG INT Y;
UNSIGNED INT Z;
```

Слово INT в таких ситуациях может быть опущено, что обычно и делается.

Количество битов, отводимых под эти объекты зависит от имеющейся машины; в таблице ниже приведены некоторые характерные значения.

Таблица 1

! DEC PDP-11 HONEYWELL IBM 370 6000 8/32 !	INTERDATA	!
ASCII ASCII EBCDIC	ASCII	!
CHAR 8-BITS 9-BITS 8 INT 16 36 32 SHORT 16 36 16 LONG 32 36 32 FLOAT 32 36 32 DOUBLE 64 72 64	BITS 8 32 16 32 32 64	BITS ! ! ! ! ! !
----------	----------	--------

Цель состоит в том, чтобы SHORT и LONG давали возможность в зависимости от практических нужд использовать различные длины целых; тип INT отражает наиболее "естественный" размер конкретной машины. Как вы видите, каждый компилятор свободно интерпретирует SHORT и LONG в соответствии со своими аппаратными средствами. Все, на что вы можете твердо полагаться, это то, что SHORT не длиннее, чем LONG.

2.3. Константы

Константы типа INT и FLOAT мы уже рассмотрели. Отметим еще только, что как обычная

```
123.456e-7,
```

так и "научная" запись

```
0.12e3
```

для FLOAT является законной.

Каждая константа с плавающей точкой считается имеющей тип DOUBLE, так что обозначение "E" служит как для FLOAT, так и для DOUBLE.

Длинные константы записываются в виде 123L. Обычная целая константа, которая слишком длинна для типа INT, рассматривается как LONG.

Существует система обозначений для восьмеричных и шестнадцатеричных констант: лидирующий 0(нуль) в константе типа INT указывает на восьмеричную константу, а стоящие впереди 0X соответствуют шестнадцатеричной константе. Например, десятичное число 31 можно записать как 037 в восьмеричной форме и как 0X1F в шестнадцатеричной. Шестнадцатеричные и восьмеричные константы могут также заканчиваться буквой L, что делает их относящимися к типу LONG.

2.3.1. Символьная константа

Символьная константа — это один символ, заключенный в одинарные кавычки, как, например, 'x'. Значением символьной константы является численное значение этого символа во внутреннем машинном наборе символов. Например, в наборе символов ASCII символьный нуль, или '0', имеет значение 48, а в коде EBCDIC — 240, и оба эти значения совершенно отличны от числа

0. Написание '0' вместо численного значения, такого как 48 или 240, делает программу не зависящей от конкретного численного представления этого символа в данной машине. Символьные константы точно так же участвуют в численных операциях, как и любые другие числа, хотя наиболее часто они используются в сравнении с другими символами. Правила преобразования будут изложены позднее.

Некоторые неграфические символы могут быть представлены как символьные константы с помощью условных последовательностей, как, например, \N (новая строка), \T (табуляция), \0 (нулевой символ), \\ (обратная косая черта), \' (одинарная кавычка) и т.д. Хотя они выглядят как два символа, на самом деле являются одним. Кроме того, можно сгенерировать произвольную последовательность двоичных знаков размером в байт, если написать

```
'\DDD'
```

где DDD — от одной до трех восьмеричных цифр, как в

```
#DEFINE  FORMFEED  '\014'  /* FORM FEED */
```

Символьная константа '\0', изображающая символ со значением 0, часто записывается вместо целой константы 0, чтобы подчеркнуть символьную природу некоторого выражения.

2.3.2. Константное выражение

Константное выражение — это выражение, состоящее из одних констант. Такие выражения обрабатываются во время компиляции, а не при прогоне программы, и соответственно могут быть использованы в любом месте, где можно использовать константу, как, например в

```
#DEFINE MAXLINE 1000
CHAR LINE[MAXLINE+1];
```

или

```
SECONDS = 60 * 60 * HOURS;
```

2.3.3. Строчная константа

Строчная константа — это последовательность, состоящая из нуля или более символов, заключенных в двойные кавычки, как, например,

```
"I AM A STRING"              /* я — строка */
```

или

```
""          /* NULL STRING */              /* нуль-строка */
```

Кавычки не являются частью строки, а служат только для ее ограничения. те же самые условные последовательности, которые использовались в символьных константах, применяются и в строках; символ двойной кавычки изображается как \".

С технической точки зрения строка представляет собой массив, элементами которого являются отдельные символы. Чтобы программам было удобно определять конец строки, компилятор автоматически помещает в конец каждой строки нуль-символ \0. Такое представление означает, что не накладывается конкретного ограничения на то, какую длину может иметь строка, и чтобы определить эту длину, программы должны просматривать строку полностью. При этом для физического хранения строки требуется на одну ячейку памяти больше, чем число заключенных в кавычки символов. Следующая функция STRLEN(S) вычисляет длину символьной строки S не считая конечный символ \0.

```
STRLEN(S)              /* RETURN LENGTH OF S */
CHAR  S[];

    INT I;

    I = 0;
    WHILE (S[I] != '\0')
            ++I;
    RETURN(I);
```

Будьте внимательны и не путайте символьную константу со строкой, содержащей один символ: 'X' — это не то же самое, что "X". Первое — это отдельный символ, использованный с целью получения численного значения, соответствующего букве x в машинном наборе символов. Второе — символьная строка, состоящая из одного символа (буква x) и \0.

2.4. Описания

Все переменные должны быть описаны до их использования, хотя некоторые описания делаются неявно, по контексту. Описание состоит из спецификатора типа и следующего за ним списка переменных, имеющих этот тип, как, например,

```
INT LOWER, UPPER, STEP;
CHAR C, LINE[1000];
```

Переменные можно распределять по описаниям любым образом; приведенные выше списки можно с тем же успехом записать в виде

```
INT LOWER;
INT UPPER;
INT STEP;
CHAR C;
CHAR LINE[1000];
```

Такая форма занимает больше места, но она удобна для добавления комментария к каждому описанию и для последующих модификаций.

Переменным могут быть присвоены начальные значения внутри их описания, хотя здесь имеются некоторые ограничения. Если за именем переменной следуют знак равенства и константа, то эта константа служит в качестве инициализатора, как, например, в

```
CHAR BACKSLASH = '\\';
INT I = 0;
FLOAT EPS = 1.0E-5;
```

Если рассматриваемая переменная является внешней или статической, то инициализация проводится только один раз, согласно концепции до начала выполнения программы. Инициализируемым явно автоматическим переменным начальные значения присваиваются при каждом обращении к функции, в которой они описаны. Автоматические переменные, не инициализируемые явно, имеют неопределенные значения, (т.е. мусор). Внешние и статические переменные по умолчанию инициализируются нулем, но, тем не менее, их явная инициализация является признаком хорошего стиля.

Мы продолжим обсуждение вопросов инициализации, когда будем описывать новые типы данных.

2.5. Арифметические операции

Бинарными арифметическими операциями являются +, -, *, / и операция деления по модулю %. Имеется унарная операция -, но не существует унарной операции +.

При делении целых дробная часть отбрасывается. Выражение

```
X % Y
```

дает остаток от деления X на Y и, следовательно, равно нулю,

когда x делится на Y точно. Например, год является високосным, если он делится на 4, но не делится на 100, исключая то, что делящиеся на 400 годы тоже являются високосными. Поэтому

```
IF(YEAR % 4 == 0 && YEAR % 100 != 0 \!\! YEAR % 400 == 0) год високосный
```

ELSE

год невисокосный

Операцию % нельзя использовать с типами FLOAT или DOUBLE.

Операции + и — имеют одинаковое старшинство, которое младше одинакового уровня старшинства операций *, / и %, которые в свою очередь младше унарного минуса. Арифметические операции группируются слева направо. (Сведения о старшинстве и ассоциативности всех операций собраны в таблице в конце этой главы). Порядок выполнения ассоциативных и коммутативных операций типа + и — не фиксируется; компилятор может перегруппировывать даже заключенные в круглые скобки выражения, связанные такими операциями. таким образом, a+(B+C) может быть вычислено как (A+B)+C. Это редко приводит к какому-либо расхождению, но если необходимо обеспечить строго определенный порядок, то нужно использовать явные промежуточные переменные.

Действия, предпринимаемые при переполнении и антипереполнении (т.е. При получении слишком маленького по абсолютной величине числа), зависят от используемой машины.

2.6. Операции отношения и логические операции

Операциями отношения являются

=> > =< <

все они имеют одинаковое старшинство. Непосредственно за ними по уровню старшинства следуют операции равенства и неравенства:

== !=

которые тоже имеют одинаковое старшинство. операции отношения младше арифметических операций, так что выражения типа I<LIM-1 понимаются как I<(LIM-1), как и предполагается.

Логические связки && и \!\! более интересны. Выражения, связанные операциями && и \!\!, вычисляются слева направо, причем их рассмотрение прекращается сразу же как только становится ясно, будет ли результат истиной или ложью. учет этих свойств очень существенен для написания правильно работающих программ. Рассмотрим, например, оператор цикла в считывающей строку функции GETLINE, которую мы написали в главе

1.

```
FOR(I=0;I<LIM-1 && (C=GETCHAR()) != '\N' && C != EOF; ++I)
S[I]=C;
```

Ясно, что перед считыванием нового символа необходимо проверить, имеется ли еще место в массиве S, так что условие I<LIM-1 должно проверяться первым. И если это условие не выполняется, мы не должны считывать следующий символ.

Так же неудачным было бы сравнение 'C' с EOF до обращения к функции GETCHAR : прежде чем проверять символ, его нужно считать.

Старшинство операции && выше, чем у \!\!, и обе они младше операций отношения и равенства. Поэтому такие выражения, как

```
I<LIM-1 && (C = GETCHAR()) != '\N' && C != EOF
```

не нуждаются в дополнительных круглых скобках. Но так как

операция != старше операции присваивания, то для достижения

правильного результата в выражении

```
(C = GETCHAR()) != '\N'
```

кобки необходимы.

Унарная операция отрицания ! Преобразует ненулевой или истинный операнд в 0, а нулевой или ложный операнд в 1. Обычное использование операции ! Заключается в записи

```
IF( ! INWORD )
```

Вместо

```
IF( INWORD == 0 )
```

Трудно сказать, какая форма лучше. Конструкции типа ! INWORD

Читаются довольно удобно ("если не в слове"). Но в более сложных случаях они могут оказаться трудными для понимания.

Упражнение 2-1.

Напишите оператор цикла, эквивалентный приведенному выше оператору FOR, не используя операции &&.

2.7. Преобразование типов

Если в выражениях встречаются операнды различных типов, то они преобразуются к общему типу в соответствии с небольшим набором правил. В общем, автоматически производятся только преобразования, имеющие смысл, такие как, например, преобразование целого в плавающее в выражениях типа F+I. Выражения же, лишенные смысла, такие как использование переменной типа FLOAT в качестве индекса, запрещены.

Во-первых, типы CHAR и INT могут свободно смешиваться в арифметических выражениях: каждая переменная типа CHAR автоматически преобразуется в INT. Это обеспечивает значительную гибкость при проведении определенных преобразований символов. Примером может служить функция ATOI, которая ставит в соответствие строке цифр ее численный эквивалент.

```
ATOI(S)                 /* CONVERT S TO INTEGER */
CHAR S[];
```

```
    INT I, N;

    N = 0;
    FOR ( I = 0; S[I]>='0' && S[I]<='9'; ++I)
            N = 10 * N + S[I] — '0';
    RETURN(N);
```

КАК Уже обсуждалось в главе 1, выражение

```
S[I] — '0'
```

имеет численное значение находящегося в S[I] символа, потому что значение символов '0', '1' и т.д. образуют возрастающую последовательность расположенных подряд целых положительных чисел.

Другой пример преобразования CHAR в INT дает функция LOWER, преобразующая данную прописную букву в строчную. Если выступающий в качестве аргумента символ не является прописной буквой, то LOWER возвращает его неизменным. Приводимая ниже программа справедлива только для набора символов ASCII.

```
LOWER(C) /* CONVERT C TO LOWER CASE; ASCII ONLY */
INT C;

    IF ( C >= 'A' && C <= 'Z' )
            RETURN( C + '@' — 'A');
    ELSE   /*@ Записано вместо 'A' строчного*/
            RETURN(C);
```

Эта функция правильно работает при коде ASCII, потому что численные значения, соответствующие в этом коде прописным и строчным буквам, отличаются на постоянную величину, а каждый алфавит является сплошным — между a и Z нет ничего, кроме букв. Это последнее замечание для набора символов EBCDIC систем IBM 360/370 оказывается несправедливым, в силу чего эта программа на таких системах работает неправильно — она преобразует не только буквы.

При преобразовании символьных переменных в целые возникает один тонкий момент. Дело в том, что сам язык не указывает, должны ли переменным типа CHAR соответствовать численные значения со знаком или без знака. Может ли при преобразовании CHAR в INT получиться отрицательное целое? К сожалению, ответ на этот вопрос меняется от машины к машине, отражая расхождения в их архитектуре. На некоторых машинах (PDP-11, например) переменная типа CHAR, крайний левый бит которой содержит 1, преобразуется в отрицательное целое ("знаковое расширение"). На других машинах такое преобразование сопровождается добавлением нулей с левого края, в результате чего всегда получается положительное число.

Определение языка "С" гарантирует, что любой символ из стандартного набора символов машины никогда не даст отрицательного числа, так что эти символы можно свободно использовать в выражениях как положительные величины. Но произвольные комбинации двоичных знаков, хранящиеся как символьные переменные на некоторых машинах, могут дать отрицательные значения, а на других положительные.

Наиболее типичным примером возникновения такой ситуации является сучай, когда значение 1 используется в качестве EOF. Рассмотрим программу

```
CHAR C;
C = GETCHAR();
IF ( C == EOF )
    ...
```

На машине, которая не осуществляет знакового расширения, переменная 'с' всегда положительна, поскольку она описана как CHAR, а так как EOF отрицательно, то условие никогда не выполняется. Чтобы избежать такой ситуации, мы всегда предусмотрительно использовали INT вместо CHAR для любой переменной, получающей значение от GETCHAR.

Основная же причина использования INT вместо CHAR не связана с каким-либо вопросом о возможном знаковом расширении. просто функция GETCHAR должна передавать все возможные символы (чтобы ее можно было использовать для произвольного ввода) и, кроме того, отличающееся значение EOF. Следовательно значение EOF не может быть представлено как CHAR, а должно храниться как INT.

Другой полезной формой автоматического преобразования типов является то, что выражения отношения, подобные I>J, и логические выражения, связанные операциями && и \!\!, по определению имеют значение 1, если они истинны, и 0, если они ложны. Таким образом, присваивание

```
ISDIGIT = C >= '0' && C <= '9';
```

полагает ISDIGIT равным 1, если с — цифра, и равным 0 в противном случае. (В проверочной части операторов IF, WHILE, FOR и т.д. "Истинно" просто означает "не нуль").

Неявные арифметические преобразования работают в основном, как и ожидается. В общих чертах, если операция типа + или *, которая связывает два операнда (бинарная операция), имеет операнды разных типов, то перед выполнением операции "низший" тип преобразуется к "высшему" и получается результат "высшего" типа. Более точно, к каждой арифметической операции применяется следующая последовательность правил преобразования.

- Типы CHAR и SHORT преобразуются в INT, а FLOAT в DOUBLE.

- Затем, если один из операндов имеет тип DOUBLE, то другой преобразуется в DOUBLE, и результат имеет тип DOUBLE.

- В противном случае, если один из операндов имеет тип LONG, то другой преобразуется в LONG, и результат имеет тип LONG.

- В противном случае, если один из операндов имеет тип UNSIGNED, то другой преобразуется в UNSIGNED и результат имеет тип UNSIGNED.

- В противном случае операнды должны быть типа INT, и результат имеет тип INT.

Подчеркнем, что все переменные типа FLOAT в выражениях преобразуются в DOUBLE; в "С" вся плавающая арифметика выполняется с двойной точностью.

Преобразования возникают и при присваиваниях; значение правой части преобразуется к типу левой, который и является типом результата. Символьные переменные преобразуются в целые либо со знаковым расширением, либо без него, как описано выше. Обратное преобразование INT в CHAR ведет себя хорошо — лишние биты высокого порядка просто отбрасываются. Таким образом

```
INT I;
CHAR C;

I = C;
C = I;
```

значение 'c' не изменяется. Это верно независимо от того,

вовлекается ли знаковое расширение или нет.

Если x типа FLOAT, а I типа INT, то как

```
x = I;
```

так и

```
I = x;
```

приводят к преобразованиям; при этом FLOAT преобразуется в

INT отбрасыванием дробной части. Тип DOUBLE преобразуется во

FLOAT округлением. Длинные целые преобразуются в более короткие целые и в переменные типа CHAR посредством отбрасывания лишних битов высокого порядка.

Так как аргумент функции является выражением, то при передаче функциям аргументов также происходит преобразование типов: в частности, CHAR и SHORT становятся INT, а FLOAT становится DOUBLE. Именно поэтому мы описывали аргументы функций как INT и DOUBLE даже тогда, когда обращались к ним с переменными типа CHAR и FLOAT.

Наконец, в любом выражении может быть осуществлено ("принуждено") явное преобразование типа с помощью конструкции, называемой перевод (CAST). В этой конструкции, имеющей вид

```
(имя типа) выражение
```

Выражение преобразуется к указанному типу по правилам преобразования, изложенным выше. Фактически точный смысл операции перевода можно описать следующим образом: выражение как бы присваивается некоторой переменной указанного типа, которая затем используется вместо всей конструкции. Например, библиотечная процедура SQRT ожидает аргумента типа DOUBLE и выдаст бессмысленный ответ, если к ней по небрежности обратятся с чем-нибудь иным. таким образом, если N — целое, то выражение

```
SQRT((DOUBLE) N)
```

до передачи аргумента функции SQRT преобразует N к типу DOUBLE. (Отметим, что операция перевод преобразует значение N в надлежащий тип; фактическое содержание переменной N при этом не изменяется). Операция перевода имрация перевода имеет тот же уровень старшинства, что и другие унарные операции, как указывается в таблице в конце этой главы.

Упражнение 2-2.

Составьте программу для функции HTOI(S), которая преобразует строку шестнадцатеричных цифр в эквивалентное ей целое значение. При этом допустимыми цифрами являются цифры от 1 до 9 и буквы от a до F.

2.8. Операции увеличения и уменьшения

В языке "С" предусмотрены две необычные операции для увеличения и уменьшения значений переменных. Операция увеличения ++ добавляет 1 к своему операнду, а операция уменьшения — вычитает 1. Мы часто использовали операцию ++ для увеличения переменных, как, например, в

```
IF(C == '\N')
    ++I;
```

Необычный аспект заключается в том, что ++ и — можно использовать либо как префиксные операции (перед переменной, как в ++N), либо как постфиксные (после переменной: N++). Эффект в обоих случаях состоит в увеличении N. Но выражение ++N увеличивает переменную N до использования ее значения, в то время как N++ увеличивает переменную N после того, как ее значение было использовано. Это означает, что в контексте, где используется значение переменной, а не только эффект увеличения, использование ++N и N++ приводит к разным результатам. Если N = 5, то

```
x = N++;
```

устанавливает x равным 5, а

```
x = ++N;
```

полагает x равным 6. В обоих случаях N становится равным 6.

Операции увеличения и уменьшения можно применять только к переменным; выражения типа x=(I+J)++ являются незаконными.

В случаях, где нужен только эффект увеличения, а само значение не используется, как, например, в

```
IF ( C == '\N' )
    NL++;
```

выбор префиксной или постфиксной операции является делом

вкуса. но встречаются ситуации, где нужно использовать именно ту или другую операцию. Рассмотрим, например, функцию SQUEEZE(S,C), которая удаляет символ 'c' из строки S, каждый раз, как он встречается.

