Представление основных структур программирования. Типы, определяемые пользователем. Массивы. Динамические данные.

1. Операторы безусловного и условного переходов.
2. Операторы циклов.
3. Структуры данных и их реализации.
4. Понятие массива данных. Задачи обработки массивов.
5. Понятие записи. Задачи обработки записей.

Литература

1. Освой самостоятельно Borland C++ 5 / Крейг Арнуш. – М.: Бином, 1997. – 720 с.
2. Borland C++ Builder 5. Энциклопедия программиста / Чарльз Калверт, Кент Рейсдорф: Пер с англ. – К.: Издательство ДиаСофт, 2001. – 944 с.
3. Начальный курс С и С++ / Б.И. Березин, С.Б. Березин – М: ДИАЛОГ-МИФИ, 1996. – 288 с.
4. С/С++ в задачах и примерах / Н.Б. Культин – СПб: БХВ-Перербург, 2001. – 280 с.
5. Освой самостоятельно Borland C++ Builder 3 / Кент Рейсдорф. Пер. с англ. – М.: ЗАО “Издательство БИНОМ”, 1999. – 736 с.

1. Операторы безусловного и условного переходов

Оператор безусловного перехода goto

Язык С обладает всеми возможностями для написания хорошо структурированных программ. Апологеты структурного программирования считают дурным тоном использование оператора goto, без которого было тяжело обойтись в таких языках, как FORTRAN, или BASIC. В языке MODULA Николас Вирт, автор языков Pascal и MODULA, исключил оператор goto совсем. Тем не менее, оператор goto в языке С есть и иногда он может быть полезен, хотя без него можно обойтись в любой ситуации. Для использования оператора goto надо ввести понятие метки (label). ***Метка*** – это идентификатор, за которым следует двоеточие. Метка должна находиться в той же функции, сто и оператор goto. Одно из полезных применений оператора goto – это выход из вложенных циклов:

```
for()
{
        while()
        {
                for()
                {
                        …
                        goto exit1:
                        …
                }
        }
}
exit1: printf(“Быстрый выход из вложенных циклов”);
```

Язык С++ поддерживает оператор goto, но до сих пор идут жаркие дебаты по поводу его использования.

Операторы условных переходов

Конструкции принятия решений

Конструкции принятия решений – методы, с помощью которых приложение (программа) может решать разные задачи в зависимости от выполнения тех или иных условий. Например, если программа запрашивает пользователя, продолжать действие или нет, то она должна действительно иметь возможность продолжать или не продолжать его в зависимости от входных данных пользователя. Это и есть пример первых конструкций, принимающих решение.

Одноальтернативный оператор if

В отличие от многих языков программирования, C++ не имеет ключевого слова then ни в одной из форм оператора if. Чтобы в этой ситуации отделять проверяемое условие заключается в круглые скобки.

Синтаксис	Пример
if(условие) – для единственного оператора оператор;	if(number < 0) number = 0

if(условие) – для последовательности операторов
{
<последовательность операторов>
}
	if((x – 54) < 8)
	{
	z = a * x;
	y += z;
	}

C++ использует открывающие и закрывающие фигурные скобки для обозначения блока операторов. На рисунке 1 показана диаграмма выполнения одноальтернативного оператора if.

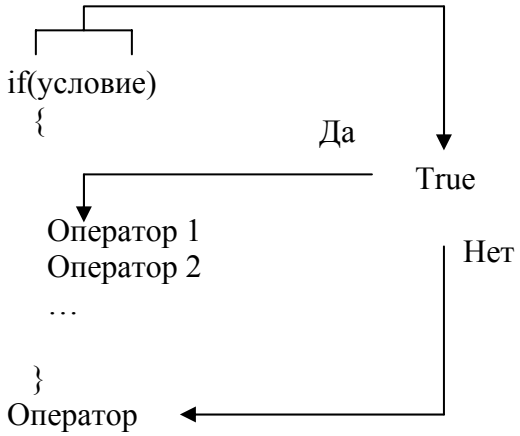

Рисунок 1. Диаграмма выполнения одноальтернативного оператора if.

В листинге 1 показана программа с одноальтернативным оператором if. Программа предлагает пользователю ввести ненулевое число и запоминает входные данные в переменной x. Если значение x не равно нулю, программа отображает значение, обратное x.

```
#include <iostream.h>
int main()
{
	double x;
	cout << “Введите ненулевое число”;
	cin >> x;
	if(x != 0)
		cout << “Обратное число от ” << x << “ это ” << (1/x) << endl;
	return 0;
}
```

Листинг 1.

Программа в листинге 1 объявляет переменную x типа double в функции main. Выходной оператор предлагает ввести ненулевое число, а входной оператор запоминает введенные данные в переменной x. Оператор if в следующей строке определяет, равно ли x нулю. Если это условие равно true, программа выполняет выходной оператор, который отображает значение x и обратное значение равное 1/x. Если проверяемое условие равно false,

Синтаксис	Пример

if(условие) – для единственного оператора в каждом
оператор1; из предложений
else
оператор2;
	if(number < 0)
	number = 0;
	else
	number ++;

if(условие) – для последовательности операторов в
{ каждом из предложений
<последовательность операторов № 1>
}
else
{
<последовательность операторов № 2>
}
	if((x – 54) < 8)
	{
	z = a * x;
	y += z;
	}
	else
	{
	z = a / x;
	y -= z;
	}

программа пропускает этот оператор и сразу переходит к оператору return.

Двухальтернативный оператор if – else

В двухальтернативной форме оператора if ключевое слово else отделяет друг от друга операторы, которые используются при выполнении каждой из альтернатив. ***Двухальтернативный оператор if – else*** обеспечивает два альтернативных направления действий в зависимости от значения проверяемого булева условия.

На рисунке 2 показана диаграмма выполнения двухальтернативного оператора if – else.

В листинге 2 показан пример использования двухальтернативного оператора if – else.

```
#include <iostream.h>
#include <ctype.h>
int main()
{
        char c;
        cout << “Введите букву ”;
        cin >> c;
        c = toupper(c);
        if(c >= ‘A’ && c <= ‘z’)
                cout << “Введена буква” << endl;
```

```
        else
                cout << “Это не буква” << endl;
        return 0;
}
```

Листинг 2.

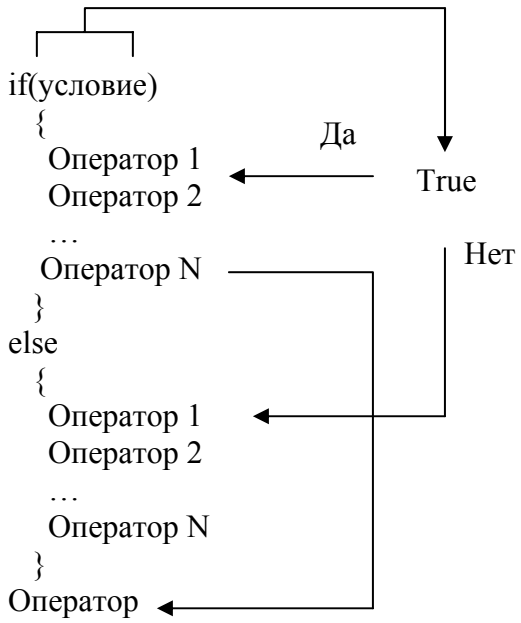

Рисунок 2. Диаграмма выполнения двухальтернативного оператора if – else.