```
SQUEEZE(S,C)          /* DELETE ALL C FROM S */
CHAR S[];
INT C;

    INT I, J;

    FOR ( I = J = 0; S[I] != '\0'; I++)
            IF ( S[I] != C )
      S[J++] = S[I];
    S[J] = '\0';
```

Каждый раз, как встечается символ, отличный от 'c', он копируется в текущую позицию J, и только после этого J увеличивается на 1, чтобы быть готовым для поступления следующего символа. Это в точности эквивалентно записи

```
    IF ( S[I] != C )
            S[J] = S[I];
            J++;
```

Другой пример подобной конструкции дает функция GETLINE, которую мы запрограммировали в главе 1, где можно заменить

```
IF ( C == '\N' )
    S[I] = C;
    ++I;
```

более компактной записью

```
IF ( C == '\N' )
    S[I++] = C;
```

В качестве третьего примера рассмотрим функцию STRCAT(S,T), которая приписывает строку т в конец строки S, образуя конкатенацию строк S и т. При этом предполагается, что в S достаточно места для хранения полученной комбинации.

```
STRCAT(S,T)             /* CONCATENATE T TO END OF S */
CHAR S[], T[];  /* S MUST BE BIG ENOUGH */

    INT I, J;

    I = J = 0;
    WHILE (S[I] != '\0') / *FIND END OF S */
            I++;
  WHILE((S[I++] = T[J++]) != '\0') /*COPY T*/
            ;
```

Так как из T в S копируется каждый символ, то для подготовки к следующему прохождению цикла постфиксная операция ++ применяется к обеим переменным I и J.

Упражнение 2-3.

Напишите другой вариант функции SQUEEZE(S1,S2), который удаляет из строки S1 каждый символ, совпадающий с каким-либо символом строки S2.

Упражнение 2-4.

Напишите программу для функции ANY(S1,S2), которая находит место первого появления в строке S1 какого-либо символа из строки S2 и, если строка S1 не содержит символов строки S2, возвращает значение -1.

2.9. Побитовые логические операции

В языке предусмотрен ряд операций для работы с битами; эти операции нельзя применять к переменным типа FLOAT или DOUBLE.

&	Побитовое AND
\!	Побитовое включающее OR
^	побитовое исключающее OR
<<	сдвиг влево
>>	сдвиг вправо
\^	дополнение (унарная операция)

"\" иммитирует вертикальную черту.

Побитовая операция AND часто используется для маскирования некоторого множества битов; например, оператор

```
C = N & 0177
```

передает в 'с' семь младших битов N, полагая остальные равными нулю. Операция 'э' побитового OR используется для включения битов:

```
C = X э MASK
```

устанавливает на единицу те биты в x, которые равны единице в MASK.

Следует быть внимательным и отличать побитовые операции & и 'э' от логических связок && и \!\!, Которые подразумевают вычисление значения истинности слева направо. Например, если x=1, а Y=2, то значение x&Y равно нулю, в то время как значение X&&Y равно единице./почему?/

Операции сдвига << и >> осуществляют соответственно сдвиг влево и вправо своего левого операнда на число битовых позиций, задаваемых правым операндом. Таким образом, x<<2 сдвигает x влево на две позиции, заполняя освобождающиеся биты нулями, что эквивалентно умножению на 4. Сдвиг вправо величины без знака заполняет освобождающиеся биты на некоторых машинах, таких как PDP-11, заполняются содержанием знакового бита /"арифметический сдвиг"/, а на других — нулем /"логический сдвиг"/.

Унарная операция \^ дает дополнение к целому; это означает, что каждый бит со значением 1 получает значение 0 и наоборот. Эта операция обычно оказывается полезной в выражениях типа

```
X & \^077
```

где последние шесть битов x маскируются нулем. Подчеркнем,

что выражение X&\^077 не зависит от длины слова и поэтому

предпочтительнее, чем, например, X&0177700, где предполагается, что x занимает 16 битов. Такая переносимая форма не требует никаких дополнительных затрат, поскольку \^077 является константным выражением и, следовательно, обрабатывается во время компиляции.

Чтобы проиллюстрировать использование некоторых операций с битами, рассмотрим функцию GETBITS(X,P,N), которая возвращает /сдвинутыми к правому краю/ начинающиеся с позиции p поле переменной x длиной N битов. мы предполагаем, что крайний правый бит имеет номер 0, и что N и p — разумно заданные положительные числа. например, GETBITS(x,4,3) возвращает сдвинутыми к правому краю биты, занимающие позиции 4,3 и 2.

```
GETBITS(X,P,N)  /* GET N BITS FROM POSITION P */
UNSIGNED X, P, N;

    RETURN((X >> (P+1-N)) & \^(\^0 << N));
```

Операция X >> (P+1-N) сдвигает желаемое поле в правый конец слова. Описание аргумента X как UNSIGNED гарантирует, что при сдвиге вправо освобождающиеся биты будут заполняться нулями, а не содержимым знакового бита, независимо от того, на какой машине пропускается программа. Все биты константного выражения \^0 равны 1; сдвиг его на N позиций влево с помощью операции \^0<<N создает

маску с нулями в N крайних правых битах и единицами в остальных; дополнение \^ создает маску с единицами в N крайних правых битах.

Упражнение 2-5.

Переделайте GETBITS таким образом, чтобы биты отсчитывались слева направо.

Упражнение 2-6.

Напишите программу для функции WORDLENGTH(), вычисляющей длину слова используемой машины, т.е. Число битов в переменной типа INT. Функция должна быть переносимой, т.е. Одна и та же исходная программа должна правильно работать на любой машине.

Упражнение 2-7.

Напишите программу для функции RIGHTROT(N,B), сдвигающей циклически целое N вправо на B битовых позиций.

Упражнение 2-8.

Напишите программу для функции INVERT(X,P,N), которая инвертирует (т.е. Заменяет 1 на 0 и наоборот) N битов X, начинающихся с позиции P, оставляя другие биты неизмененными.

2.10. Операции и выражения присваивания

Такие выражения, как

```
I = I + 2
```

в которых левая часть повторяется в правой части могут быть

записаны в сжатой форме

```
I += 2
```

используя операцию присваивания вида +=.

Большинству бинарных операций (операций подобных +, которые имеют левый и правый операнд) соответствует операция присваивания вида оп=, где оп — одна из операций

```
+ — * / % << >> & \^ \!
```

Если е1 и е2 — выражения, то

```
е1 оп= е2
```

эквивалентно

```
е1 = (е1) оп (е2)
```

за исключением того, что выражение е1 вычисляется только

один раз. Обратите внимание на круглые скобки вокруг е2:

```
X *= Y + 1
```

то

```
X = X * (Y + 1)
```

не

```
X = X * Y + 1
```

В качестве примера приведем функцию BITCOUNT, которая подсчитывает число равных 1 битов у целого аргумента.

```
BITCOUNT(N)   /* COUNT 1 BITS IN N */
UNSIGNED N;
(
INT B;
FOR (B = 0; N != 0; N >>= 1)
    IF (N & 01)
            B++;
RETURN(B);
)
```

Не говоря уже о краткости, такие операторы приваивания имеют то преимущество, что они лучше соответствуют образу человеческого мышления. Мы говорим: "прибавить 2 к I" или "увеличить I на 2", но не "взять I, прибавить 2 и поместить результат опять в I". Итак, I += 2. Кроме того, в громоздких выражениях, подобных

```
YYVAL[YYPV[P3+P4] + YYPV[P1+P2]] += 2
```

Такая операция присваивания облегчает понимание программы, так как читатель не должен скрупулезно проверять, являются ли два длинных выражения действительно одинаковыми, или задумываться, почему они не совпадают. Такая операция присваивания может даже помочь компилятору получить более эффективную программу.

Мы уже использовали тот факт, что операция присваивания имеет некоторое значение и может входить в выражения; самый типичный пример

```
WHILE ((C = GETCHAR()) != EOF)
```

присваивания, использующие другие операции присваивания (+=, -= и т.д.) также могут входить в выражения, хотя это случается реже.

Типом выражения присваивания является тип его левого операнда.

Упражнение 2-9.

В двоичной системе счисления операция X&(X-1) обнуляет самый правый равный 1 бит переменной X.(почему?) используйте это замечание для написания более быстрой версии функции BITCOUNT.

2.11. Условные выражения

Операторы

```
IF (A > B)
    Z = A;
ELSE
    Z = B;
```

конечно вычисляют в Z максимум из a и в. Условное выражение, записанное с помощью тернарной операции "?:", предоставляет другую возможность для записи этой и аналогичных конструкций. В выражении

```
e1? E2 : e3
```

сначала вычисляется выражение e1. Если оно отлично от нуля (истинно), то вычисляется выражение e2, которое и становится значением условного выражения. В противном случае вычисляется e3, и оно становится значением условного выражения. Каждый раз вычисляется только одно из выражения e2 и e3. Таким образом, чтобы положить Z равным максимуму из a и в, можно написать

```
Z = (A > B)? A : B;   /* Z = MAX(A,B) */
```

Следует подчеркнуть, что условное выражение действительно является выражением и может использоваться точно так же, как любое другое выражение. Если e2 и e3 имеют разные типы, то тип результата определяется по правилам преобразования, рассмотренным ранее в этой главе. например, если F имеет тип FLOAT, а N — тип INT, то выражение

```
(N > 0)? F : N
```

Имеет тип DOUBLE независимо от того, положительно ли N или нет.

Так как уровень старшинства операции?: очень низок, прямо над присваиванием, то первое выражение в условном выражении можно не заключать в круглые скобки. Однако, мы все же рекомендуем это делать, так как скобки делают условную часть выражения более заметной.

Использование условных выражений часто приводит к коротким программам. Например, следующий ниже оператор цикла печатает N элементов массива, по 10 в строке, разделяя каждый столбец одним пробелом и заканчивая каждую строку (включая последнюю) одним символом перевода строки.

```
OR (I = 0; I < N; I++)
```

```
PRINTF("%6D%C",A[I],(I%10==9 \!\! I==N-1)? '\N' : ' ')
```

Символ перевода строки записывается после каждого десятого элемента и после N-го элемента. За всеми остальными элементами следует один пробел. Хотя, возможно, это выглядит мудреным, было бы поучительным попытаться записать это, не используя условного выражения.

Упражнение 2-10.

Перепишите программу для функции LOWER, которая переводит прописные буквы в строчные, используя вместо конструкции IF-ELSE условное выражение.

2.12. Старшинство и порядок вычисления

В приводимой ниже таблице сведены правила старшинства и ассоциативности всех операций, включая и те, которые мы еще не обсуждали. Операции, расположенные в одной строке, имеют один и тот же уровень старшинства; строки расположены в порядке убывания старшинства. Так, например, операции *, / и % имеют одинаковый уровень старшинства, который выше, чем уровень операций + и -.

OPERATOR	ASSOCIATIVITY
() [] -> .	LEFT TO RIGHT
! \^ ++ — — (TYPE) * & SIZEOF	RIGHT TO LEFT
* / %	LEFT TO RIGHT
+ -	LEFT TO RIGHT
<< >>	LEFT TO RIGHT
< <= > >=	LEFT TO RIGHT
== !=	LEFT TO RIGHT
&	LEFT TO RIGHT
^	LEFT TO RIGHT
\!	LEFT TO RIGHT
&&	LEFT TO RIGHT
\!\!	LEFT TO RIGHT
?:	RIGHT TO LEFT
= += -= ETC.	RIGHT TO LEFT
, (CHAPTER 3)	LEFT TO RIGHT

данные из верхнего и нижнего регистров сортируются вместе, так что буква 'a' прописное и 'a' строчное оказываются соседними, а не разделенными целым алфавитом.

Упражнение 5-13.

Добавьте необязательный аргумент -D ("словарное упорядочивание"), при наличии которого сравниваются только буквы, числа и пробелы. Позаботьтесь о том, чтобы эта функция работала и вместе с -F.

Упражнение 5-14.

Добавьте возможность обработки полей, так чтобы можно было сортировать поля внутри строк. Каждое поле должно сортироваться в соответствии с независимым набором необязательных аргументов. (предметный указатель этой книги сортировался с помощью аргументов -DF для категории указателя и с -N для номеров страниц).

6. Структуры

Структура — это набор из одной или более переменных, возможно различных типов, сгруппированных под одним именем для удобства обработки. (В некоторых языках, самый известный из которых паскаль, структуры называются "записями").

Традиционным примером структуры является учетная карточка работающего: "служащий" описывается набором атрибутов таких, как фамилия, имя, отчество (ф.и.о.), адрес, код социального обеспечения, зарплата и т.д. Некоторые из этих атрибутов сами могут оказаться структурами: ф.и.о. Имеет несколько компонент, как и адрес, и даже зарплата.

Структуры оказываются полезными при организации сложных данных особенно в больших программах, поскольку во многих ситуациях они позволяют сгруппировать связанные данные таким образом, что с ними можно обращаться, как с одним целым, а не как с отдельными объектами. В этой главе мы постараемся продемонстрировать то, как используются структуры. Программы, которые мы для этого будем использовать, больше, чем многие другие в этой книге, но все же достаточно умеренных размеров.

6.1. Основные сведения

Давайте снова обратимся к процедурам преобразования даты из главы 5. Дата состоит из нескольких частей таких, как день, месяц, и год, и, возможно, день года и имя месяца. Эти пять переменных можно объеденить в одну структуру вида:

```
STRUCT DATE \(
INT     DAY;
INT     MONTH;
INT     YEAR;
INT     YEARDAY;
CHAR    MON_NAME[4];
\);
```

Описание структуры, состоящее из заключенного в фигурные скобки списка описаний, начинается с ключевого слова STRUCT. За словом STRUCT может следовать

необязательное имя, называемое ярлыком структуры (здесь это DATe). Такой ярлык именует структуры этого вида и может использоваться в дальнейшем как сокращенная запись подробного описания.

Элементы или переменные, упомянутые в структуре, называются членами. Ярлыки и члены структур могут иметь такие же имена, что и обычные переменные (т.е. Не являющиеся членами структур), поскольку их имена всегда можно различить по контексту. Конечно, обычно одинаковые имена присваивают только тесно связанным объектам.

Точно так же, как в случае любого другого базисного типа, за правой фигурной скобкой, закрывающей список членов, может следовать список переменных.

Оператор

```
STRUCT   \( ...\) X,Y,Z;
```

синтаксически аналогичен

```
INT X,Y,Z;
```

в том смысле, что каждый из операторов описывает X, Y и Z в

качестве переменных соотвествующих типов и приводит к выделению для них памяти.

Описание структуры, за которым не следует списка переменных, не приводит к выделению какой-либо памяти; оно только определяет шаблон или форму структуры. Однако, если такое описание снабжено ярлыком, то этот ярлык может быть использован позднее при определении фактических экземпляров структур. Например, если дано приведенное выше описание DATE, то

```
STRUCT  DATE D;
```

определяет переменную D в качестве структуры типа DATE.

Внешнюю или статическую структуру можно инициализировать, поместив вслед за ее определением список инициализаторов для ее компонент:

```
STRUCT DATE D=\( 4, 7, 1776, 186, "JUL"\);
```

Член определенной структуры может быть указан в выражении с помощью конструкции вида

имя структуры . Член

Операция указания члена структуры "." связывает имя структуры и имя члена. В качестве примера определим LEAP (признак високосности года) на основе даты, находящейся в структуре D,

```
LEAP = D.YEAR % 4 == 0 && D.YEAR % 100 != 0
    \!\! D.YEAR % 400 == 0;
```

или проверим имя месяца

```
IF (STRCMP(D.MON_NAME, "AUG") == 0) ...
```

Или преобразуем первый символ имени месяца так, чтобы оно начиналось со строчной буквы

```
D.MON_NAME[0] = LOWER(D.MON_NAME[0]);
```

Структуры могут быть вложенными; учетная карточка служащего может фактически выглядеть 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 содержит две структуры типа DATE . Если мы определим EMP как

```
STRUCT PERSON EMP;
```

то

```
EMP.BIRTHDATE.MONTH
```

будет ссылаться на месяц рождения. Операция указания члена структуры "." ассоциируется слева направо.

6.2. Структуры и функции

В языке "С" существует ряд ограничений на использование структур. Обязательные правила заключаются в том, что единственные операции, которые вы можете проводить со структурами, состоят в определении ее адреса с помощью операции & и доступе к одному из ее членов. Это влечет за собой то, что структуры нельзя присваивать или копировать как целое, и что они не могут быть переданы функциям или возвращены ими. (В последующих версиях эти ограничения будут сняты). На указатели структур эти ограничения однако не накладываются, так что структуры и функции все же могут с удобством работать совместно. И наконец, автоматические структуры, как и автоматические массивы, не могут быть инициализированы; инициализация возможна только в случае внешних или статических структур.

Давайте разберем некоторые из этих вопросов, переписав с этой целью функции перобразования даты из предыдущей главы так, чтобы они использовали структуры. Так как правила запрещают непосредственную передачу структуры функции, то мы должны либо передавать отдельно компоненты, либо передать указатель всей структуры. Первая возможность демонстрируется на примере функции DAY_OF_YEAR, как мы ее написали в главе 5:

```
D.YEARDAY = DAY_OF_YEAR(D.YEAR, D.MONTH, D.DAY);
```

другой способ состоит в передаче указателя. если мы опишем HIREDATE как

```
STRUCT  DATE HIREDATE;
```

и перепишем DAY_OF_YEAR нужным образом, мы сможем тогда написать

```
HIREDATE YEARDAY = DAY_OF_YEAR(&HIREDATE);
```

передавая указатель на HIREDATE функции DAY_OF_YEAR . Функция должна быть модифицирована, потому что ее аргумент теперь является указателем, а не списком переменных.

```
    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);
      \)
```

Описание

```
STRUCT DATE *PD;
```

говорит, что PD является указателем структуры типа DATE. Запись, показанная на примере

```
PD->YEAR
```

является новой. Если P — указатель на структуру, то

P-> член структуры

обращается к конкретному члену. (Операция -> — это знак минус, за которым следует знак ">".)

Так как PD указывает на структуру, то к члену YEAR можно обратиться и следующим образом

```
(*PD).YEAR
```

но указатели структур используются настолько часто, что запись -> оказывается удобным сокращением. Круглые скобки в (*PD).YEAR необходимы, потому что операция указания члена

стуктуры старше, чем * . Обе операции, "->" и ".", ассоциируются слева направо, так что конструкции слева и справа зквивалентны

```
P->Q->MEMB          (P->Q)->MEMB
EMP.BIRTHDATE.MONTH          (EMP.BIRTHDATE).MONTH
```

Для полноты ниже приводится другая функция, MONTH_DAY, переписанная с использованием 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;
      \)
```

Операции работы со структурами "->" и "." наряду со () для списка аргументов и [] для индексов находятся на самом верху иерархии страшинства операций и, следовательно, связываются очень крепко. Если, например, имеется описание

```
  STRUCT \(
      INT X;
      INT *Y;
  \) *P;
```

то выражение

```
++P->X
```

увеличивает x, а не p, так как оно эквивалентно выражению

++(P->x). Для изменения порядка выполнения операций можно

использовать круглые скобки: (++P)->x увеличивает P до доступа к x, а (P++)->X увеличивает P после. (круглые скобки в последнем случае необязательны. Почему?)

Совершенно аналогично *P->Y извлекает то, на что указывает Y; *P->Y++ увеличивает Y после обработки того, на что он указывает (точно так же, как и *S++); (*P->Y)++ увеличивает то, на что указывает Y; *P++->Y увеличивает P после выборки того, на что указывает Y.

6.3. Массивы структур

Структуры особенно подходят для управления массивами связанных переменных. Рассмотрим, например, программу подсчета числа вхождений каждого ключевого слова языка "C". Нам нужен массив символьных строк для хранения имен и массив целых для подсчета. одна из возможностей состоит в использовании двух параллельных массивов KEYWORD и KEYCOUNT:

```
CHAR *KEYWORD [NKEYS];
INT  KEYCOUNT [NKEYS];
```

Но сам факт, что массивы параллельны, указывает на возможность другой организации. Каждое ключевое слово здесь по существу является парой:

```
CHAR *KEYWORD;
INT  KEYCOUNT;
```

и, следовательно, имеется массив пар. Описание структуры

```
STRUCT KEY  \(
    CHAR *KEYWORD;
    INT KEYCOUNT; \) KEYTAB [NKEYS];
```

оперделяет массив KEYTAB структур такого типа и отводит для

них память. Каждый элемент массива является структурой. Это

можно было бы записать и так:

```
STRUCT KEY  \(
    CHAR *KEYWORD;
    INT KEYCOUNT; \);
STRUCT KEY KEYTAB [NKEYS];
```

Так как структура KEYTAB фактически содержит постоянный набор имен, то легче всего инициализировать ее один раз и для всех членов при определении.

Инициализация структур вполне аналогична предыдущим инициализациям — за определением следует заключенный в фигурные скобки список инициализаторов:

```
STRUCT KEY  \(
    CHAR *KEYWORD;
    INT KEYCOUNT; \) KEYTAB[] =\(
    "BREAK", 0,
    "CASE", 0,
    "CHAR", 0,
    "CONTINUE", 0,
    "DEFAULT", 0,
    /* ... */
    "UNSIGNED", 0,
    "WHILE", 0
\);
```

Инициализаторы перечисляются парами соответственно членам структуры. Было бы более точно заключать в фигурные скобки инициализаторы для каждой "строки" или структуры следующим образом:

```
\( "BREAK", 0 \),
\( "CASE", 0 \),
. . .
```

Но когда инициализаторы являются простыми переменными или символьными строками и все они присутствуют, то во внутренних фигурных скобках нет необходимости. Как обычно, компилятор сам вычислит число элементов массива KEYTAB, если инициализаторы присутствуют, а скобки [] оставлены пустыми.

Программа подсчета ключевых слов начинается с определения массива KEYTAB. ведущая программа читает свой файл ввода, последовательно обращаясь к функции GETWORD, которая извлекает из ввода по одному слову за обращение. Каждое слово ищется в массиве KEYTAB с помощью варианта функции бинарного поиска, написанной нами в главе 3. (Конечно, чтобы эта функция работала, список ключевых слов должен быть расположен в порядке возрастания).

```
#DEFINE         MAXWORD   20

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);
    \)
```

Мы вскоре приведем функцию GETWORD; пока достаточно сказать, что она возвращает LETTER каждый раз, как она находит слово, и копирует это слово в свой первый аргумент.

Величина NKEYS — это количество ключевых слов в массиве KEYTAB . Хотя мы можем сосчитать это число вручную, гораздо легче и надежнее поручить это машине, особенно в том случае, если список ключевых слов подвержен изменениям. Одной из возможностей было бы закончить список инициализаторов указанием на нуль и затем пройти в цикле сквозь массив KEYTAB, пока не найдется конец.

Но, поскольку размер этого массива полностью определен к моменту компиляции, здесь имеется более простая возможность. Число элементов просто есть

SIZE OF KEYTAB / SIZE OF STRUCT KEY

дело в том, что в языке "С" предусмотрена унарная операция

SIZEOF, выполняемая во время компиляции, которая позволяет вычислить размер любого объекта. Выражение

SIZEOF(OBJECT)

выдает целое, равное размеру указанного объекта. (Размер определяется в неспецифицированных единицах, называемых "байтами", которые имеют тот же размер, что и переменные типа CHAR). Объект может быть фактической переменной, массивом и структурой, или именем основного типа, как INT или DOUBLE, или именем производного типа, как структура. В нашем случае число ключевых слов равно размеру массива, деленному на размер одного элемента массива. Это вычисление используется в утверждении #DEFINE для установления значения NKEYS:

#DEFINE NKEYS (SIZEOF(KEYTAB) / SIZEOF(STRUCT KEY))

Теперь перейдем к функции GETWORD. Мы фактически написали более общий вариант функции GETWORD, чем необходимо для этой программы, но он не на много более сложен. Функция GETWORD возвращает следующее "слово" из ввода, где словом считается либо строка букв и цифр, начинающихся с буквы, либо отдельный символ. Тип объекта возвращается в качетве значения функции; это — LETTER, если найдено слово, EOF для конца файла и сам символ, если он не буквенный.

```
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);
```

```
\)
```

Функция GETWORD использует функции GETCH и UNGETCH, которые мы написали в главе 4: когда набор алфавитных символов прерывается, функция GETWORD получает один лишний символ. В результате вызова UNGETCH этот символ помещается назад во ввод для следующего обращения.

Функция GETWORD обращается к функции TYPE для определения типа каждого отдельного символа из файла ввода. Вот вариант, справедливый только для алфавита ASCII.

```
TYPE(C)  /* RETURN TYPE OF ASCII CHARACTER */
INT C;
  \(
IF (C>= 'A' && C<= 'Z' \!\! C>= 'A' && C<= 'Z')
        RETURN(LETTER);
ELSE IF (C>= '0' && C<= '9')
        RETURN(DIGIT);
ELSE
        RETURN(C);
\)
```

Символические константы LETTER и DIGIT могут иметь любые значения, лишь бы они не вступали в конфликт с символами, отличными от буквенно-цифровых, и с EOF; очевидно возможен следующий выбор

```
#DEFINE   LETTER   'A'
#DEFINE   DIGIT    '0'
```

функция GETWORD могла бы работать быстрее, если бы обращения к функции TYPE были заменены обращениями к соответствующему массиву TYPE[ ]. В стандартной библиотеке языка "С" предусмотрены макросы ISALPHA и ISDIGIT, действующие необходимым образом.

Упражнение 6-1.

Сделайте такую модификацию функции GETWORD и оцените, как изменится скорость работы программы.

Упражнение 6-2.

Напишите вариант функции TYPE, не зависящий от конкретного наборасимволов.

Упражнение 6-3.

Напишите вариант программы подсчета ключевых слов, который бы не учитывал появления этих слов в заключенных в кавычки строках.

6.4. Указатели на структуры

Чтобы проиллюстрировать некоторые соображения, связанные с использованием указателей и массивов структур, давайте снова составим программу подсчета ключевых строк, используя на этот раз указатели, а не индексы массивов.

Внешнее описание массива KEYTAB не нужно изменять, но функции MAIN и BINARY требуют модификации.

```
    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);
```

\)

Здесь имеется несколько моментов, которые стоит отметить. Во-первых, описание функции BINARI должно указывать, что она возвращает указатель на структуру типа KEY, а не на целое; это объявляется как в функции MAIN, так и в BINARY. Если функция BINARI находит слово, то она возвращает указатель на него; если же нет, она возвращает NULL.

Во-вторых, все обращения к элементам массива KEYTAB осуществляются через указатели. Это влечет за собой одно существенное изменение в функции BINARY: средний элемент больше нельзя вычислять просто по формуле

```
MID = (LOW + HIGH) / 2
```

потому что сложение двух указателей не дает какого-нибудь

полезного результата (даже после деления на 2) и в действительности является незаконным. эту формулу надо заменить на

```
MID = LOW + (HIGH-LOW) / 2
```

в результате которой MID становится указателем на элемент, расположенный посередине между LOW и HIGH.

Вам также следует разобраться в инициализации LOW и HIGH. указатель можно инициализировать адресом ранее определенного объекта; именно как мы здесь и поступили.

В функции MAIN мы написали

```
FOR (P=KEYTAB; P < KEYTAB + NKEYS; P++)
```

Если P является указателем структуры, то любая арифметика с

P учитывает фактический размер данной структуры, так что P++ увеличивает P на нужную величину, в результате чего P указывает на следующий элемент массива структур. Но не считайте, что размер структуры равен сумме размеров ее членов, — из-за требований выравнивания для различных объектов в структуре могут возникать "дыры".

И, наконец, несколько второстепенный вопрос о форме записи программы. Если возвращаемая функцией величина имеет тип, как, например, в

```
STRUCT KEY *BINARY(WORD, TAB, N)
```

То может оказаться, что имя функции трудно выделить среди текста. В связи с этим иногда используется другой стиль записи:

```
STRUCT KEY *
BINARY(WORD, TAB, N)
```

Это главным образом дело вкуса; выберите ту форму, которая вам нравится, и придерживайтесь ее.

6.5. Структуры, ссылающиеся на себя.

Предположим, что нам надо справиться с более общей задачей, состоящей в подсчете числа появлений всех слов в некотором файле ввода. Так как список слов заранее не известен, мы не можем их упорядочить удобным образом и использовать бинарный поиск. Мы даже не можем осуществлять последовательный просмотр при поступлении каждого слова, с тем чтобы установить, не встречалось ли оно ранее; такая программа будет работать вечно. (Более точно, ожидаемое время работы растет как квадрат числа вводимых слов). Как же нам организовать программу, чтобы справиться со списком произвольных слов?

Одно из решений состоит в том, чтобы все время хранить массив поступающих до сих пор слов в упорядоченном виде, помещая каждое слово в нужное место по мере их поступления. ОДнако это не следует делать, перемещая слова в линейном массиве, — это также потребует слишком много времени. Вместо этого мы используем структуру данных, называемую доичным деревом.

Каждому новому слову соответствует один "узел" дерева; каждый узел содержит:

указатель текста слова

счетчик числа появлений

указатель узла левого потомка

указатель узла правого потомка

Никакой узел не может иметь более двух детей; возможно отсутсвие детей или наличие только одного потомка.

Узлы создаются таким образом, что левое поддерево каждого узла содержит только те слова, которые меньше слова в этом узле, а правое поддерево только те слова, которые больше. Чтобы определить, находится ли новое слово уже в дереве, начинают с корня и сравнивают новое слово со словом, хранящимся в этом узле. Если слова совпадают, то вопрос решается утвердительно. Если новое слово меньше слова в дереве, то переходят к рассмотрению левого потомка; в противном случае исследуется правый потомок. Если в нужном направлении потомок отсутствует, то значит новое слово не находится в дереве и место этого недостающего потомка как раз и является местом, куда следует поместить новое слово. Поскольку поиск из любого узла приводит к поиску одного из его потомков, то сам процесс поиска по существу является рекурсивным. В соответствии с этим наиболее естественно использовать рекурсивные процедуры ввода и вывода.

Возвращаясь назад к описанию узла, ясно, что это будет структура с четырьмя компонентами:

```
STRUCT TNODE \( /* THE BASIC NODE */
    CHAR *WORD; /* POINTS TO THE TEXT */
    INT   COUNT; /* NUMBER OF OCCURRENCES */
    STRUCT TNODE *LEFT; /* LEFT CHILD */
    STRUCT TNODE *RIGHT; /* RIGHT CHILD */
\);
```

Это "рекурсивное" описание узла может показаться рискованным, но на самом деле оно вполне корректно. Структура не имеет права содержать ссылку на саму себя, но

```
STRUCT TNODE *LEFT;
```

описывает LEFT как указатель на узел, а не как сам узел.

Текст самой программы оказывается удивительно маленьким, если, конечно, иметь в распоряжении набор написанных нами ранее процедур, обеспечивающих нужные действия. Мы имеем в виду функцию GETWORD для извлечения каждого слова из файла ввода и функцию ALLOC для выделения места для хранения слов.

Ведущая программа просто считывает слова с помощью функции GETWORD и помещает их в дерево, используя функцию TREE.

```
#DEFINE   MAXWORD   20
MAIN()          /* WORD FREGUENCY COUNT */
\(
      STRUCT TNODE *ROOT, *TREE();
      CHAR WORD[MAXWORD];
      INT   T;
      ROOT = NULL;
      WHILE ((T = GETWORD(WORD, MAXWORD)) \! = EOF)
          IF (T == LETTER)
                ROOT = TREE(ROOT, WORD);
      TREEPRINT(ROOT);
\)
```

Функция TREE сама по себе проста. Слово передается функцией MAIN к верхнему уровню (корню) дерева. На каждом этапе это слово сравнивается со словом, уже хранящимся в этом узле, и с помощью рекурсивного обращения к TREE просачивается вниз либо к левому, либо к правому поддереву. В конце концов это слово либо совпадает с каким-то словом, уже находящимся в дереве (в этом случае счетчик увеличивается на единицу), либо программа натолкнется на нулевой указатель, свидетельствующий о необходимости создания и добавления к дереву нового узла. В случае создания нового узла функция TREE возвращает указатель этого узла, который помещается в родительский узел.

```
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);
    \)
```

Память для нового узла выделяется функцией TALLOC, являющейся адаптацией для данного случая функции ALLOC, написанной нами ранее. Она возвращает указатель свободного пространства, пригодного для хранения нового узла дерева. (Мы вскоре обсудим это подробнее). Новое слово копируется функцией STRSAVE в скрытое место, счетчик инициализируется единицей, и указатели обоих потомков полагаются равными нулю. Эта часть программы выполняется только при добавлении нового узла к ребру дерева. Мы здесь опустили проверку на ошибки возвращаемых функций STRSAVE и TALLOC значений (что неразумно для практически работающей программы).

Функция TREEPRINT печатает дерево, начиная с левого поддерева; в каждом узле сначала печатается левое поддерево (все слова, которые младше этого слова), затем само слово, а затем правое поддерево (все слова, которые старше). Если вы неуверенно оперируете с рекурсией, нарисуйте дерево сами и напечатайте его с помощью функции TREEPRINT ; это одна из наиболее ясных рекурсивных процедур, которую можно найти.

```
TREEPRINT (P) /* PRINT TREE  P  RECURSIVELY */
STRUCT TNODE *P;
\(
    IF (P != NULL)              \(
        TREEPRINT (P->LEFT);
        PRINTF("%4D %S\N", P->COUNT, P->WORD);
        TREEPRINT (P->RIGHT);
    \)
\)
```

Практическое замечание: если дерево становится "несбалансированным" из-за того, что слова поступают не в случайном порядке, то время работы программы может расти слишком быстро. В худшем случае, когда поступающие слова уже упорядочены, настоящая программа осуществляет дорогостоящую имитацию линейного

поиска. Существуют различные обобщения двоичного дерева, особенно 2-3 деревья и AVL деревья, которые не ведут себя так "в худших случаях", но мы не будем здесь на них останавливаться.