Программа предлагает ввести букву и определяет, действительно ли введена буква или же какой-то другой символ. Входной оператор получает данные и запоминает в символьной переменной c. Следующий оператор преобразует значение переменной c в верхний регистр вызовом функции toupper (прототип находится в файле ctype.h). Это преобразование упрощает проверяемое условие в операторе if – else. Оператор if – else определяет, находится ли элемент, содержащийся в переменной c, в диапазоне от ‘A’ до ‘z’. Если это условие имеет значение true, программа выполняет выходной оператор в котором указывается, что введенный символ является буквой. В противном случае, если значение проверяемого условия равно false, программа выполняет предложение else, в котором выводится сообщение, что введенный символ не является буквой.

С двухальтернативным оператором if существует потенциальная проблема, возникающая в том случае, когда предложение if включает другой одноальтернативный оператор if. В этом случае компилятор считает, что предложение else относиться к вложенному оператору if (вложенный оператор if – это оператор, который содержится в другом операторе if и/или предложении else).

Рассмотрим следующий пример:

```
if (i > 0)
        if (i == 10)
```

```
                cout << “Вы отгадали заданное число”;
    else
                cout << “Число вышло за пределы”;
```

В этом фрагменте кода, когда переменная i есть положительное число, отличное от 10, код отображает сообщение “Число вышло за пределы”, что, разумеется, неверно. Это происходит потому, что компилятор обрабатывает операторы и трактует фрагмент кода как:

```
if (i > 0)
        if (i == 10)
                cout << “Вы отгадали заданное число”;
else
                cout << “Число вышло за пределы”;
```

Чтобы решить эту проблему, необходимо поместить вложенный оператор if в операторный блок:

```
if (i > 0)
{
        if (i == 10)
                cout << “Вы отгадали заданное число”;
}
else
                cout << “Число вышло за пределы”;
```

Многоальтернативный оператор if – else

C++ предоставляет возможность вкладывать операторы if – else друг в друга для создания многоальтернативных форм, что повышает мощь и гибкость приложений. Диаграмма выполнения многоальтернативного оператора if – else представлена на рисунке 24.

Многоальтернативный оператор if **– else** содержит вложенные операторы if – else.

Многоальтернативный оператор if – else выполняет ряд последовательных проверок до тех пор, пока не произойдет одно из следующих событий:

- Одно из условий в предложении if или в предложении else if имеет значение true. В этом случае выполняются соответствующие операторы.
- Ни одно из проверяемых условий не имеет значения true. Программа выполняет операторы во всеохватывающем предложении else (если оно существует).

Многоальтернативный оператор if-else имеет следующий общий синтаксис.

```
if(проверяемое_условие1)
        оператор1; | {< последовательность оператор № 1>}
else if (проверяемое_условие2)
        оператор2; | {< последовательность оператор № 2>}
…
```

```
else if (проверяемое_условиеN)
        операторN; | {< последовательность оператор № N>}
[else
        операторN + 1; | {< последовательность оператор № N + 1>}]
```

Рисунок 3. Диаграмма выполнения многоальтернативного оператора if – else.

Рассмотрим листинг 3 иллюстрирующий использование многоальтернативного оператора if-else.

```
//Программа, демонстрирующая многоальтернативный оператор if
#include <iostream.h>
int main()
{
        char c;
        cout << “Введите символ: ”;
        cin >> c;
        if(c >= ‘A’ && c <= ‘Z’)
                cout << “Вы ввели прописную букву” << endl;
        else if(c >= ‘a’ && c <= ‘z’)
                cout << “Вы ввели строчную букву” << endl;
        else if(c >= ‘0’ && c <= ‘9’)
                cout << “Вы ввели цифру” << endl;
        else
                cout << “Вы ввели не буквенно-цифровой символ” << endl;
        return 0;
}
```

Листинг 3.

В программе объявляется переменная с типа char. Выходной оператор предлагает ввести символ, а входной оператор получает введенные данные и запоминает их в переменной c. Многоальтернативный оператор if-else проверяет следующие условия:

- В строке 8 оператор if определяет, содержит ли переменная с буквы в диапазоне от ‘A’ до ‘Z’. Если это условие есть true, программа сообщает об этом в строке 9. Дальнейшее выполнение программы продолжается со строки 16.
- Если условие в строке 8 имеет значение false, программа переходит к первому предложению else if в строке 10. Здесь программа определяет, содержит ли переменная с букву в диапазоне от ‘a’ до ‘z’. Если это условие есть true, программа в операторе вывода в строке 11 сообщает о вводе строчной буквы. Дальнейшее выполнение программы продолжается со строки 16.
- Если условие в строке 10 имеет значение false, программа переходит ко второму предложению else if в строке 12. Здесь программа определяет, содержит ли переменная с цифру. Если это условие имеет значение true, программа выполняет выводной оператор в строке 13, который информирует о том, что была введена цифра. Дальнейшее выполнение программы продолжается со строки 16.
- Если условие в строке 12 имеет значение false, программа переходит к всеохватывающему else в строке 14 и выполняет выводной оператор в строке 15, сообщающий о том, что введенный символ не является ни буквой, ни цифрой.

Оператор switch

Оператор switch предлагает специальную форму создания многоальтернативного решения. Это позволяет исследовать разнообразные значения выражения, тип которого совместим с целым, и выбирать соответствующее направление действий.

Общий синтаксис оператора switch имеет вид:

```
switch(выражение)
{
        case constant1_1:
        [case constant1_2: …]
                <один или несколько операторов>
                break;
        case constant2_1:
        [case constant2_2: …]
                <один или несколько операторов>
                break;
…
        case constantN_1:
        [case constantN_2: …]
                <один или несколько операторов>
```

```
                break;
        default:
                <один или несколько операторов>
                break;
}
```

Пример:

```
ok = true;
switch(op)
{
        case '+':       z = x + y;
                break;
        case '-':       z = x - y;
                break;
        case '*':       z = x * y;
                break;
        case '/':
                if(y != 0)
                        z = x / y;
                else
                        ok = false;
                break;
        default:
                ok = false;
                break;
}
```

Правила использования оператора switch сводятся к следующему:

1. Switch требует совместимого с целым значения. Это значение может быть константой, переменной, вызовом функции или выражением. Оператор switch не работает с типами данных с плавающей точкой.
2. Значение после каждой метки case должно быть константой.
3. C++ не поддерживает метки case с диапазоном значений. В этом случае каждое значение из диапазона должно появляться с отдельной меткой case.
4. Окончание оператора case обычно отмечается словом break. Это вызывает переход к выполнению первого оператора, который следует после switch. Если вы не включаете break, то выполнение будет продолжаться со следующего оператора case. Это действительно иногда полезно, но в большинстве случаев вам необходимо использовать break. Как альтернативу break можно использовать оператор return. Это вызовет завершение работы текущей функции, если текущая функция – main, то произойдет завершение программы.
5. Предложение default – всеохватывающее, но оно не обязательно, если вы хотите проверить только ряд случаев.
6. Ряд операторов в каждой метке case или в групповых метках case можно не заключать в фигурные скобки.