Прежде чем расстаться с этим примером, уместно сделать небольшое отступление в связи с вопросом о распределении памяти. Ясно, что в программе желательно иметь только один распределитель памяти, даже если ему приходится размещать различные виды объектов. Но если мы хотим использовать один распределитель памяти для обработки запросов на выделение памяти для указателей на переменные типа CHAR и для указателей на STRUCT TNODE, то при этом возникают два вопроса. Первый: как выполнить то существующее на большинстве реальных машин ограничение, что объекты определенных типов должны удовлетворять требованиям выравнивания (например, часто целые должны размещаться в четных адресах)? Второй: как организовать описания, чтобы справиться с тем, что функция ALLOC должна возвращать различные виды указателей?

Вообще говоря, требования выравнивания легко выполнить за счет выделения некоторого лишнего пространства, просто обеспечив то, чтобы распределитель памяти всегда возвращал указатель, удовлетворяющий всем ограничениям выравнивания. Например, на PDP-11 достаточно, чтобы функция ALLOC всегда возвращала четный указатель, поскольку в четный адрес можно поместить любой тип объекта. единственный расход при этом — лишний символ при запросе на нечетную длину. Аналогичные действия предпринимаются на других машинах. Таким образом, реализация ALLOC может не оказаться переносимой, но ее использование будет переносимым. Функция ALLOC из главы 5 не предусматривает никакого определенного выравнивания; в главе 8 мы продемонстрируем, как правильно выполнить эту задачу.

Вопрос описания типа функции ALLOC является мучительным для любого языка, который серьезно относится к проверке типов. Лучший способ в языке "С" — объявить, что ALLOC возвращает указатель на переменную типа CHAR, а затем явно преобразовать этот указатель к желаемому типу с помощью операции перевода типов. Таким образом, если описать P в виде

CHAR *P; то

(STRUCT TNODE *) P

преобразует его в выражениях в указатель на структуру типа

TNODE . Следовательно, функцию TALLOC можно записать в виде:

```
STRUCT TNODE *TALLOC()
\(
    CHAR *ALLOC();

        RETURN ((STRUCT TNODE *) ALLOC(SIZEOF(STRUCT TNODE)));
\)
```

это более чем достаточно для работающих в настоящее время

компиляторов, но это и самый безопасный путь с учетом будущего.

Упражнение 6-4.

Напишите программу, которая читает "С"-программу и печатает в алфавитном порядке каждую группу имен переменных, которые совпадают в первых семи символах, но отличаются где-то дальше. (Сделайте так, чтобы 7 было параметром).

Упражнение 6-5.

Напишите программу выдачи перекрестных ссылок, т.е. Программу, которая печатает список всех слов документа и для каждого из этих слов печатает список номеров строк, в которые это слово входит.

Упражнение 6-6.

Напишите программу, которая печатает слова из своего файла ввода, расположенные в порядке убывания частоты их появления. Перед каждым словом напечатайте число его появлений.

6.6. Поиск в таблице

Для иллюстрации дальнейших аспектов использования структур в этом разделе мы напишем программу, представляющую собой содержимое пакета поиска в таблице. Эта программа является типичным представителем подпрограмм управления символьными таблицами макропроцессора или компилятора. Рассмотрим, например, оператор #DEFINE языка "С". Когда встречается строка вида

```
#DEFINE YES          1
```

то имя YES и заменяющий текст 1 помещаются в таблицу. Позднее, когда имя YES появляется в операторе вида

```
INWORD = YES;
```

Оно должно быть замещено на 1.

Имеются две основные процедуры, которые управляют именами и заменяющими их текстами. Функция INSTALL(S,T) записывает имя S и заменяющий текст T в таблицу; здесь S и T просто символьные строки. Функция LOOKUP(S) ищет имя S в таблице и возвращает либо указатель того места, где это имя найдено, либо NULL, если этого имени в таблице не оказалось.

При этом используется поиск по алгоритму хеширования — поступающее имя преобразуется в маленькое положительное число, которое затем используется для индексации массива указателей. Элемент массива указывает на начало цепочных блоков, описывающих имена, которые имеют это значение хеширования. Если никакие имена при хешировании не получают этого значения, то элементом массива будет NULL.

Блоком цепи является структура, содержащая указатели на соответствующее имя, на заменяющий текст и на следующий блок в цепи. Нулевой указатель следующего блока служит признаком конца данной цепи.

```
STRUCT NLIST  \(  /* BASIC TABLE ENTRY */
        CHAR *NAME;
```

```
        CHAR *DEF;
        STRUCT NLIST *NEXT; /* NEXT ENTRY IN CHAIN */
\);
```

Массив указателей это просто

```
DEFINE        HASHSIZE          100
TATIC STRUCT NLIST *HASHTAB[HASHSIZE] /* POINTER TABLE */
```

Значение функции хеширования, используемой обеими функциями LOOKUP и INSTALL, получается просто как остаток от деления суммы символьных значений строки на размер массива. (Это не самый лучший возможный алгоритм, но его достоинство состоит в исключительной простоте).

```
HASH(S)   /* FORM HASH VALUE FOR STRING */
CHAR *S;
\(
  INT HASHVAL;

  FOR (HASHVAL = 0; *S != '\0'; )
        HASHVAL += *S++;
  RETURN(HASHVAL % HASHSIZE);
\)
```

В результате процесса хеширования выдается начальный индекс в массиве HASHTAB ; если данная строка может быть где-то найдена, то именно в цепи блоков, начало которой указано там. Поиск осуществляется функцией LOOKUP. Если функция LOOKUP находит, что данный элемент уже присутствует, то она возвращает указатель на него; если нет, то она возвращает 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 */
```

Функция INSTALL использует функцию LOOKUP для определения, не присутствует ли уже вводимое в данный момент имя; если это так, то новое определение должно вытеснить старое. В противном случае создается совершенно

новый элемент. Если по какой-либо причине для нового элемента больше нет места, то функция INSTALL возвращает 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);
    \)
```

Функция STRSAVE просто копирует строку, указанную в качестве аргумента, в место хранения, полученное в результате обращения к функции ALLOC. Мы уже привели эту функцию в главе 5. Так как обращение к функции ALLOC и FREE могут происходить в любом порядке и в связи с проблемой выравнивания, простой вариант функции ALLOC из главы 5 нам больше не подходит; смотрите главы 7 и 8.

Упражнение 6-7.

Напишите процедуру, которая будет удалять имя и определение из таблицы, управляемой функциями LOOKUP и INSTALL.

Упражнение 6-8.

Разработайте простую, основанную на функциях этого раздела, версию процессора для обработки конструкций #DEFINE, пригодную для использования с "C"-программами. Вам могут также оказаться полезными функции GETCHAR и UNGETCH.

6.7. Поля

Когда вопрос экономии памяти становится очень существенным, то может оказаться необходимым помещать в одно машинное слово несколько различных объектов; одно из особенно распространенных употреблений — набор однобитовых признаков в применениях, подобных символьным таблицам компилятора. внешне обусловленные форматы данных, такие как интерфейсы аппаратных средств также зачастую предполагают возможность получения слова по частям.

Представьте себе фрагмент компилятора, который работает с символьной таблицей. С каждым идентификатором программы связана определенная информация, например, является он или нет ключевым словом, является ли он или нет внешним и/или статическим и т.д. Самый компактный способ закодировать такую информацию — поместить набор однобитовых признаков в отдельную переменную типа CHAR или INT.

Обычный способ, которым это делается, состоит в определении набора "масок", отвечающих соответствущим битовым позициям, как в

```
#DEFINE   KEYWORD   01
#DEFINE   EXTERNAL  02
#DEFINE   STATIC            04
```

(числа должны быть степенями двойки). Тогда обработка битов

сведется к "жонглированию битами" с помощью операций сдвига,

маскирования и дополнения, описанных нами в главе 2.

Некоторые часто встречающиеся идиомы:

```
FLAGS \!= EXTERNAL  \! STATIC;
```

включает биты EXTERNAL и STATIC в FLAGS, в то время как

```
FLAGS &= \^(eXTERNAL \! STATIC);
```

их выключает, а

```
IF ((FLAGS & (EXTERNAL \! STATIC)) == 0) ...
```

истинно, если оба бита выключены.

Хотя этими идиомами легко овладеть, язык "С" в качестве альтернативы предлагает возможность определения и обработки полей внутри слова непосредственно, а не посредством побитовых логических операций. Поле — это набор смежных битов внутри одной переменной типа INT. Синтаксис определения и обработки полей основывается на структурах. Например, символьную таблицу конструкций #DEFINE, приведенную выше, можно бы было заменить определением трех полей:

```
    STRUCT  \(
UNSIGNED IS_KEYWORD : 1;
UNSIGNED IS_EXTERN  : 1;
```

```
    UNSIGNED IS_STATIC  : 1;
    \)          FLAGS;
```

Здесь определяется переменная с именем FLAGS, которая содержит три 1-битовых поля. Следующее за двоеточием число задает ширину поля в битах. Поля описаны как UNSIGNED, чтобы подчеркнуть, что они действительно будут величинами без знака.

На отдельные поля можно ссылаться, как FLAGS.IS_STATIE, FLAGS. IS_EXTERN, FLAGS.IS_KEYWORD И т.д., то есть точно так же, как на другие члены структуры. Поля ведут себя подобно небольшим целым без знака и могут участвовать в арифметических выражениях точно так же, как и другие целые. Таким образом, предыдущие примеры более естественно переписать так:

```
FLAGS.IS_EXTERN = FLAGS.IS_STATIC = 1;
```

для включения битов;

```
FLAGS.IS_EXTERN = FLAGS.IS_STATIC = 0;
```

для выключения битов;

```
IF (FLAGS.IS_EXTERN == 0 &&FLAGS.IS_STATIC == 0)...
```

для их проверки.

Поле не может перекрывать границу INT; если указанная ширина такова, что это должно случиться, то поле выравнивается по границе следующего INT. Полям можно не присваивать имена; неименованные поля (только двоеточие и ширина) используются для заполнения свободного места. Чтобы вынудить выравнивание на границу следующего INT, можно использовать специальную ширину 0.

При работе с полями имеется ряд моментов, на которые следует обратить внимание. По-видимому наиболее существенным является то, что отражая природу различных аппаратных средств, распределение полей на некоторых машинах осуществляется слева направо, а на некоторых справа налево. Это означает, что хотя поля очень полезны для работы с внутренне определенными структурами данных, при разделении внешне определяемых данных следует тщательно рассматривать вопрос о том, какой конец поступает первым.

Другие ограничения, которые следует иметь в виду: поля не имеют знака; они могут храниться только в переменных типа INT (или, что эквивалентно, типа UNSIGNED); они не являются массивами; они не имеют адресов, так что к ним не применима операция &.

6.8. Объединения

Объединения — это переменная, которая в различные моменты времени может содержать объекты разных типов и размеров, причем компилятор берет на себя отслеживание размера и требований выравнивания. Объединения представляют возможность работать с различными видами данных в одной области памяти, не вводя в программу никакой машинно-зависимой информации.

В качестве примера, снова из символьной таблицы компилятора, предположим, что константы могут быть типа INT, FLOAT или быть указателями на символы.

значение каждой конкретной константы должно храниться в переменной соотвествующего типа, но все же для управления таблицей самым удобным было бы, если это значение занимало бы один и тот же объем памяти и хранилось в том же самом месте независимо от его типа. это и является назначением объединения — выделить отдельную переменную, в которой можно законно хранить любую одну из переменных нескольких типов. Как и в случае полей, синтаксис основывается на структурах.

```
UNION U_TAG \(
INT IVAL;
FLOAT FVAL;
CHAR *PVAL;
\) UVAL;
```

Переменная UVAL будет иметь достаточно большой размер,чтобы хранить наибольший из трех типов, независимо от машины, на которой осуществляется компиляция, — программа не будет зависить от характеристик аппаратных средств. Любой из этих трех типов может быть присвоен UVAR и затем использован в выражениях, пока такое использование совместимо: извлекаемый тип должен совпадать с последним помещенным типом. Дело программиста — следить за тем, какой тип хранится в объединении в данный момент; если что-либо хранится как один тип, а извлекается как другой, то результаты будут зависеть от используемой машины.

Синтаксически доступ к членам объединения осуществляется следующим образом:

имя объединения.член

или

указатель объединения ->член

то есть точно так же, как и в случае структур. если для отслеживания типа, хранимого в данный момент в UVAL, используется переменная UTYPE, то можно встретить такой участок программы:

```
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);
```

Объединения могут появляться внутри структур и массивов и наоборот. Запись для обращения к члену объединения в структуре (или наоборот) совершенно идентична той, которая используется во вложенных структурах. например, в массиве структур, определенным следующим образом

```
STRUCT \(
CHAR *NAME;
INT FLAGS;
INT UTYPE;
UNION \(
INT IVAL;
FLOAT FVAL;
CHAR *PVAL;
\) UVAL;
  \) SYMTAB[NSYM];
```

на переменную IVAL можно сослаться как

```
SYMTAB[I].UVAL.IVAL
```

а на первый символ строки PVAL как

```
*SYMTAB[I].UVAL.PVAL
```

В сущности объединение является структурой, в которой все члены имеют нулевое смещение. Сама структура достаточно велика, чтобы хранить "самый широкий" член, и выравнивание пригодно для всех типов, входящих в объединение. Как и в случае структур, единственными операциями, которые в настоящее время можно проводить с объединениями, являются доступ к

члену и извлечение адреса; объединения не могут быть присвоены, переданы функциям или возвращены ими. указатели объединений можно использовать в точно такой же манере, как и указатели структур.

Программа распределения памяти, приводимая в главе 8, показывает, как можно использовать объединение, чтобы сделать некоторую переменную выровненной по определенному виду границы памяти.

6.9. Определение типа

В языке "С" предусмотрена возможность, называемая TYPEDEF для введения новых имен для типов данных. Например, описание

```
TYPEDEF INT LENGTH;
```

делает имя LENGTH синонимом для INT. "Тип" LENGTH может быть использован в описаниях, переводов типов и т.д. Точно таким

же образом, как и тип INT:

```
LENGTH   LEN, MAXLEN;
LENGTH   *LENGTHS[];
```

Аналогично описанию

```
TYPEDEF CHAR *STRING;
```

делает STRING синонимом для CHAR*, то есть для указателя на символы, что затем можно использовать в описаниях вида

```
STRING P, LINEPTR[LINES], ALLOC();
```

Обратите внимание, что объявляемый в конструкции TYPEDEF тип появляется в позиции имени переменной, а не сразу за словом TYPEDEF. Синтаксически конструкция TYPEDEF подобна описаниям класса памяти EXTERN, STATIC и т. Д. мы также использовали прописные буквы, чтобы яснее выделить имена.

В качестве более сложного примера мы используем конструкцию TYPEDEF для описания узлов дерева, рассмотренных ранее в этой главе:

```
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 результате получаем два новых ключевых слова: TREENODE (структура) и TREEPTR (указатель на структуру). Тогда функцию TALLOC можно записать в виде

```
                TREEPTR TALLOC()
\(
    CHAR *ALLOC();
    RETURN((TREEPTR) ALLOC(SIZEOF(TREENODE)));
\)
```

Необходимо подчеркнуть, что описание TYPEDEF не приводит к созданию нового в каком-либо смысле типа; оно только добавляет новое имя для некоторого существующего типа. при этом не возникает и никакой новой семантики: описанные таким способом переменные обладают точно теми же свойствами, что и переменные, описанные явным образом. По существу конструкция TYPEDEF сходна с #DEFINE за исключением того, что она интерпретируется компилятором и потому может осуществлять подстановки текста, которые выходят за пределы возможностей макропроцессора языка "С". Например,

```
TYPEDEF INT (*PFI) ();
```

создает тип PFI для "указателя функции, возвращающей значение типа INT", который затем можно было бы использовать в программе сортировки из главы 5 в контексте вида

```
PFI STRCMP, NUMCMP, SWAP;
```

Имеются две основные причины применения описаний TYPEDEF. Первая причина связана с параметризацией программы, чтобы облегчить решение проблемы переносимости. Если для типов данных, которые могут быть машинно-зависимыми, использовать описание TYPEDEF, то при переносе программы на другую машину придется изменить только эти описания. Одна из типичных ситуаций состоит в использовании определяемых с помощью TYPEDEF имен для различных целых величин и в последующем подходящем выборе типов SHORT, INT и LONG для каждой имеющейся машины.

Второе назначение TYPEDEF состоит в обеспечении лучшей документации для программы — тип с именем TREEPTR может оказаться более удобным для восприятия, чем тип, который описан только как указатель сложной структуры.

И наконец, всегда существует вероятность, что в будущем компилятор или некоторая другая программа, такая как LINT, сможет использовать содержащуюся в описаниях TYPEDEF информацию для проведения некоторой дополнительной проверки программы.

7. Ввод и вывод

Средства ввода/вывода не являются составной частью языка "с", так что мы не выделяли их в нашем предыдущем изложении. Однако реальные программы взаимодействуют со своей окружающей средой гораздо более сложным образом, чем мы видели до сих пор. В этой главе будет описана "стандартная библиотека ввода/вывода", то есть набор функций, разработанных для обеспечения стандартной системы ввода/вывода для "с"- программ. Эти функции предназначены для удобства программного интерфейса, и все же отражают только те операции, которые могут быть обеспечены на большинстве современных операционных систем. Процедуры достаточно эффективны для того, чтобы пользователи редко чувствовали необходимость обойти их "ради эффективности", как бы ни была важна конкретная задача. И, наконец, эти процедуры задуманы быть "переносимыми" в том смысле, что они должны существовать в совместимом виде на любой системе, где имеется язык "с", и что программы, которые ограничивают свои взаимодействия с системой возможностями, предоставляемыми стандартной библиотекой, можно будет переносить с одной системы на другую по существу без изменений.

Мы здесь не будем пытаться описать всю библиотеку ввода/вывода; мы более заинтересованы в том, чтобы продемонстрировать сущность написания "с"-программ, которые взаимодействуют со своей операционной средой.

7.1. Обращение к стандартной библиотеке

Каждый исходный файл, который обращается к функции из стандартной библиотеки, должен вблизи начала содержать строку

УДК **004.413**

Д. Зербіно, Ю. Цимбал
Національний університет “Львівська політехніка”,
кафедра автоматизованих систем управління

ТЕНДЕНЦІЇ НЕВИЗНАЧЕНОСТІ У МОВАХ ПРОГРАМУВАННЯ

Розглянуто підходи до створення мов програмування, які призначені для розв’язання інтелектуальних задач. Введено різні типи невизначеностей та проаналізовано їхню роль в підвищенні рівня інтелектуальності та зручності мови програмування.

In the article the approaches to the creation of programming languages that are intended for solving the intellectual problems have been studied. Different types of uncertainity have been introduced and their role in increasing the intellectual level and the convenience of programming languages has been presented.

Вступ

Реальне життя характерне тим, що необхідно розв’язувати велику кількість неточно поставлених задач. У таких задачах припускається певна невизначеність вхідних даних. Невизначеність може бути як у постановці задачі, так у і засобах її вирішення. Під невизначеністю даних в програмуванні будемо розуміти можливість застосування до них апарату теорії нечітких множин. Під невизначеністю алгоритмів – можливість уточнення або доповнення алгоритму під час роботи програми, а також послідовного вибору його альтернативних варіантів (з можливістю повернення стану програми до точки вибору). Основна мета введення невизначеності в програмуванні – це спрощення розуміння програми. Нечіткість мови програмування не означатиме, що вона не містить чітких операцій. Просто чітка операція розглядається як дія з ймовірністю 100%, а нечітку дію під час виконання програми в разі необхідності можна замінити іншою. Для програмування невизначеність дає певну зручність, тому що програміст, коли не знає як саме скласти певну програму, може ввести в неї нечіткі поняття. В процесі програмування ці поняття будуть уточнюватись і структуризуватись. Отже, невизначеність стає частиною власне технології програмування.

Історія проблеми

Тривалий час думка про можливість розроблення мов програмування з використанням невизначеностей була джерелом різного роду жартів. Наприклад, висувалася “ідея” створення мови C+– (“більш або менш C” – за аналогією з мовою C++). Зокрема, пропонувалося перевантажити операції “>” та “<” як “краще за” та “гірше за”, а “>>” та “<<” як “значно краще за” та “забудьте про це !” [1]. Тобто, зверталась увага на недостатньо очевидні особливості синтаксису мови.

Із розвитком нечіткої логіки та теорії нечітких множин на початку 1990-х років розпочалася розробка мов програмування на їх основі. Серед таких мов можна виділити fuzzy programming language (fpl) фірми togai infralogic [2] та dcu fuzzy compiler, розроблений в університеті м. Дубліна (ірландія) [3]. Мови нечіткого програмування надають можливість оголосити та ініціалізувати вхідні змінні, описати відповідні нечіткі множини (або функції належності) для їхніх значень і утворити базу нечітких знань та ввести правила їх опрацювання. Як ми бачимо, цей напрямок тісно пов’язаний з класичною теорією баз знань і менше стосується власне програмування.

Серед новітніх підходів до розвитку мов програмування зазначимо підхід, запропонований засновником нечіткої логіки л. заде [4]. Зокрема, запропоновано при створенні мови максимально наблизити її складові (зокрема значення змінних та операції над ними) до конструкцій, вживаних у природній мові людини. наприклад,

“якщо *(X* є “дуж е малим*”)* – т о *(Y* є “не дуж е великим*”),*

або

якщо *(X* є "великим*")* – т о *(Y* є "малим*").*

тоді можна поставити запитання на зразок:

"яким є значення *X*, якщо *Y* є "великим*"* ?

Для опису таких задач уведено концепцію узагальненої мови обмежень (generalized constraint language, gcl). Мови, які можна розробити на основі цієї концепції, потенційно нададуть більше інтелектуальності у представленні задач та їх подальшому вирішенні.

Усі ці підходи у той чи інший спосіб спрямовані на представлення фактів, хоча і в нечіткій формі. Але, на наш погляд, для програмування важливим є вибір альтернативи у програмі. нечітка альтернатива надає можливість програмі самостійно вибрати шлях виконання (послідовність наступних операцій). Залежно від результатів обчислень за певною альтернативою програма зможе повернутись до вибору іншої альтернативи. Тобто, вказаний вище приклад можна було б переформулювати так:

якщо *(X* є "великим*")* – т о *(Y* є "малим*"* або *Z* є "середнім*").*

Отже, невизначеність даних доповнюється певною невизначеністю коду.

Типи невизначеностей

Зі сказаного вище можна зробити висновок, що мови програмування з невизначеностями певною мірою сприятимуть розвитку теорії самоорганізації алгоритмів. Можна сказати і навпаки, що алгоритми, які самостійно організуються, стимулюватимуть розроблення таких мов програмування. Раніше розв'язати неточно поставлені задачі було неможливо, тому що будь-яка класична алгоритмічна мова працює з конкретною інформацією. Тепер з'являється певний рівень розуміння (як у розробників систем програмування, так і прикладних програмістів), який дає змогу розв'язувати неточно поставлені задачі.

будемо розрізняти такі типи невизначеностей.

- Перший тип невизначеності – це класичне розуміння нечіткої логіки [5], тобто, якщо ймовірність більша за задану межу, то обирається одне рішення (або певне значення), якщо менша – то інше. Якщо ймовірність певного значення дуже мала, то це не означатиме, що воно не буде братися до уваги. Просто насамперед для знаходження розв'язку розглядатимуть ймовірніші варіанти.
- Другий тип – це невизначеність самої мови програмування, тобто, один і той самий текст програми можна інтерпретувати по-різному. Наприклад, один і той самий оператор "приєднат и" можна трактувати по-різному залежно від контексту, до якого належить цей оператор. Відповідно до свого рівня інтелектуальності система програмування може правильно інтерпретувати цей текст чи неправильно (з погляду програміста).
- Третій тип невизначеності полягає у тому, що невідомо, коли програма повинна припинити пошук результату, тобто сама прийняти рішення про те, що цей напрямок перебору варіантів є безперспективним для пошуку і зробити так званий "бектрекінг".

Зв'язок невизначеностей з концептуальністю

Концептуальність потрібна для того, щоб розібратись в "океані" невизначеностей, виділити структуру задачі, відокремити головні дані від другорядних [6]. Концептуальність необхідно розглядати одночасно з невизначеністю ще і тому, що вона зв'язана з різноманітністю інтерпретації понять, що введені у програмах, або з інтерпретацією самого тексту програми. Невизначеність також стосується і структури задачі, і того, які дані вважати головними, а які – другорядними.

Непоганим прикладом концептуальності в програмуванні є мова логічного програмування prolog, де спочатку формулюються факти, задаються правила отримання нових даних, далі дається постановка задачі, формулюється мета. Невизначеним залишається лише рішення задачі, яке система знаходить сама. деякі автори намагалися ввести нечіткість в логічне програмування [2, 3], але поширеними їхні системи не стали.

Програма в такій системі складається з набору правил: якщо деякий об'єкт має деякі властивості, то до нього можуть бути застосовані певні дії, причому кожна дія має певну ймовірність. Наприклад, дія номер 1 може бути застосована з ймовірністю p1, дія номер 2 – з ймовірністю p2. Все, що залишилося: p3 = 1-(p1+p2) – ймовірність того, що до об'єкта не може бути застосована жодна дія.

Концептуальні мови програмування необхідні в такій сфері, де не справляються звичайні алгоритмічні мови програмування – в області нечітко поставлених інтелектуальних задач.

Підвищення інтелектуальності систем

Один з основних параметрів інтелектуальних систем – міра інтелектуальності – визначається можливістю роботи з нечіткою інформацією. Інтелектуальність системи полягає у тому, що вона сама доповнює та інтерпретує ту інформацію, котра вважається нечіткою чи неповною, тобто, система на свій розсуд додає іншу інформацію, спираючись на власний досвід. Коефіцієнт якості інтелектуальності такої системи залежить від того, наскільки точно вона вгадала те, чого хоче користувач. Є суттєва різниця в тому, чи система сама вирішує, які треба брати дані для обчислень, чи вона запрограмована на вибір певних даних. системи першого типу, як правило, інтелектуальні, другого – ні.

З невизначеністю близько зв'язані деякі допущення. для виконання певних дій система повинна обумовлювати, що: 1) існують деякі дані; 2) існують певні рішення на основі цих даних; 3) для рішення необхідно зробити певні припущення. Тоді невизначеність усувається тим, що система, зробивши такі припущення, перевіряє кожен з варіантів рішення, пропонованих системою, і обирає один з них.

Отже, коли система сама вибирає дані, ми не знаємо заздалегідь, які саме дані вона вибере. У цьому полягає невизначеність і інтелектуальність системи.

Нечіткі операції

Однією з особливостей нечіткого програмування є можливість перенесення властивостей одних об'єктів на інші. Це може бути використано при пошуку рішень за аналогією. взагалі операція нечіткого пошуку об'єктів є складовою частиною нечіткого програмування. В результаті нечіткої операції можна отримати нечіткий результат, якому можна надати нечітку оцінку. Тобто, влаштовує нас даний результат чи ні можна сказати лише після певного порівняння з деякими іншими результатами. У структурі такої мови програмування можна виділити фільтри ситуацій та аналізатори різного рівня і типу: аналізатор обмежень, аналізатор близькості мети, аналізатор можливих дій і т.д. з кожним фільтром (аналізатором) пов'язаний еквівалентний пояснювальний текст (природною мовою). Фрагменти цього тексту можуть бути використані системою при виявленні помилок та їх роз'ясненні.

Розглянемо конкретний приклад – запис числа в масив. в алгоритмічних мовах це абсолютно чітка операція – якась комірка масиву отримує значення. В мовах програмування з невизначеностями деякий масив пов'язується з певним поняттям, що розбите на градації – комірки масиву. Тоді операція присвоєння виглядає як припущення, що деяка градація поняття отримує певне (неточне) значення, зміст якого буде зрозумілим в подальшому тексті програми. Також довільне присвоєння необхідно розглядати як наділення певного об'єкта певними властивостями. Однак, на відміну від класичних мов програмування, це присвоєння не є остаточним, і система може повернутися до цієї операції для вибору іншої альтернативи або для уточнення значення, яке було присвоєне.

Для нечіткого програмування дуже важливим є формулювання критеріїв – що саме вважати правильною програмою, а також – коли можна знайти рішення і коли їх шукати безперспективно. Дуже важливим є формулювання теоретичних можливостей системи, а також їх класифікація. Нечітке програмування дає змогу, по-перше, представити думку людини у вигляді програми для комп'ютера, тобто максимально їх зблизити; а по-друге, систематизувати власну логіку, або представити своє бачення певної проблеми.

Виправлення помилок у нечітких програмах

Всі поняття в мові програмування з невизначеностями створює сам програміст і далі з ними оперує. Отже, з появою невизначеності помилки в програмах стають нечіткими, тобто неможливо точно сказати, чи є в програмі помилки. тобто, виправлення помилок в таких мовах програмування полягає у поступовому збільшенні визначеності. При складанні нечітких програм можна використовувати позитивні та негативні приклади результатів, тоді як у чітких програмах використовуються лише позитивні приклади.

У системі можливі виправлення помилок лише в тому випадку, коли вона розвивається. тому під час програмування виникають дві серйозні пов'язані задачі – нарощування абстрактного понятійного рівня алгоритму, за яким працює програма, і виправлення помилок, які найчастіше пов'язані з неправильним трактуванням цих понять. Чим більший рівень абстракції програми, тим більшою є її невизначеність і тим більше понять вона охоплює. Певний рівень невизначеності в мові програмування якраз полегшить нарощування понятійного рівня та виправлення помилок, пов'язаних з неправильним трактуванням понять, які були введені в мові програмування.

Отже, програміст і програма діють як одне ціле – аналізуючи позитивні та негативні приклади, поступово уточнюють та структуризують ті поняття, які введені в програмі.

Висновки

Свого часу більшість програмістів вибрали алгоритмічні мови, відмовившись від використання декларативних мов програмування. Значною мірою це було викликано нечіткістю методів складання декларативних програм. Тому ми намагалися обґрунтувати необхідність розроблення мов програмування з врахуванням невизначеності, притаманної мисленню людини, а також зробили деякі припущення щодо розвитку таких мов. Звичайно, що практична реалізація таких мов програмування – це справа майбутнього. сьогодні необхідно зацікавити широкі маси програмістів "таємничою" невизначеністю і зробити її зручним інструментом під час розроблення програм.

1. Прат т Т., Зелковиц М. Языки программирования*:* разработ ка и реализация. – СПб.*:* Пит ер*, 2002. – 688* с. *2. Fuzzy Programming Language: http://www.ortech-engr.com/fuzzy/fpl.html. 3. Yan J., Ryan M., Power J. Using Fuzzy Logic: Towards Intelligent Systems. – Prentice-Hall, 1995. 4. Zadeh L.A. Toward a Perception-based Theory of Probabilistic Reasoning // Melo-Pinto P., Teodorescu H.-N., Fukuda T. (eds.) Systematic Organisation of Information in Fuzzy Systems. – IOS Press, 2003. –* Р. *3–5. 5. Yager R..R., Zadeh L.A. Fuzzy Sets, Neural Networks and Soft Computing. – New York, Van Nostrand Reinhold, 1994.6.* Тыугу Э.Х. Концепт уальное программирование. – М.*:* Наука*, 1984. – 255* с.

УДК: 004.432

ДОСЛІДЖЕННЯ МОВ ПРОГРАМУВАННЯ JAVA ТА C# ДЛЯ СЕРВЕРНИХ ПЛАТФОРМ ТА РОБОЧИХ СТАНЦІЙ

***Устілкін В.В., *Люта М.В., **Розломій І.О.**

*Київський національний університет технологій та дизайну (ЧФ), Україна, Черкаси

**Черкаський національний університет імені Богдана Хмельницького, Україна, Черкаси

В роботі проведено аналіз особливостей використання відомих мов програмування Java та C#, та використання їх у розробці програмного забезпечення для серверних платформ та робочих станцій. Досліджено їх основні переваги та недоліки, проведено порівняння деяких відмінностей у їх синтаксисі. На основі існуючої динаміки популярності, показана перспективність мов програмування Java та C#. Шляхом простого синтетичного тесту було проведено порівняння їх швидкостей на платформі Java virtual machine версії 1.8 та Microsoft .NET Framework версії 4.5. В результаті запропоновано рекомендації, щодо подальшого використання мов програмування Java та C#.

Ключові слова: Java, C#, JVM, .NET, програмування.

Устилкин В.В. Люта М.В. Розломий И.А. Исследование языков программирования Java и C# для серверных платформ и рабочих станций / Киевский национальный университет технологий и дизайна, Украина, Черкассы

В работе проведен анализ особенностей использования известных языков программирования Java и C#, а также использование их в разработке программного обеспечения для

серверных платформ и рабочих станций. Исследованы их основные преимущества и недостатки, проведено сравнение некоторых различий в их синтаксисе. На основе существующей динамики популярности, показана перспективность языков программирования Java и C#. Путем простого синтетического теста было проведено сравнение их скоростей на платформе java virtual machine версии 1.8 и .NET Framework версии 4.5. В результате предложены рекомендации относительно дальнейшего использования языков программирования Java и C#.

Ключевые слова: Java, C#, JVM, .NET, программирование.

Ustilkin V.V. Luta M.V. Rozlomiy I.O. Research programming languages Java and C# for server platforms and workstation / Kyiv national university of Technologies and Design, Kyiv, Ukraine

The paper analyzes the features of the use of known programming languages Java and C#, as well as their use in the development of software for server and workstation platforms. We studied their main advantages and disadvantages compared to some differences in their syntax. On the basis of the existing dynamics of popularity shows prospects of programming languages Java and C#. By simple synthetic test was conducted comparing their speeds on a platform java virtual machine version 1.8 and the .NET Framework version 4.5. As a result of proposed recommendations for future use of the programming languages Java and C#.

Key words: Java, C#, JVM, .NET, programming.

Вступ. В наш час є дві популярні конкуруючі мови програмування, що стрімко розвиваються – мова Java та мова C#.
Не дивлячись на те, що C# з'явився значно пізніше ніж Java, вони мають багато спільного, але як кажуть, нічого ідеального не існує.

Даний випадок не є виключенням і в цій статті, проведено аналіз переваг та недоліків кожної з них.

Мови C# і Java з'явилися в різний час. Мова Java була створена задовго до появи C#. Під назвою Oak Java була розробленна компанією Sun Microsystems в 1990 р, а в 1995 була випущена перша бета-версія Java. Створення C# було анонсовано в 2000 році, а в 2002 році вийшла перша версія платформи .NET, що підтримує C#. Таким чином, якщо Java створювалась спираючись більшою мірою на досвід мов Objective C і C, то для C# такою опорою були C ++ і сама Java. І, незважаючи на свою назву, C# виявилась ближче до Java, ніж до C ++.

З точки зору розробника мови Java і C# дуже схожі. Обидві мови є строго типізованими, об'єктно-орієнтованими. Обидві увібрали в себе багато чого з синтаксису C ++, але на відміну від C ++, простіше в освоєнні для початківців. Обидва запозичили з C набір основних ключових слів і службових символів, в тому числі фігурні дужки для виділення блоків. Обидві мови спираються на збірку сміття. Обидві мови супроводжуються багатими колекціями бібліотек. Але є в мовах також свої особливості і відмінності, сильні і слабкі сторони. C# врахувала багато недоліків Java, і виправила їх у своїй реалізації. Але і Java не стоїть на місці, розвиваючись паралельно з C#.

Кік Редек з Microsoft вважає C # більш складною мовою, ніж Java. На його думку, «мова Java була побудована таким чином, щоб уберегти розробника від стрільби собі в ногу» (англ. «Java was built to keep a developer from shooting himself in the foot»), а «C # була побудована так, щоб дати розробникові пістолет, але залишити його на запобіжнику»(англ.« C# was built to give the developer a gun but leave the safety turned on ») [1, с. 36].

Експериментальна частина. C# є продуктом Microsoft і основним направленням даної мови є продукти для операційної системи сімейства Windows, як для стаціонарних комп'ютерів, так і для мобільних пристроїв.

Також популярність набирає ігровий двигун Unity 3d, на якому розроблюється багато ігор, в першу чергу за рахунок його кроссплатформеності (можливість запускатися на різних операційних системах) [2, с. 12].

Використання платформи Xamarin дозволяє використовувати C# для розробки програм під різні операційні системи, в тому числі і Android. Хоча система досить нова і на перший погляд дуже перспективна, точно не відомо, наскільки вона приживеться. Це пов'язано з тим, що корпорація Microsoft інколи закриває свої продукти, якщо бачить їх безперспективність. В цьому можна легко переконатись, встановивши їх основний засіб для розробки програмного забезпечення Microsoft visual studio. Якщо порівняти їх версії, то можна помітити, що більш старі версії містять такі засоби для розробки програм, які виключені в більш сучасних версіях Microsoft visual studio, наприклад J#[3, с. 18].

В такому підході Microsoft, є як переваги, так і недоліки. Перевагами є те, що на підтримку актуальності продуктів, необхідно витрачати ресурси компанії, які можна використати для розвитку більш перспективних продуктів. Основним недоліком є те, що програміст вимушений буде вчити іншу мову і починати все спочатку, що для нього не є добре, як і для репутації компанії в його очах.

Java, за статистикою 2016 року займає перше місце по популярності. Область використання даної мови дуже велика – вона підтримує можливість запускати програми майже всюди – майже всі операційні системи, персональні комп'ютери, сервера, мобільні

телефони (що використовують java mini), операційна система андроїд, GPS навігатори, відео магнітофони, супутникові системи, вимірювачі тиску, пульсометри, навіть браслети для вимірювання якості сну людини, тощо... За офіційними даними розробника, Java використовується більш ніж на 3 000 000 000 (трьох мільярдах) пристроях[4, с. 8].

Підхід розробників Java в області стабільності абсолютно протилежний компанії Microsoft. Нові компоненти Java тестуються довгий час до того, як повністю вийти на ринок і не видаляються після їх виходу. Безперечно у даного методу є суттєвий недолік, новизна продукту вводиться дуже повільно. Не дивлячись на це, на сьогоднішній день остання версія мови Java майже не поступається в нововведеннях C#[5, с. 6].

Аналіз популярності. *В зв'язку із тим, що мов програмування дуже багато, проводиться багато аналізів щодо них:*

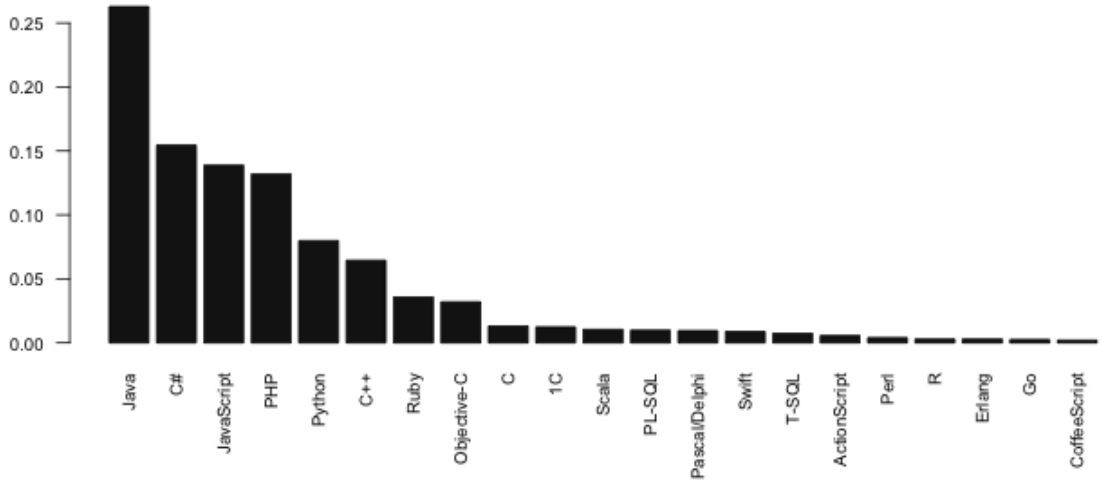


Рис. 1. Графік популярності мов програмування станом на 2016 рік

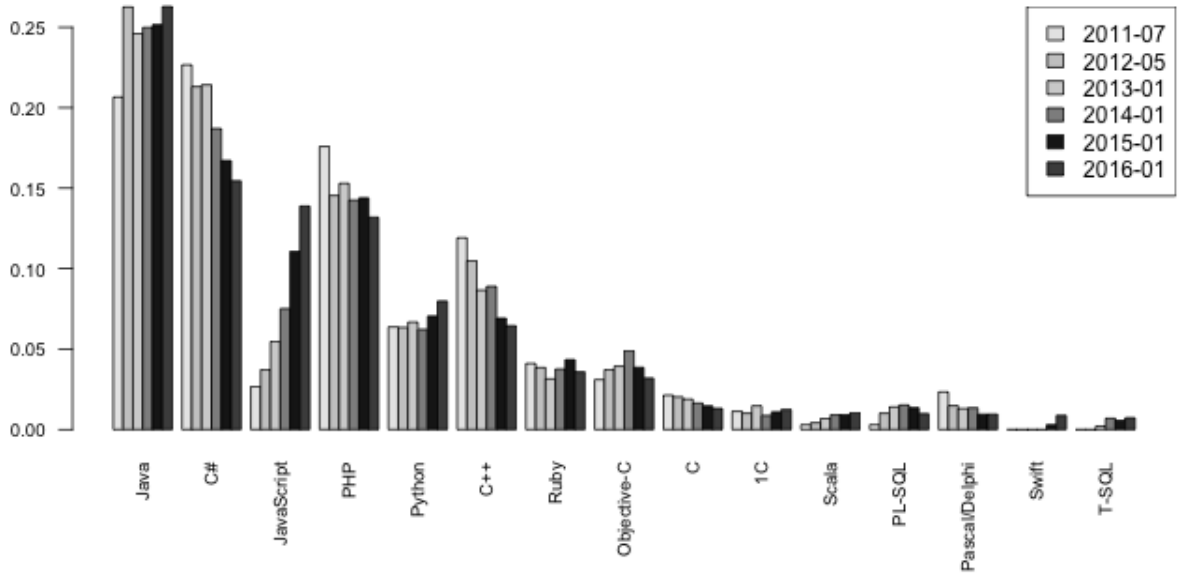


Рис. 2. Динаміка популярності мов програмування за останні роки (зліва 2011 рік, зправа 2016)

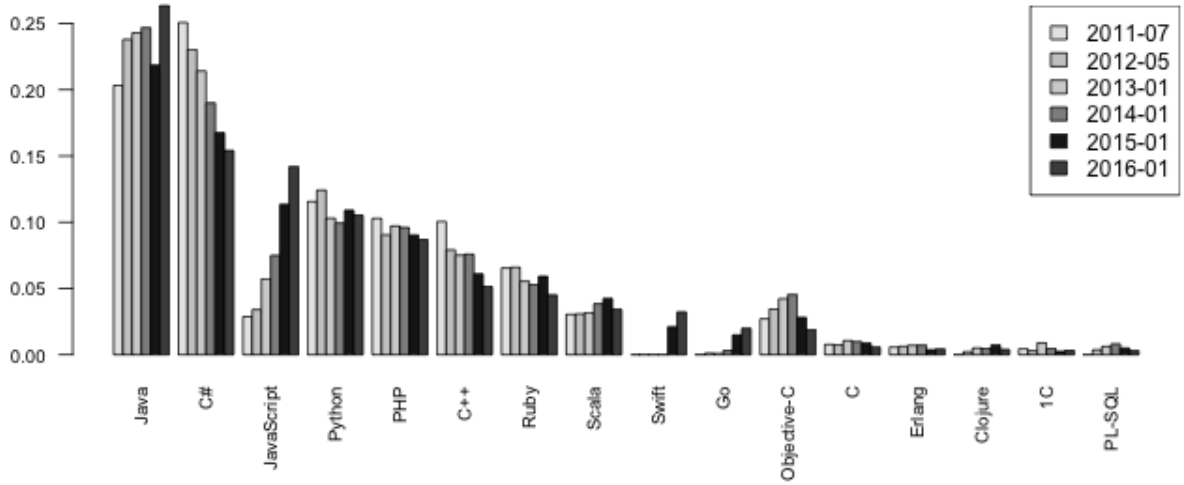


Рис. 3. Мови, яким програмісти віддають перевагу (зліва 2011 рік, справа 2016)

Порівняння швидкості написаної програми. *Дане порівняння є синтетичним тестом. Його мета порівняти, скільки часу потрібно буде програмам написаним на C# (net 4.5) версія «Release» та Java (1.8) для того, щоб виконати свої задачі в операційній системі Windows 10.*

Програми створюють процес створення однакового класу задану кількість разів (для тесту вибрано 1 000 000 000 разів), в якому виконують однакові математичні функції