Отсутствие возможности использования единственной метки case для диапазона значений делает более привлекательным использование многоальтернативного оператора if-else в случае, когда вы имеете большой непрерывный диапазон значений.

На рисунке 4 показана диаграмма выполнения оператора switch.

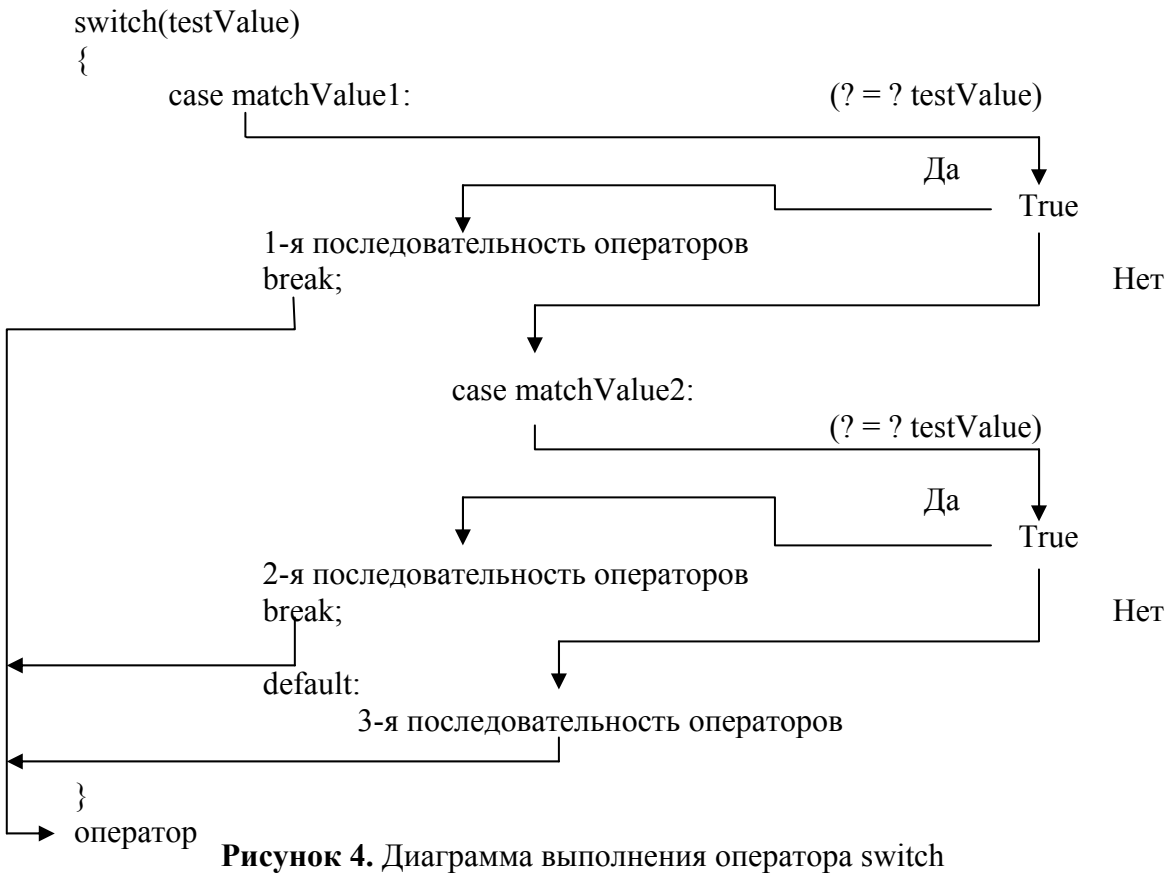

Рисунок 4. Диаграмма выполнения оператора switch

Самостоятельно изучите листинг 4. В нем представлено решение задачи классификации введенного пользователем символа с использованием оператора switch.

```
//Программа, демонстрирующая оператор switch
#include <iostream.h>
int main()
{
        char c;
        cout << “Введите символ: ”;
        cin >> c;
        switch(c)
        {
                case ‘A’:
                case ‘B’;
                case ‘C’:
                case ‘D’:
        …
```

```
            case ‘Z’:
                cout << “Вы ввели прописную букву” << endl;
                break;
            case ‘a’:
            case ‘b’:
            case ‘c’:
            case ‘d’:
        …
            case ‘z’:
                cout << “Вы ввели строчную букву” << endl;
                break;
            case ‘0’:
            case ‘1’:
        …
            case ‘9’:
                cout << “Вы ввели цифру” << endl;
                break;
            default:
            cout << “Вы ввели не буквенно-цифровой символ” << endl;
                break;
        }
        return 0;
}
```

Листинг 4.

2. Операторы циклов

Циклы применяются для того, чтобы C++ мог автоматически выполнять некоторые действия многократно.

Цикл for

Цикл for в C++ отличается большой гибкостью, поскольку с его помощью можно организовывать как фиксированные, так и условные итерации. Последнее свойство цикла for отличается от его типового использования в других языках программирования, таких как Pascal или BASIC. В то время как эти языки позволяют вам использовать исключительно целые числа и иметь единственные точки начала и конца цикла, C++ предоставляет возможности спецификации более сложных операций.

Общий синтаксис оператора цикла for имеет вид:

for(<инициализация переменных управления циклом>;
<проверка продолжения цикла>;
<модификация переменных управления циклом, часто их приращение или уменьшение>)

Пример

```
for (i =0; i < 10; i++)
cout << “Текущее значение i равно ” << i << “.” << endl;
```

Оператор цикла for имеет три компонента, каждый из них необязателен. Первый компонент инициализирует переменные управления циклом (C++ предоставляет вам возможность использовать более одной переменной управления циклом). Второй компонент цикла – это условие, которое определяет, будет ли цикл выполнять следующую итерацию (нечто вроде оператора if, но без самого ключевого слова if). Последний компонент цикла for – предложение, которое изменяет переменные управления циклом, часто это просто операция инкремента и/или декремента.

Цикл for в C++ дает вам возможность объявлять переменные управления циклом. Такие переменные существуют только в области действия цикла. Например:

```
int i = 5;
cout << “Начальное значение i ” << i << endl;
for (int i =0; i < 10; i++)
cout << “Шаг №” << i +1 << endl;
cout << “Конечное значение i” << i << endl;
```

Результат этого фрагмента кода следующий:

```
Начальное значение i 5
Шаг № 1
Шаг № 2
Шаг № 3
Шаг № 4
Шаг № 5
Шаг № 6
Шаг № 7
Шаг № 8
Шаг № 9
Шаг № 10
Конечное значение i 5
```

Рассмотрим работу программы из листинга 5. Программа предлагает вам определить диапазон целых чисел путем спецификации нижней и верхней границ. Затем программа вычисляет сумму чисел, а также среднее значение в указанном вами диапазоне.

```
// Программа рассчитывает сумму и среднее значение
// ряда целых чисел, используя цикл for
#include <iostream.h>
int main ( )
{
    int count = 0;
    double sum = 0.0;
    int first, last, temp;
    cout << “Введите первое целое число: ";
```

```
        cin >> first;
        cout << “Введите последнее целое число: ”;
        cin >> last;
        if(first > last)
        {
                temp = first;
                first = last;
                last = temp;
        }
        for(int i = first; i <= last; i++)
        {
                count++;
                sum += (double)i;
        }
        cout << “Сумма целых чисел от ”
        << first << “ до ” << last << “ = ”
        << sum << endl;
        cout << “Среднее значение = ” << sum / count << endl;
        return 0;
}
```

Листинг 5.

Вот пример сеанса работы программы в листинге 5:

Введите первое целое число: 1

Введите последнее целое число: 100

Сумма целых чисел от 1 до 100 = 5050

Среднее значение = 50.5

Программа в листинге 15 объявляет набор переменных типа int и double в функции main. Функция инициирует переменные суммирования sum и count нулевыми значениями. Входной и выходной операторы предлагают вам ввести целые, определяющие диапазон значений. Программа запоминает эти целые в переменных first и last. Оператор if в строке определяет, больше ли значение переменной first, чем значение переменной last. Если это условие есть true, то программа выполняет группу операторов в строках с 15 по 17. Эти операторы обменивают значения переменных first и last, используя переменную temp как буфер обмена. Таким образом, оператор if гарантирует, что число в переменной first меньше или равно числу в переменной last.

Программа выполняет суммирование, используя оператор цикла for. Цикл объявляет собственную переменную управления, i, и инициирует ее значением переменной first. Условием продолжения цикла является i <= last. Это условие означает, что итерации цикла продолжаются, пока i меньше или равно значению переменной last. Компонент инкремента цикла i++ увеличивает переменную управления цикла на 1 в конце каждой итерации. Цикл содержит два оператора. Первый оператор увеличивает на 1 значение

переменной count. Второй оператор прибавляет значение i (после приведения типа к double) к переменной sum.

Вы можете переписать цикл for, переместив первый оператор цикла в компонент инкремента цикла:

```
for( int i = first; i <= last; i++, count++)
        sum += (double)i;
```

Отметим использование запятой в последней части оператора for. Это наиболее типичное место, в котором использована особенность C++, допускающая запись нескольких операторов там, где обычно разрешен только один. Хотя это не имеет никакого значения для нашего примера, отметим все же, что значение выражения (i++, count++) такое же, как просто count++.

В конце программы выходной оператор отображает сумму и среднее значение чисел в указанном вами диапазоне.

Для иллюстрации гибкости цикла for модифицируем программу из листинга 5.

```
// Программа рассчитывает сумму и среднее значение
// ряда целых чисел, используя цикл for
#include <iostream.h>
int main()
{
        int count = 0;
        double sum = 0.0;
        int first, last, temp;
        cout << “Введите первое целое число: ”;
        cin >> first;
        cout << “Введите последнее целое число: ”;
        cin >> last;
        if(first > last)
        {
                temp = first;
                first = last;
                last = temp;
        }
        cout << “Сумма целых чисел от ”
                << first << “ до ” << last << “ = ”;
        for(; first <= last;)
        {
                count++;
                sum += (double)first++;
        }
        cout << sum << endl;
        cout << “Среднее значение = ” << sum / count << endl;
        return 0;
}
```

Листинг 6.

Вот пример сеанса работы программы из листинга 6:

```
Введите первое целое число: 1
Введите последнее целое число: 100
Сумма целых чисел от 1 до 100 = 5050
Среднее значение = 50.5
```

Программы представленные в листингах 5 и 6 выполняют одинаковые задачи и одинаково взаимодействуют с пользователем. Изменения сделаны лишь в циклах for. Вместо заранее определенной переменной управления используется first, переменная, которая отмечает начало цикла. Поскольку в цикле предполагается ее изменение, прежде чем будет произведена любая ее модификация, перед циклом выводится пользователю ее значение. Затем цикл for использует first вместо i. Отметим, что раз first инициализирована как начальная точка, нет никакой необходимости делать это снова в первом выражении for. Кроме того, удален компонент инкремента цикла который компенсирован применением постинкрементной операции к переменной first.

Открытые циклы, использующие цикл for

Когда вы знакомились с циклом for в C++, вы узнали, что три компонента этого цикла необязательны. Более того, в C++ допускается, что все три компонента могут быть пустыми.

Если вы оставите все три компонента цикла пустыми, результатом будет открытый цикл (open loop).

Стоит указать, что и другие языки, такие как ADA или Modula, поддерживают открытые циклы и механизмы выхода из них. C++ позволяет вам выходить из цикла следующими четырьмя способами:

- Оператор break вызывает переход к выполнению кода, следующего за текущим циклом, во многом подобно тому, как он мог бы быть использован, чтобы продолжить выполнение вне оператора switch. Используйте оператор break, если вы хотите выйти из цикла и продолжить работу оставшейся части программы.
- Оператор return осуществляет возврат из текущей функции, включая main.
- Оператор throw вызывает генерацию исключения. Это используется, если произошла ошибка и вы не можете продолжать выполнение оставшейся части программы без какого-либо обработчика ошибок. Однако применяйте этот метод с осторожностью, исключения, строго говоря, предназначены для использования в нештатных обстоятельствах – в случае ошибок, например.
- В самых чрезвычайных случаях выйти из программы можно с помощью функции exit (объявленной в заголовочном файле STDLIB.H). Используйте функцию exit только в случае крайней необходимости, когда не остается никакой надежды на восстановление

работоспособности программы после ошибки. Функция exit прекратит выполнение итераций и приведет к выходу из программы.

Рассмотрим листинг 7, который содержит исходный код программы использующей открытый цикл, снова и снова предлагающий ввести число. Программа воспринимает данные и отображает число и его обратное значение. Затем программа спрашивает, хотите ли вы рассчитать обратное значение другого числа. Если вы печатаете букву Y или y, программа выполняет следующую итерацию. В противном случае программа завершает работу. Если вы продолжаете печатать Y или y для последующих запросов, программа будет продолжать работать – по меньшей мере до тех пор, пока компьютер не сломается!

```
// Программа, демонстрирующая использование цикла for
// для эмуляции неопределенного цикла
#include <iostream.h>
#include <ctype.h>
int main()
{
	char ch;
	double x, y;
	for(;;) // цикл for с пустыми компонентами
	{
		cout << endl << “Введите число: ”;
		cin >> x;
		if (x != 0)
		{
			y=1/x;
			cout << “1/” << x << “ = ” << y << endl;
			cout << “Продолжаем? (Y/N) ”;
			cin >> ch;
			if (toupper(ch) != ' Y')
			break;
		}
		else
			cout << “Число не должно быть равно 0!” << endl;
	}
	return 0;
}
```

Листинг 7.

Рассмотрим пример сеанса работы программы, представленной в листинге 7.