```
1 reference
private void button1_Click(object sender, EventArgs e)
{
    DateTime startTime, stopTime;
    double differentTime = 0;
    long cycles = Convert.ToInt32(textBox1.Text);
    startTime = System.DateTime.Now;

    for (int i = 0; i < cycles; i++)
    {
        ProcessLoader processLoader = new ProcessLoader();
    }
    stopTime = System.DateTime.Now;
    differentTime = (stopTime - startTime).TotalMilliseconds;
    label2.Text = "Start time: " + startTime.ToString();
    label3.Text = "Stop time: " + stopTime.ToString();
    label4.Text = differentTime.ToString()+" ms";
}
```

```
3 references
class ProcessLoader
{
    public static double result = 0;
    1 reference
    public ProcessLoader()
    {
        result*=2;
        result/=2;

        result *= 0.2;
        result /= 0.2;

        result++;

    }
}
```

Рис. 4. Код програми на C#

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    long startTime, stopTime, differentTime;
    long cycles = Long.valueOf(jTextField1.getText());
    startTime=System.currentTimeMillis();
    for(int i=0;i<cycles;i++){
        ProcessLoader processLoader = new ProcessLoader();
    }
    stopTime=System.currentTimeMillis();
    differentTime = stopTime-startTime;
    jLabel2.setText("Start time: "+String.valueOf(startTime));
    jLabel3.setText("Stop time: "+String.valueOf(stopTime));
    jLabel4.setText("Different time: "+String.valueOf(differentTime));
}
```

```
public class ProcessLoader {
    public static double result=0;
    public ProcessLoader(){
        result*=2;
        result/=2;

        result*=0.2;
        result/=0.2;

        result++;
    }
}
```

Рис. 5. Код програми на Java

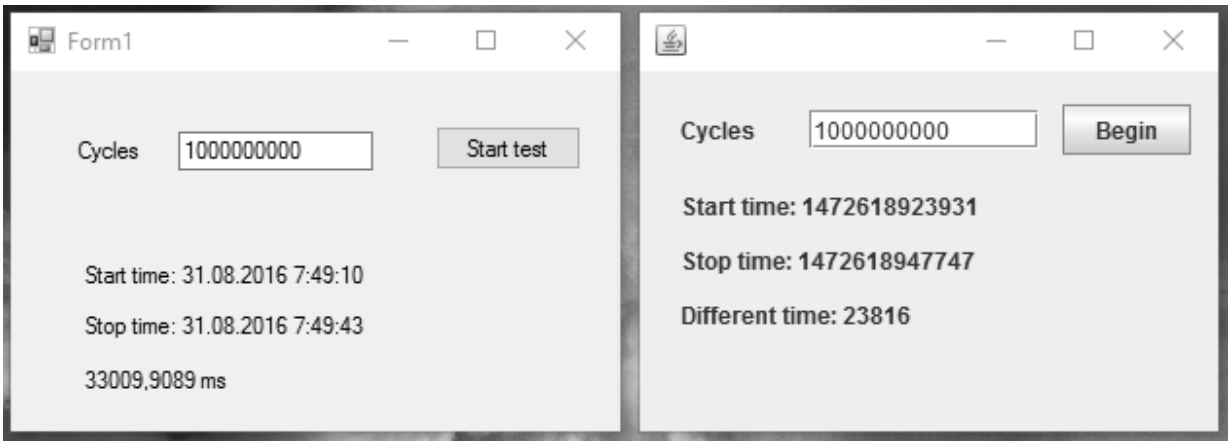


Рис. 6. Тестування швидкості виконання програми

На рис. 6 зображено 2 скриншоти з часом виконання однакової програми на одному комп'ютері, з лівої сторони C#, з правої – Java. В останньому рядку виведено час, що знадобився для виконання в мілісекундах. Програми було запущено не з редакторів, а зі створених файлів, що створювались не як Debug версії, а як Release.

В результаті тесту видно, що для виконання однакових операцій програмі на C# потрібно було більше часу ніж програмі на Java, тобто програма на Java відпрацювала в 1,38 рази швидше.

Висновки. На сьогоднішній день, Java популярніша за C#, і якщо враховувати графіки популярності – найближчим часом ситуація не зміниться.

Синтаксис C# більш лаконічний, тому що там замість довгих слів ставляться знаки. Особливою його перевагою є можливість розділяти класи на окремі частини та файли інструкцією «partial».

У плані швидкості розробки програмного забезпечення знову в лідерах C# – на ньому написати не складну програму з використанням GUI (графічний інтерфейс користувача) можна швидше ніж на Java.

У разі, якщо програма має бути кросплатформенною, при виборі серед C# та Java варіантів немає – Java в цьому випадку буде краще.

В швидкості проведеного синтетичного тесту Java перемогла зі значним відривом.

Таким чином, можна зробити висновок, якщо необхідна стабільність, кросплатформеність і швидкість виконання самої програми, то краще писати на Java, а якщо необхідно якомога швидше написати програму, яка працюватиме на базі операційної системи Windows — то вигідніше буде вибрати C#.

Література:

1. Джеффри Рихтер CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C#: Питер. – 2016. – 896 с.

2. Эндрю Троелсен, Филипп Джепикс Язык программирования C# 6.0 и платформа .NET 4.6: Вильямс. – 2016. – 1440 с.

3. Джозеф Албахари, Бен Албахари C# 6.0. Справочник. Полное описание языка: Вильямс. – 2016. – 1040 с.

4. Герберт Шилдт Java 8. Полное руководство: Вильямс. – 2016. – 1376 с.

5. Роберт Лафоре Структуры данных и алгоритмы в Java: Питер. – 2016. –704 с.

References:

1. Jeffrey Richter, CLR via C#. Programming Microsoft.NET Framework 4.5 platform in the language C#: Peter. - 2016. - 896 p.

2. Andrew Troelsen, Philip Dzhepiks C# 6.0 programming language and platform .NET 4.6: Williams. - 2016 - 1440.

3. Joseph Albahari, Ben Albahari C# 6.0. Directory. Full description of the language: Williams. - 2016 - 1040.

4. Herbert Schildt Java 8. Complete Guide: Williams. - 2016 - 1376 p.

5. Robert Laforêt Data Structures and Algorithms in Java: Peter. - 2016. - 704 p.

function. IEEE Trans. on Neural Networks 9, 1496-1506 (2010). 15. Tymoshchuk, P.V.: A dynamic K-winners take all analog neural circuit. In: IVth IEEE Int. Conf. "Perspective technologies and methods in MEMS design", pp. 13-18. IEEE Press, L'viv (2008). 16. Hopfield J.J. Neurons with graded response have collective computational properties like those of two-state neurons // in Proceedings of the Natonal Academy of Sciences, №81, pp. 3088-3092, 1984. 17. Grossberg S. Non-Linear Neural Networks: Principles, Mechanisms, and Architectures, Neural Networks, vol. 1, pp. 17-61, 1988. 18. Calvert B.D. and Marinov C.A. Another k-Winner-take-all analog neural network // IEEE Trans. Neural Networks, vol. 11, № 4, pp. 829-838, July 2000. 19. Marinov C.A. and Calvert B.D. Performance analysis for a K-winners-take-all analog neural network // basic theory // IEEE Trans. Neural Networks, vol. 14, № 4, pp. 766-780, July 2003. 20 16. Liu, Q., Wang, J.: Two k-winners-take-all networks with discontinuous activation functions. Neural Networks 21, 406-413 (2008).

УДК **621.452.001.57:681.54**

М.Ю. Шабатура
Національний університет "Львівська політехніка",
кафедра електронних обчислювальних машин

СПЕЦІАЛІЗОВАНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ІНТЕРАКТИВНОЇ КОМП'ЮТЕРИЗОВАНОЇ СИСТЕМИ

Описано функціональну структуру та особливості розробленого спеціалізованого програмного забезпечення інтерактивної комп'ютеризованої системи корекції стану користувача зі зворотним зв'язком.

Ключові слова: інтерфейс, інтерактивність, програмне забезпечення, Fuzzy logic.

Described functional structure and features of the special software designed for user's state correction interactive computerized system with feedback.

Key words: interface, interactive, software, Fuzzy logic.