Введите число: 5
1/5 = 0.2
Продолжаем? (Y/N) y
Введите число: 12

1/12 = 0.0833333

Продолжаем? (Y/N) n

Программа в листинге 7 объявляет переменную ch типа char и две переменных, x и y, типа double. Функция main использует цикл for как открытый цикл путем исключения всех трех компонент. Выходной оператор предлагает вам ввести число. Входной оператор получает ваши данные и запоминает их в переменной x. Оператор if-else проверяет значение переменной x на неравенство нулю. Если его условие имеет значение true, программа выполняет группу операторов в строках с 15 по 20.

В противном случае программа выполняет оператор предложения else который отображает сообщение об ошибке.

Оператор в строке 15 присваивает переменной y значение, обратное значению переменной x. Выходной оператор в строке 16 отображает значения переменных x и y. Выходной оператор в строке 17 предлагает вам продолжить вычисления (Y/N), напечатав прописной или строчной буквой ваше решение. Входной оператор в строке 18 запоминает ваш односимвольный ответ в переменной ch. Оператор в строке 19 преобразует его в прописную букву, используя функцию toupper (прототип которой находится в заголовочном файле CTYPE.H) и проверяет неравенство символа букве Y. Если условие имеет значение true, программа выполняет оператор break в строке 20. Этот оператор вызывает программный выход из открытого цикла и продолжение работы программы со строки 25.

Цикл do-while

Цикл do-while в C++ – это условный цикл, который проверяет итерационное условие в конце цикла. Поэтому в цикле do-while всегда выполняется по крайней мере одна итерация. Цикл do-while называется циклом с постусловием или постфиксным циклом.

Условный цикл выполняется до тех пор, пока условие имеет значение true. Это условие проверяется в конце цикла в цикле do-while и в начале обычного цикла while.

Цикл do-while имеет следующий общий синтаксис:

```
do {
        < последовательность операторов>
        } while (условие);
```

Пример

Нижеследующий цикл возводит в квадрат числа от 2 до 10:

```
int i=2;
do
{
        cout << i << " ^ 2 = " << i * i << endl;
} while (++i < 11 );
```

В листинге 8 показан исходный код программы использующий цикл do-while, которая по существу не отличается от программы из листинга 7, за

исключением того, что неудобный открытый цикл for заменен более подходящим циклом do-while.

```
// Программа, демонстрирующая использование
// цикла do-while
#include <iostream.h>
#include <ctype.h>
int main ()
{
        char ch;
        double x, y;
        do // цикл do-while выполняет вычисления
        {
                do // цикл do-while собирает числа
                {
                        cout << endl << “Введите число: ”;
                        cin >> x;
                        if (x == 0)
                        cout << “Число не должно быть равно 0!” << endl;
                } while (x == 0);
                y = 1 / x;
                cout << “1/” << x << “ = ” << y << endl;
                cout << “Продолжаем? (Y/N) ”;
                cin >> ch;
        } while (toupper(ch) == 'Y');
        return 0;
}
```

Листинг 8.

Различие между программами из листингов 7 и 8 довольно мало и состоит в том, что цикл for заменены циклом do-while.

Прежде всего, внешний цикл изменен так, что он работает уже не как открытый цикл for. Хотя открытые циклы имеют свою область применений, но она весьма узка; данное решение гораздо более элегантно. Цикл начинается в строке 9 с оператора do, а заканчивается в строке 22 оператором while. Заметим, что этот оператор while заменяет оператор if в предыдущей программе, который вызвал бы выполнение оператора break в случае ввода пользователем букв, отличных от y или Y.

Кроме того, в программе имеется внутренний цикл do-while в строках с 11 по 17, который продолжает запрашивать у пользователя число до тех пор, пока оно не окажется равным нулю.

Цикл while

Цикл while в C++ – второй условный цикл, в котором итерации выполняются до тех пор, пока условие имеет значение true. Таким образом,

цикл while может не выполнить ни одной итерации, если проверяемое условие изначально имеет значение false. Цикл while называется циклом с предусловием или префиксным циклом.

Общий синтаксис цикла while имеет вид:

while (условие)

оператор; | { последовательность операторов }

Пример

// Вычисляет x в степени n double pwr = 1; while (n --> 0)

pwr *= x; cout << x << " ^ " << pwr << endl;

В листинге 9 представлен исходный код программы с использованием цикла while. Эта программа выполняет те же операции, что и программа, показанная в листинге 6. Эти две программы взаимодействуют с пользователем одинаковым образом и приводят к одинаковым результатам.

```
// Программа рассчитывает сумму и среднее значение
// ряда целых чисел, используя цикл while
#include <iostream.h>
int main()
{
	int count = 0;
	double sum = 0.0;
	int first, last, temp;
	cout << “Введите первое целое число: ”;
	cin >> first;
	cout << “Введите последнее целое число: ”;
	cin >> last;
	if(first > last)
	{
		temp = first;
		first = last;
		last = temp;
	}
	cout << "Сумма целых чисел от" << first << " до " << last << " = ";
	while (first <= last)
	{
		count++;
		sum += (double)first++;
	}
	cout << sum << endl;
	cout << "Среднее значение = " << sum / count << endl;
	return 0;
}
```

Листинг 9.

Единственное различие между листингами 9 и 6 заключается в строке, где цикл for заменен циклом while. Если единственной вещью, ради которой вы собираетесь применить цикл for, является условное выражение, то, похоже, вам на самом деле нужен цикл while.

Пропуск итераций цикла

C++ предоставляет вам возможность сразу переходить к концу цикла и переходить к следующей итерации, используя оператор continue. Эта особенность позволяет вашему циклу пропускать итерации для отдельных значений, которые могут вызвать ошибки времени исполнения.

Общая форма использования оператора continue имеет вид:

```
<предложение начала цикла>
{
// последовательность операторов №1
        if (условие пропуска)
                continue;
        // последовательность операторов №2
} <предложение конца цикла>
```

Пример

```
double x, y;
for (int i = -10; i < 11; i++)
{
        x = i;
        if(i == 0)
                continue;
        y = l/x;
        cout << “1/” << x << “ = ” << y << endl;
}
```

Эта форма показывает, что оценка первой последовательности операторов в цикле for инициирует условие, проверяемое в операторе if. Если это условие имеет значение true, то оператор if выполняет оператор continue, чтобы пропустить вторую последовательность операторов в цикле for.

В листинге 10 представлен исходный код программы использующей оператор continue. Программа просто выполняет короткий цикл for, отображая по очереди каждое число. Чтобы продемонстрировать оператор continue в действии, средняя часть списка пропущена.

```
// Программа, демонстрирующая использование оператора
// continue для пропуска итераций.
#include <iostream.h>
# include <match.h>
int main ( )
{
        for(int i = 0; i < 10; ++i)
        {
```

```
        if(i >= 4 && i <= 6)
                continue;
        cout << "Шаг № " << i << endl;
    }
    return 0;
}
```

Листинг 10.

Вот пример сеанса работы программы в листинге 20:

Шаг № 0
Шаг № 1
Шаг № 2
Шаг № 3
Шаг № 7
Шаг № 8
Шаг № 9

Программа в листинге 10 довольно самоочевидна. Строка 7 инициирует цикл for, выполняемый от 0 до 9. Строка 9 проверяет, находится ли номер текущей итерации между 4 и 6, и если это так, то исполняется оператор continue в строке 10. Наконец, строка 11 выводит пользователю номер текущей итерации. Заметим, что в примере выходных данных номера между 3 и 7 пропущены.

Выход из цикла

C++ поддерживает оператор break для выхода из цикла. Оператор break вызывает переход к выполнению кода, следующего за текущим циклом, подобно тому как это делается в операторе switch.

Оператор break

Общая форма использования оператора break в цикле имеет вид:

```
<предложение начала цикла>
 {
        // последовательность операторов №1
        if (<условие выхода из цикла>)
                break;
        // последовательность операторов №2
} <предложение конца цикла>
// последовательность операторов № 3
```

Пример

```
// расчет факториала числа n
factorial = 1;
for (int i = 1; i++)
{
        if ( i > n )
                break;
```

```
        factorial *= (double) i;
}
```

Эта форма означает, что оценка первой последовательности операторов в цикле for инициирует условие, проверяемое в операторе if. Если это условие имеет значение true, то оператор if вызывает оператор break, чтобы совсем выйти из цикла. Далее программа продолжает работу с третьей последовательностью операторов.

В качестве хорошего примера, иллюстрирующего применение оператора break, необходимо еще раз проанализировать программу представленную в листинге 7.

Вложенные циклы

Вложенные циклы позволяют вам представлять повторяющиеся задачи как часть других повторяющихся задач. C++ допускает вложение цикла любого типа на любую требуемую глубину. Вложенные циклы часто используются для обработки массивов. В качестве хорошего примера, иллюстрирующего применение вложенных циклов, рассмотрите и проанализируйте программу представленную в листинге 8.

3. Структуры данных и их реализация

Создание определяемых пользователем типов данных является одной из необходимых особенностей современных языков программирования. Для определения новых типов данных как псевдонимов существующих типов в C++ имеется ключевое слово typedef.

Общий синтаксис использования typedef имеет вид:

typedef известныйТип новыйТип;

Пример:

typedef unsigned word;

typedef unsigned char byte;

Ключевое слово typedef определяет новый тип, исходя из уже существующего. Typedef можно использовать для создания псевдонимов, укорачивающих имена существующих типов данных, или для определения имен типов данных, более привычных для вас или лучше описывающих способ их использования. Такое применение typedef иллюстрируется во втором из вышеприведенных примеров. Можно использовать typedef и для определения типа массива.

Общий синтаксис определения имени типа массива таков:

typedef базовыйТип имяТипаМассива[размерМассива];

Оператор typedef определяет имяТипаМассива, для которого базовый тип и размер равны базовыйТип и размерМассива соответственно.

Примеры

typedef double vector[10];

typedef double matrix[10][30];

Таким образом, идентификаторы vector и matrix становятся именами типа данных.

В известном смысле вы можете воспринимать typedef как особый вид макроса C++, подобный препроцессорной директиве #define. Различие состоит в том, что макрос указывает C++ просто заменить один текст на другой, тогда как typedef создает совершенно новый тип, который можно использовать наряду с любыми другими типами. Рассмотрим следующий фрагмент кода:

```
unsigned char aChar;
typedef unsigned char byte;
byte aByte;
```

Фрагмент начинается объявлением переменной aChar типа unsigned char. Вы это уже видели раньше и это должно быть вам знакомо. Вторая строка создает новый тип с именем byte. Теперь напомним, что это не синоним для unsigned char, а скорее совершенно новый тип – как если бы компилятор сам включил byte в свой ассортимент базовых типов. Третья строка затем объявляет переменную abyte типа byte, как если бы byte всегда был частью C++. Необходимо подчеркнуть, что aChar и aByte – совершенно различные типы. Так, переменная aByte – типа byte, который был ранее определен как тип unsigned char, а переменная aChar – типа unsigned char, и эти две переменные относятся к совершенно разным типам.

Перечислимые типы данных

Правило работы с перечислимыми типами данных состоит в том, что хотя перечисляемые идентификаторы должны быть уникальными, присваиваемые им значения могут уникальными и не быть.

Перечислимый тип определяет список уникальных идентификаторов и ассоциирует с ними определенные значения.

Общий синтаксис объявления перечислимого типа имеет вид:

```
enum перечислимыйТип {<список перечисляемых идентификаторов>};
```

Примеры

```
enum YesNo { no, yes, dontCare, maybe };
enum weekday { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

Вот еще один пример объявления перечислимого типа:

```
enum CPUtype { i8088, i80286, i80386, i80486, i80586 };
```

C++ ассоциирует с перечисляемыми идентификаторами целочисленные значения. Например, в последнем примере компилятор присваивает значение 0 идентификатору i8088, значение 1 идентификатору i80286 и т.д.

C++ весьма гибок в отношении объявления перечислимого типа. В первую очередь, язык предоставляет возможность явно присваивать значение перечисляемому идентификатору, например:

```
enum weekday { Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

Это объявление явно присваивает значение 1 идентификатору Sunday. Компилятор затем присваивает следующее целое значение 2 следующему идентификатору Monday и т.д. C++ позволяет явно присвоить значение

каждому элементу перечисляемого списка. Более того, эти значения могут быть не уникальными. Вот некоторые примеры возможных объявлений перечислимых типов в C++:

```
// присваивание значений каждому элементу списка
enum colors { black = 1, red = 2, blue = 3, green = 5, yellow = 7, white = 11 };
// присваивание нерегулярных значений
enum colors { black = 1, red, blue, green = 5, yellow = 7, white = 11 };
// повторяющиеся значения
enum CPUtype { i8088 = 0, i80286 = 2, i80386DX = 3, i80386SX = 3 i80486 = 4,
i80486SX = 4 };
enum choiceType { false, true, dontCare = 0 };
```

В последнем примере компилятор ассоциирует идентификатор false с 0 по умолчанию. Однако компилятор ассоциирует также значение 0 с dontCare вследствие явного определения.

C++ позволяет объявлять переменные перечислимого типа следующим образом:

- Объявление перечислимого типа может включать объявление переменных этого типа. Общий синтаксис таков:

  enum перечислимыйТип {<список перечисляемых идентификаторов>} <список переменных>;

  Пример:

  enum weekDay { Sun = l, Mon, Wed, Thu, Fri, Sat } recycleDay, payDay, movieDay;

- Отдельное объявление перечислимого типа и его переменных включает множества операторов для раздельного объявления типа и ассоциированных с ним переменных. Общий синтаксис таков:

  enum перечислимыйТип {<список перечислимых идентификаторов>};
  перечислимыйТип varl, var2, ...,varN;

Листинг 11 показывает исходный код программы реализующей простейший однострочный калькулятор с четырьмя действиями, который выполняет следующие задачи:

1. Предлагает вам ввести первое число, операцию (+, -, * или /) и второе число.
2. Выполняет требуемое действие, если все верно.
3. Отображает операнды, операцию и результат, если операция выполнена правильно; в противном случае выводит сообщение об ошибке, которое определяет тип ошибки. (Вы либо ввели непредусмотренную операцию, либо попытались разделить на 0).

```
// Программа на C++, демонстрирующая перечислимые типы
#include <iostream.h>
enum mathError { noError, badOperator, divideByZero };
int main()
{
```

```
        double x, y, z;
        char op;
        mathError error = noError;
        cout << “Введите число действие число: ”;
        cin >> x >> op >> y;
        switch (op)
        {
                case ‘+’: z = x + y;
                        break;
                case ‘-’: z = x - y;
                        break;
                case ‘*’: z = x * y;
                        break;
                case ‘/’:
                        if (y != 0)
                                z = x / y;
                        else
                                error = divideByZero;
                        break;
                default:
                        error = badOperator;
                        break;
        }
        if (error == noError)
                cout << x << “  ” << op << “  ” y << “ = ” << z << endl;
        else
        switch (error)
        {
                case noError: cout << “Ошибок нет ” << endl;
                        break;
case badOperator: cout << “Ошибка: недопустимое действие ” << endl;
                        break;
case divideByZero: cout << “Ошибка: деление на ноль” << endl;
                break;
        }
        return 0;
}
```

Листинг 11.

Программа объявляет перечислимый тип mathError в строке. Этот тип данных имеет три перечисляемых значения: noError, badOperator и divideByZero.

Функция main объявляет тип double для переменных x, y и z, представляющих соответственно два операнда и результат. Кроме того, функция объявляет тип char для переменной op, хранящей требуемую операцию, и перечислимую переменную error, хранящую статус ошибки. Функция инициализирует переменную error значением noError.

Оператор вывода в предлагает вам ввести операнды и оператор. Оператор ввода запоминает введенные данные в переменных x, op и y в том же порядке. Функция использует оператор switch, чтобы проанализировать значение переменной op и выполнить требуемую операцию. Метки case определяют значения для четырех поддерживаемых арифметических действий. Последняя метка case содержит оператор if, который обнаруживает попытку деления на 0. В случае true оператор предложения else присваивает переменной error значение divideByZero.

Предложение default обрабатывает неверные операции и присваивает переменной error значение badOperator.

Оператор if определяет, содержит ли переменная error значение noError. Если это так, то программа выполняет оператор вывода, который отображает операнды, арифметическое действие и результат. В противном случае программа выполняет предложение else – оператор switch, который определяет характер ошибки и отображает соответствующее сообщение.

Объединения

Размер объединения равен размеру его наибольшего элемента.

Объединение – это специальная структура, которая хранит элементы в разделяемом адресном пространстве.

Общий синтаксис для объединения таков:

```
union меткаОбъединения
{
        тип1 элемент1;
        тип2 элемент2;
        …
        типN элементN;
};
```

Пример

```
union Long
{
        struct
        {
                unsigned short lo;
                unsigned short hi;
        } w;
        long l;
};
Long 1;
l.l = 0x12345678L;
```

```
cout << hex << l.l << endl;
cout << hex << l.w.lo << endl;
cout << hex << l.w.hi << endl;
```

Объединение Long хранит либо структуру w (которая содержит два коротких двухбайтовых целых числа без знака), либо длинное целое четырехбайтовое число. Кроме того, объединение Long позволяет обращаться к нижнему или верхнему словам (двухбайтовым целым) длинного целого числа. Предыдущий фрагмент кода выдает следующий результат:

```
12345678
5678
1234
```

Объединения обеспечивают простой способ быстрого преобразования данных. Объединения имели гораздо большее значение в недавнем прошлом, когда цена компьютерной памяти была много выше и объединения было целесообразно использовать для мобилизации ресурсов памяти. Доступ к элементам объединения, так же как и в структурах, осуществляется посредством операции-точки.

Ссылочные переменные

C++ поддерживает ссылочные переменные. Применяя ссылки, можно обращаться к переменным, используя их псевдонимы. Если вы начинающий программист, ваше использование ссылочных переменных будет скорее всего ограниченным. По мере освоения C++ вы увидите, как ссылки позволяют реализовать различные программные приемы, используемые при разработке сложных классов.

Ссылочные переменные – это псевдонимы переменных, к которым они обращаются.

Объявление ссылочных переменных имеет следующий синтаксис:

```
тип& ссылПер (переменная);
тип& ссылПер = переменная;
```

ССылПер – это ссылочная переменная, которая инициализируется после того, как она объявлена. Перед использованием ссылочной переменной вам необходимо удостовериться в том, что она инициализирована или ей присвоено значение.

Примеры

```
int x = 10, y = 3;
int& rx(x);
int& ry = y; // взять ссылку
```

Рассмотрим простой пример, демонстрирующий ссылочную переменную в действии. В листинге 12 показан исходный код программы использующей как саму переменную, так и ссылку.

```
// Программа на C++, демонстрирующая ссылочные переменные
#include <iostream.h>
```

```
int main()
{
        int x = 10;
        int& rx = x;
        // отобразить x, используя x и rx
        cout << “x содержит ” << x << endl;
        cout << “x содержит (используется ссылка rx) ” << rx << endl;
        // изменить x и отобразить его значение, используя rx
        x *= 2;
        cout << “x содержит (используется ссылка rx)” << rx << endl;
        //изменить rx и отобразить его значение, используя x
        rx *= 2;
        cout << “x содержит ” << x << endl;
        return 0;
}
```

Листинг 12.

Результат работы программы:
x содержит 10
x содержит (используется ссылка rx) 10
x содержит (используется ссылка rx) 20
x содержит 40

Программа в листинге 12 объявляет переменную x типа int и ссылочную переменную rx типа int. Программа инициализирует переменную x значением 10 и ссылочную переменную rx переменной x.

Оператор в строке 8 выводит значение переменной x, используя саму переменную x. В противоположность этому оператор в строках 9 и 12 отображает значение переменной x, используя ссылочную переменную rx.

Оператор в строке 11 удваивает значение переменной x.

Оператор в строке 12 выводит новое значение переменной x, используя ссылочную переменную rx. Как показывают выходные данные, ссылочная переменная точно отображает обновленное значение переменной x.

Оператор в строке 14 удваивает значение переменной x, используя ссылочную переменную rx. Выходной оператор в строке 15 отображает обновленное значение переменной x, используя переменную x. И снова выходные данные показывают, что переменная x и ссылочная переменная rx согласованы.

Указатели

Каждая частица информации, будь то код или данные, в компьютерной памяти находится по определенному адресу и занимает определенное количество байт. При выполнении программы ваши переменные имеют определенные адреса. При работе с языками высокого уровня, такими, как C++, вы не заботитесь об истинных адресах ваших переменных. Эта задача

невидимым для вас образом решается компилятором и исполнительной системой C++. Логически каждая переменная в вашей программе играет роль этикетки адреса памяти. Манипуляция данными с использованием таких "этикеток" много легче работы с действительными адресами, такими, например, как 0F63:01AF4.