Вступ

Широке впровадження комп'ютерних технологій передбачає не тільки створення досконаліших технічних елементів комп'ютерних систем, але й розроблення потужного програмно-математичного забезпечення. Причому аналіз світових тенденцій [1] показує, що програмна складова є вагомішою у цьому процесі. Сьогодні є чимало технічних систем, до складу яких комп'ютер входить як одна з ключових ланок, що водночас формує всю ідеологію функціонування таких систем і забезпечує стійкість їх інформаційного простору та узгоджену роботу всіх складових. Апаратні пристрої таких систем повинні мати відповідне програмне представлення, яке може бути у двох формах: у формі підпрограми-драйвера, яку розпізнає операційна система, що дозволить комп'ютеру використовувати цей пристрій як власний технічний ресурс: або ж у формі повноцінного спеціалізованого програмного забезпечення, яке відобразиться в комп'ютері у вигляді певного додаткового сервісу з власним інтерфейсом інтерактивної взаємодії з користувачем, що даватиме йому змогу використовувати додаткові функції оперування технічним ресурсом.

Отже, для повноцінного функціонування розробленої в роботі [2] інтерактивної комп'ютеризованої системи корекції стану користувача зі зворотним зв'язком (ІКСЗЗ) створення спеціалізованого програмного забезпечення є необхідним, актуальним і важливим фактором розвитку і вдосконалення.

Постановка задачі дослідження

Розробити спеціалізоване програмне забезпечення для інтерактивної комп'ютеризованої системи корекції стану користувача зі зворотним зв'язком відповідно до функціональної структури системи.

Аналіз досліджень і публікацій

Невід'ємною і, очевидно, найважливішою складовою розробленої у роботах [2–5] ІКСЗЗ є спеціалізоване програмне забезпечення. Окрім виконання класичних функцій, які потрібні для забезпечення узгодженої роботи усіх технічних складових ІКСЗЗ, спеціалізоване програмне забезпечення повинно формувати специфічний інтерфейс, який, власне, і створюватиме інтерактивну, інтелектуальну взаємодію користувача із системою.

Основна частина

ІКСЗЗ є складним програмно-апаратним комплексом, який об'єднує: персональний комп'ютер, мікроконтролерний модуль, систему спеціальних сенсорів, штатні периферійні пристрої відтворення аудіо та візуальних впливів, клавіатуру, маніпулятор тощо. Метою функціонування ІКСЗЗ є комплексний аналіз стану користувача і, якщо виявлено його відхилення від умовно-еталонного для певного користувача стану, забезпечення, за допомогою інтерактивної взаємодії аудіо та візуальних впливів, відповідної корекції поточного стану користувача ПК.

Функціональну структуру ІКСЗЗ зображено на рис. 1.

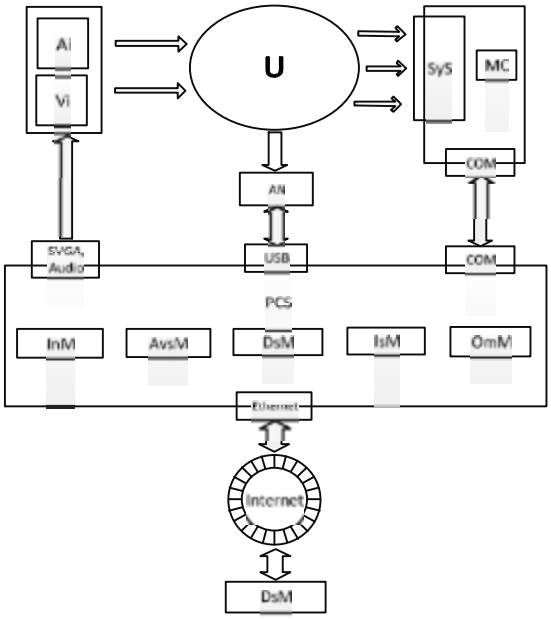


Рис. 1. Функціональна структура інтерактивної комп'ютеризованої системи зі зворотним зв'язком

Основними компонентами зображеної на рисунку функціональної структури ІКСЗЗ є:

PCS – персональна комп'ютеризована система (ПК, КПК, планшет тощо);

Ai – засіб аудіовпливу (навушники, звукові системи);

Vi – засіб візуального впливу (генеровані зображення на моніторі PCS);

U – користувач (характеризується множиною параметрів, які описують його стан), володіє знаннями з дотримання алгоритму поведінки під час інтерактивної взаємодії із системою;

SyS – система сенсорів, містить неінвазивні датчики для об'єктивного оцінювання параметрів стану користувача (з метою спрощення викладень і оптимізації обсягу статті розглянемо лише використання сенсорів дихання та пульсу);

MC – мікроконтролер, дозволяє узгодити та синхронізувати роботу системи сенсорів і забезпечити передачу даних на PCS;

AN – засоби активної взаємодії (монітор, клавіатура, маніпулятори);

USB – послідовний інтерфейс передачі даних для середньо- та низькошвидкісних периферійних пристроїв у обчислювальній техніці;

COM (RS-232C) – двонапрямлений послідовний інтерфейс;

Ethernet (IEEE 802.3) – інтерфейс для мережевого підключення;

SVGA, Audio – інтерфейси пристроїв аудіо та візуального впливу;

InM – інформаційний пакет для ознайомлення з правилами взаємодії із системою;

AvsM – модуль синтезу аудіо та візуальних впливів, доповнений компонентою управління темпом дихання;

DsM – система накопичення і збереження образів стану користувача;

IsM – система ідентифікації поточного стану користувача;

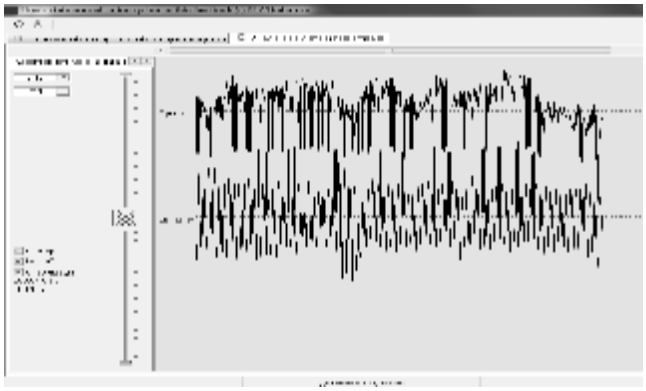
OmM – система оперативного моделювання прийняття рішень на основі Fuzzy logic;

DsM – сервер зі спеціалізованим програмним забезпеченням для дистанційної підтримки функціонування ІКСЗЗ.

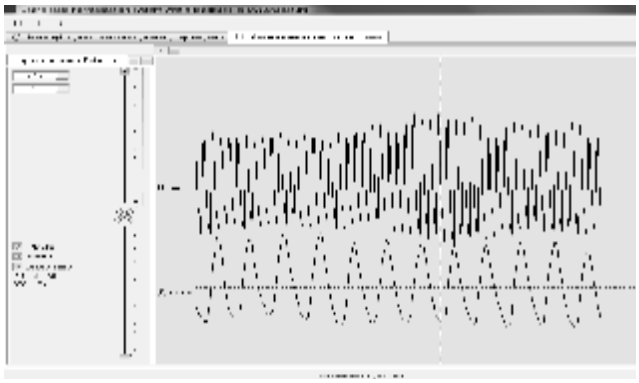
У цій роботі розглянемо найважливішу функцію створеного спеціалізованого програмного забезпечення, суть якої полягає у формуванні комунікативного інтерфейсу між користувачем та комп'ютеризованою системою.

Програмне забезпечення написане мовою програмування C++ в середовищі Embarcadero C++ Builder з використанням компонентів: Embarcadero Audio API – для роботи зі звуком, OpenGL бібліотеки – для рендерингу графічних елементів та бібліотеки TMS – для роботи з портами персонального комп'ютера.

Існує два режими функціонування ІКСЗЗ: навчальний та робочий.



а

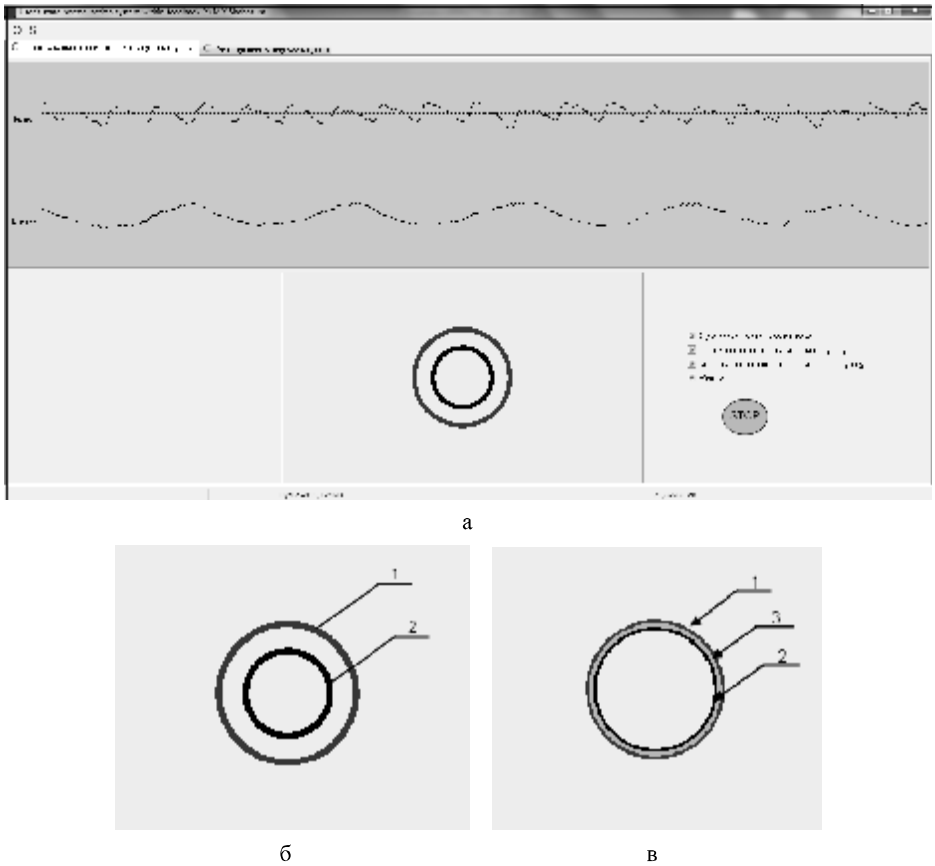


б

Рис. 2. Вигляд робочого вікна програми у режимі навчання при:
а – зашумлених; б – відфільтрованих показниках частоти дихання та частоти пульсу користувача

На рис. 2 зображено робоче вікно програмного забезпечення ІКСЗЗ у режимі навчання, яке ініціалізується під час першого запуску програми, з метою індивідуального аналізу відкоригованого стану, тобто створення умовно-еталонної моделі стану користувача. Зокрема, на рис. 2, а показано

неперервне накопичення значень зміни показників пульсу та рівня дихання користувача в режимі реального часу, відповідно подано у вигляді частотних графіків. Відфільтровані показники стану користувача, як результат застосування Fuzzy logic фільтра моделі нечіткого виведення типу Мамдані, зображено на рис. 2, б. Користувач, за бажанням, може самостійно простежити візуалізовану динаміку зміни свого стану, в будь-який часовий проміжок, однак набагато краще і без будь-яких можливих суб'єктивних помилок за нього цю функцію виконає програма системи, яка автоматично проаналізує та збереже отримані дані в спеціалізованій базі даних (DsM).



а

б в

Рис. 3. Вигляд вікна програми у робочому реж імі: а – загальний вигляд робочого вікна; б – фаза недот римання рекомендованого рівня дихання; в – фаза дот римання рекомендованого рівня дихання

На рис. 3, а зображено вигляд вікна програми в робочому режимі, зокрема візуальне представлення комбінації параметрів частоти дихання та частоти пульсу користувача у формі кола червоного кольору (рис. 3, б, позначка 1), яке звужується/розширюється відповідно до зміни параметрів стану в реальному масштабі часу, також наявне коло чорного кольору (рис. 3, б, позначка 2), як представлення необхідного умовно-еталонного, збалансованого показника рівня дихання та пульсу користувача, та вигляд під час дотримання користувачем рекомендованого рівня дихання (поява зеленого кола, рис. 3, в, позначка 3).

Щоб досягти максимального ефекту інтерактивної взаємодії ІКСЗЗ та користувача, рекомендовано активувати функцію аудіовпливу (вибравши у відповідному пункті меню опцію «Аудіовплив на стан користувача») та використовувати навушники із широким частотним діапазоном.

Під час проектування програми було враховано і спеціально підібрано кольорову гаму та графічне компонування елементів у вікні з метою підвищення інтуїтивної зрозумілості для користувача системи. Розроблена програма може бути інстальована не лише на традиційних настільних

комп'ютерах і ноутбуках, але й на різноманітних портативних пристроях, зокрема планшетах та КПК, які функціонують на основі операційних систем сім'ї ОС Microsoft Windows.

Висновки

Розроблене спеціалізоване програмне забезпечення інтерактивної комп'ютеризованої системи корекції стану користувача зі зворотним зв'язком з інтуїтивно зрозумілим дизайном та вимогами, поставленими відповідно до раніше розробленої моделі комп'ютеризованої системи на основі Fuzzy logic математичного апарату.

Інтерактивна взаємодія користувача та ІКСЗЗ відбувається на рівні аудіо та візуальних впливів на стан користувача, з метою його корекції у реальному масштабі часу.

1. The Software & Information Industry Association / http://www.siia.net/ 2. Shabatura M. Комп'ют еризована сист ема впливу на ст ан корист увача з інт елект уальним зворот ним зв'язком. // *Materialy Miedzynarodowej Naukowi-Praktycznej Konferencji «Perspektywy rozwoju nauki we współczesnym świecie /* мат ериалы меж дународной научно-практ ической конференции «Перспект ивы развит ия науки в современном мире.», *29.03.2012 – 31.03 2012, Poland,* Краков/*Kraków, 71–73 p. 3. Shabatura M.* Програмні аспект и комп'ют ерної т ехнології оцінки ст ану корист увача в процесі інт еракт ивної взаємодії // *Materialy Miedzynarodowej Naukowi-Praktycznej Konferencji «Innowacje i badania naukowe, jak rowniez ich zastosowanie w praktyce" /* мат ериалы меж дународной научно-практ ической конференции «Инновации и научные исследования, а т акж е их применение на практ ике», *29.05.2012 – 31.05.2012, Poland,* Варшава*Warszawa, 48–50 p. 4. M. Shabatura. User's state normalization computerized system within an intelligent feedback // Materialy Miedzynarodowej Naukowi-Praktycznej Konferencji «Postepow w nauce. Nowe poglady, problemy, innowacje. //* Мат ериалы меж дународной научно-практ ической конференции «Дост иж ения в науке. Новые взгляды, проблемы, инновации», *29.07.2012 – 31.07.2012, Poland,* Лодзь*Lodz, 14-16 p. 5.* Шабат ура М.Ю., Мельник А.О. Мет одологія вибору т а заст осування аудіовізуальних пот оків в комп'ют еризованій сист емі впливу на ст ан корист увача з інт елект уальним зворот ним зв'язком // Сист еми озброєння і військова т ехніка. *[*підрозділ*:* Теорет ичні сист еми розробки сист ем озброєння*].* Харківський університ ет повіт ряних сил ім. Івана Кож едуба. – *4(28)2011.* – С. *109 – 115.*

6П

УДК 681.325.36

І.Р. Піт ух
Карпатський державний центр інформаційних засобів і технологій НАН України, м. Івано-Франківськ, Україна
bima@buc.tr.ukrtel.net

Матричні моделі архітектур розподілених комп'ютерних систем та методологія побудови алгоритму діагностування руху даних центральним сервером

У статті викладена методологія побудови матричних моделей руху даних архітектур комп'ютерних систем. Запропоновані принципи побудови комп'ютерних мереж з глибоким розпаралелюванням інформаційних потоків на основі двовимірних і тривимірних матричних моделей руху даних вирішують актуальну науково-технічну задачу методології розрахунку ефективності руху даних в таких мережах.

В даний час існує широка різноманітність архітектур інформаційних систем, до яких належать концентровані та розподілені системи обробки даних. До систем першого класу можна віднести монопольні архітектури, архітектури з розподіленим часом, архітектури з мультипрограмною та мультипроцесорною обробкою даних. Другий клас представлений значним числом однорівневих архітектур сучасних комп'ютерних мереж, в тому числі: магістральні, зіркові, кільцеві, систолічні [1], [2]. До класу багаторівневих розподілених архітектур інформаційних систем слід віднести ієрархічні, багаторівнево-магістральні та зірково-магістральні архітектури [3].

Окремим класом архітектур представлені безпровідні радіотехнічні інформаційні системи та комп'ютерні мережі наступного типу:
– безретрансляторні;
– з пасивними ретрансляторами;
– з активними ретрансляторами, в тому числі сотові мережі.

Комп'ютерні системи з оптичними каналами зв'язку охоплюються архітектурами на основі:
– дуплексних оптичних ретрансляторів;
– оптичних активних ретрансляторів;
– оптичних сканерів;
– волоконно-оптичних ліній зв'язку.

Значною оригінальністю архітектур характеризуються спеціалізовані комп'ютерні системи (СКС), які часто можуть базуватися на об'єднанні окремих елементів різних типових архітектур [4-6]. До такого класу інформаційних систем, наприклад, належать:
– системи обліку витрат енергоносіїв з глибоким розпаралелюванням потоків даних;
– комп'ютерні розподілені системи екологічного моніторингу;
– спеціалізовані охоронні системи;
– проблемно-орієнтовані корпоративні системи промислових та адміністративних організацій.

Така велика кількість архітектур інформаційних систем значною мірою ускладнює вирішення задач оптимізації проектних рішень при побудові інформаційних систем, що потребує розробки відповідних моделей архітектур, які б дозволили шляхом формалізації структурних елементів різних мереж з єдиних позицій провести дослідження та порівняння їх системних характеристик. Одним з перспективних підходів до вирішення такої задачі є використання теорії та технології побудови одновимірних та багатовимірних матричних моделей руху даних [3], що визначає актуальність таких досліджень.

На рис. 1 подана класифікація архітектур комп'ютерних систем з фізичними лініями зв'язку.

Рисунок 1 – Класифікація архітектур КС з фізичними лініями зв'язку

На рис. 2 подана класифікація архітектур КС з безпровідними лініями зв'язку.

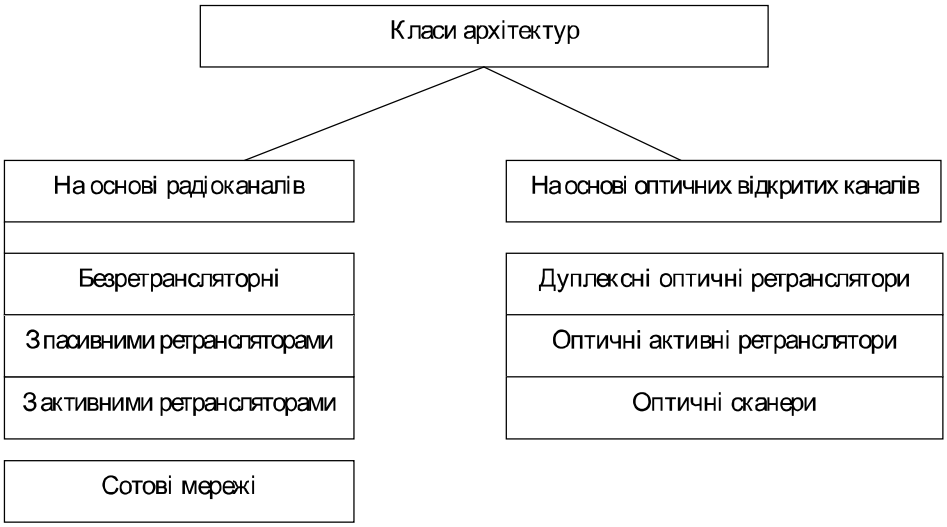


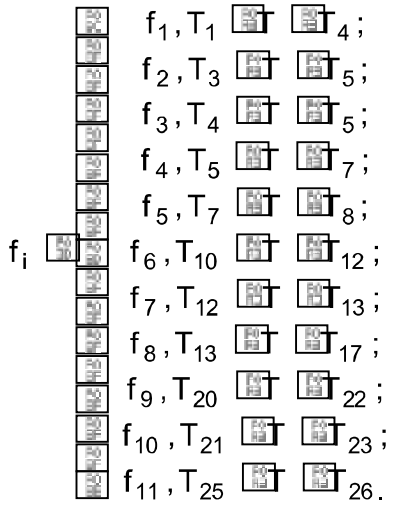
Рисунок 2 – Класифікація архітектур КС з безпровідними радіоканалами

Подані класифікації архітектур КС дозволяють виконати формалізований вибір відповідного класу архітектури КС залежно від їх проблемної орієнтації та необхідних системних характеристик.

Суть побудови даної моделі методично виконується формалізацією процедур побудови оптимального несуперечного змістовного графа розгалуженого алгоритму у наступному порядку:

– формалізація умови задачі;
– побудова суміщеного часового графа;
– побудова логічного розгалуженого графа;
– покриття логічного графа блок-схемою;
– нумерація операторів блок-схеми.

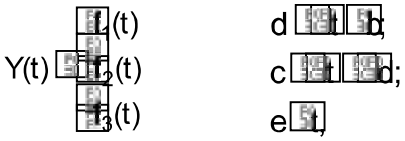
Формалізація моделі «блок-схема алгоритму»:



Формалізація умови задачі полягає у відповідності до нумерації системних функцій $f_i(r)$, які в реальній процедурі описуються системою аналітичних виразів, які виконуються при заданих часових організаціях.

Наприклад, для умови задачі $f_i(r)$ – багатоканальне аналого-цифрове перетворення технологічних параметрів; $f_2(t_2)$ – ковзне усереднення формуючих відліків процесів; $f_3(t_3)$ – обчислення матриці коефіцієнтів кореляції; $f_4(t_4)$ – індикація параметрів.

Формалізація умови задачі побудови моделі «блок-схеми алгоритму»:



де a, b, c, d, e – часові обмеження.

Нехай a = 1, b = 2, c = 4, d = 6, e = 5. Тоді суміщений часовий граф (СЧГ) має вигляд, наведений на рис. 3 – 4, де стрілки вказують, що відповідні системні процедури виконуються до настання часу t b або пізніше часу t e.

Змінюючи значення часових обмежень a, b, c, d, e і здійснюючи розпаралелювання операцій виконання системних процедур $f_i(t)$, одержимо відповідно різні реалізації суміщеного графа (рис. 3).

Для визначення формалізованої методики побудови розгалуженого логічного графа (РЛГ) сформулюємо ряд стверджень:

1. Системними атрибутами логічного графа є 5 вершин: початок, ввід-вивід, оператор системної функції, умова і кінець.

2. Основним атрибутом розгалуженого логічного графа є умова, причому:

2.1) якщо умова виконується, ЛГ розширюється вправо, в протилежному випадку – вниз;

2.2) якщо умова не виконується, то вимагається її уточнення, граф розширюється зліва вниз;

3. Вид суміщеного часового графа одночасно визначає структуру логічного графа, причому:

3.1) якщо системні функції на СЧГ не перетинаються і не накладаються, то ЛГ розширюється вправо і вниз, а для виводу використовується одна загальна вершина;

3.2) в протилежному випадку ЛГ розширюється тільки вниз, а для виводу використовується автономна вершина після кожного оператора системної функції.

Приклади побудови розгалужених ЛГ для суміщених часових моделей ілюструє рис. 3.

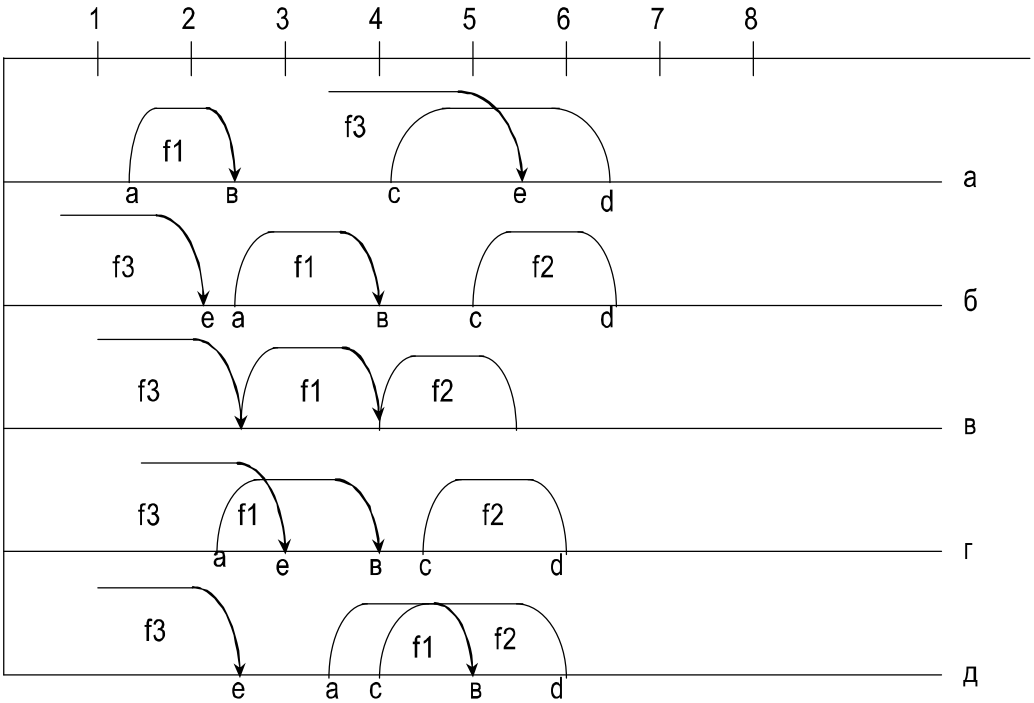


Рисунок 3 – Приклади суміщених часових графів

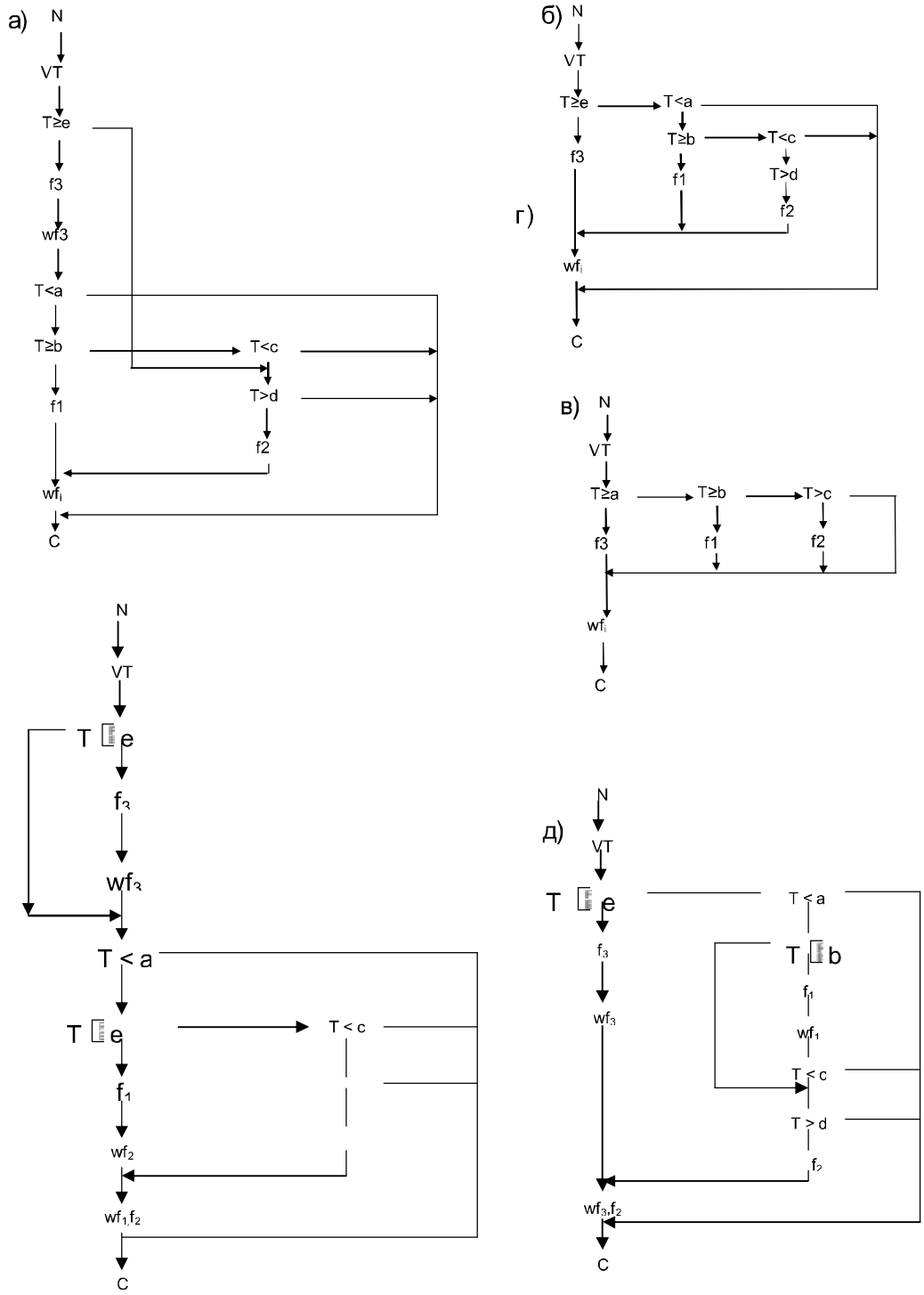


Рисунок 4 – Приклади побудови розгалужених логічних графів

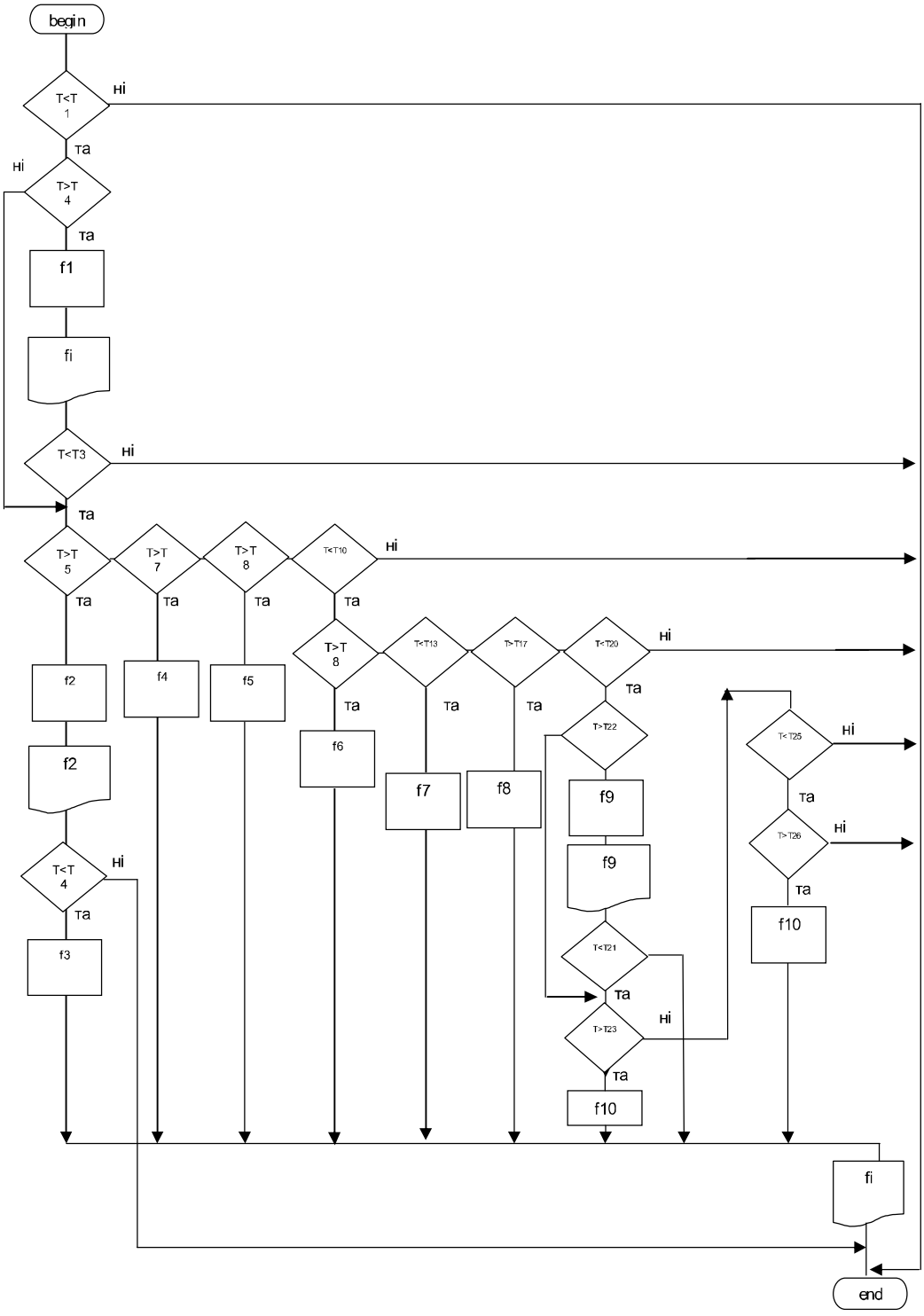


Рисунок 5 – Блок-схема алгоритму оброблення даних

Викладена методологія побудови граф-схем та алгоритму діагностування руху даних в КС реалізована у вигляді САПР у середовищі Delphi.

Література

1. Цикритзис Д., Лоховски Ф. Модели данных. – М.: Финансы и статистика, 1985. – 343 с.
2. Столлингс В. Современные компьютерные сети. – СПб: Питер, 2003.– 783 с.
3. Хаусли Т. Системы передачи и телеобработки данных: Пер. с англ. – М.: Радио и связь, 1994. – 456 с.
4. Гриценко В.И., Урсатьев А.А. Распределенные информационные системы. Состояния. Перспективы развития // Управляющие системы и машины. – 2003. – №4. – С. 11-21.
5. Гриценко В.И., Котиков Е.А., Урсатьев А.А. и др. Модель распределенной информационной системы широкого применения // УСиМ. – 1999. – №5. – С. 32-42.
6. Pitukh I., Nykolaychuk Y., Vozna N. Principles of computer networks construktion with deep paralleling of information flows on the basis of matrix models of data movement // Матеріали VIII Міжнародної науково-технічної конференції TCSET' 2004. – Львів; Славське. – С. 417-419.

И.Р. Пит ух
Матричные модели архитектур распределенных компьютерных систем и методология построения алгоритма диагностирования движения данных по центральному серверу
В статье изложена методология построения матричных моделей движения данных архитектур компьютерных систем. Предложенные принципы построения компьютерных сетей с глубоким распараллеливанием информационных потоков на основе двумерных и трёхмерных матричных моделей движения данных решают актуальную научно-техническую задачу методологии расчета эффективности движения данных в таких системах.

I.R. Pitukh
The Matrix Models of the Distributed Computer Systems Architecture and Methodology of Algorithm Construction of Data Motion Diagnosis by the Central Server
In the article the methodology of construction of matrix models of motion of the architecture of computer systems is given. The offered principles of construction of computer networks with deep paralleling of information streams on the basis of two-dimensional and three-dimensional matrix models of data motion solve an actual scientific and technical problem of methodology of calculation of efficiency of data motion in such systems.

Ст ат т я надійшла до редакції 10.07.2008.