Адрес – это местоположение ячейки памяти. ***Этикетка адреса*** – это имя переменной. Представьте себе, что память компьютера – это большое количество крошечных ящичков. Теперь представьте себе, что когда вы объявляете переменную, вы на самом деле отставляете в сторону ящик для хранения значения этой переменной. Но что если вы, не интересуясь содержимым ящика, захотели бы узнать, где этот ящик находится в памяти? Вы бы создали указатель на этот ящик. Создание указателя – это просто объявление еще одной переменной, но значение, которое может быть найдено в ящике указателя, в действительности является адресом ящика другой переменной. Рассмотрите рисунок 5, соотнося его со следующим фрагментом кода:

```
int myInt = 42;
int *pInt = &myInt;
```

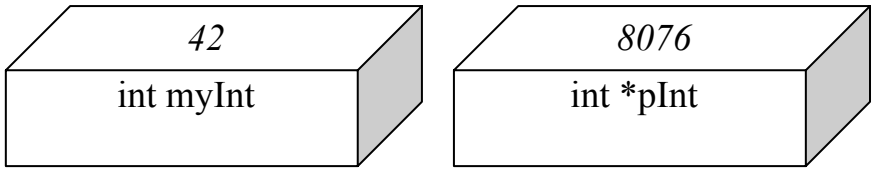

Рисунок 5.

Была объявлена переменная myInt; тем самым для нее был создан ящик, и в нем было размещено значение 42. Допустим, что этот ящик имеет в памяти адрес 8076. Когда был объявлен указатель pInt, был создан другой ящик с другим адресом памяти. Пусть в данном частном случае этот адрес оказался равным 8094. Важно отметить, что адреса памяти вовсе не обязательно должны следовать друг за другом, как видно из нашего примера. Когда был объявлен указатель pInt, он был инициализирован адресом (вот что означает &) переменной myInt. Итак, адрес памяти, в котором находится переменная myInt, был помещен в ящик pInt. А pInt сама является переменной, но ее значение оказывается адресом другой переменной. Заметим, что в этот момент вполне возможно объявить еще один ящик в качестве указателя на pInt; это называлось бы указателем на указатель.

C++ и его предок C – это языки программирования, которые используются также для программирования систем на низком уровне. На самом деле многие программисты считают C языком ассемблера высокого уровня. В программировании систем на низком уровне часто приходится работать с адресами данных. Именно здесь, вообще говоря, и вступают в

игру указатели. Знание адреса элемента данных позволяет устанавливать и опрашивать его значение.

Указатель — это специальная переменная, которая хранит адрес другой переменной или иной информации.

Указатели – мощный компонент языка. Но они могут быть опасны при небрежном обращении с ними, потому, что могут вызывать зависание системы. Эта неприятность происходит, если указатель не был установлен. Не забывайте, что объявление какого-либо объекта еще не присваивает ему никакого значения. До тех пор, пока вы не установите значение указателя адресом другой переменной, ваш указатель относится к чему-то случайному в памяти и его использование может привести к фатальным результатам.

Указатели на существующие переменные

C++ требует, чтобы вы ассоциировали с объявленным указателем определенный тип данных, возможно, void. Ассоциированный тип данных может быть предопределенным типом или структурой, определяемой пользователем.

Общий синтаксис объявления указателя имеет вид:

тип* имяУказателя;

тип* имяУказателя = переменнаяУказатель;

тип* имяУказателя = &переменная;

Операция & – это операция взятия адреса (это не операция ссылки, которая тоже использует символ &), предназначенная для получения адреса переменной. Операция взятия адреса возвращает адрес переменной, структуры, функции и т. д. В противоположность этому операция ссылки создает псевдоним переменной.

Пример

```
int* intPtr; // указатель на int
double* realPtr; // указатель на double
char* aString; // указатель на char
long lv;
long* lp = &lv;
```

Вы можете также объявлять обычные переменные в тех же самых строках, где объявляются указатели.

```
int *intPtr, anint;
double x, *realPtr;
char *aString, aKey;
```

C++ допускает размещение символа звездочки непосредственно справа от ассоциированного типа данных. Этот способ синтаксиса не нужно интерпретировать как обозначение того, что всякий другой идентификатор, появляющийся в том же самом объявлении, автоматически становится указателем.

Инициализируйте указатель перед его использованием, так же как обычную переменную. В действительности инициализация указателей даже более необходима, так как использование неинициализированных указателей

может привести к непредсказуемому поведению программы и даже к зависанию системы. Всякий раз, когда вы встречаетесь в программе с общим нарушением защиты (General Protection Fault – GPF), оно обычно вызвано использованием указателей, которые не были соответствующим образом инициализированы. Не думайте, что неинициализированные указатели – безобидная вещь!

Если указатель содержит адрес переменной, вы можете получить значение этой переменной, используя операцию *, за которой следует имя указателя. Например, если px – указатель на переменную x, то можно использовать *px для доступа к значению переменной x.

Ставьте операцию * слева от указателя, чтобы получить доступ к переменной, адрес которой хранится в указателе.

Не забывайте использовать операцию *. Без этого оператор будет обращаться к адресу в указателе вместо данных, находящихся по этому адресу.

Рассмотрим пример, иллюстрирующий указатель в действии. Листинг 13 содержит исходный код программы которая отображает и изменяет значения переменных, используя либо сами переменные, либо их указатели.

```
// Программа на C++, демонстрирующая указатели
#include <iostream.h>
int main()
{
	int x = 10;
	int *px = x;
	// отобразить x, используя x и px
	cout << “x содержит ” << x << endl;
	cout << “x содержит (используется указатель px) ” << px << endl;
	// изменить x и отобразить его значение, используя px
	x *= 2;
	cout << “x содержит (используется указатель px)” << px << endl;
	//изменить px и отобразить его значение, используя x
	px *= 2;
	cout << “x содержит ” << x << endl;
	return 0;
}
```

Листинг 13.

Программа идентична программе в листинге 12, только вместо ссылочной переменной, используется указатель.

4. Понятие массива данных. Задачи обработки массивов

Массив – один из наиболее полезных компонентов языка программирования. Он позволяет программисту хранить в одном месте несколько элементов данных. Простейший вид массива – одномерный массив. В таком массиве к каждой переменной можно обратиться индивидуально, используя единственный индекс.

Массив – это группа значений, использующих общее имя (имя массива) и хранящихся в последовательных ячейках памяти.

Общий синтаксис объявления одномерного массива имеет вид:

тип имяМассива[количествоЭлементов];

Примеры

```
int intArray[10];
char name[31];
double x[100];
```

Число в квадратных скобках обозначает количество элементов в массиве. Для обращения к отдельному элементу массива используется простая конструкция: имя массива, за которым следует индексное значение в квадратных скобках.

C++ требует от вас соблюдения определенных правил при объявлении одномерных массивов:

1. Нижняя граница индекса массива в C++ устанавливается равной 0. В C++ не разрешается переопределять или изменять это нижнее значение.
2. Объявление массива в C++ включает в себя спецификацию количества его элементов. Постоянно имейте в виду, что количество элементов равно верхней границе плюс один.

Правильный диапазон индексов для этой формы массива простирается от 0 до (количествоЭлементов - 1).

Массив можно представить в виде ряда ящиков с последовательными адресами. Специфицируя индекс, вы указываете, к какому из ящиков массива вы обращаетесь. 0 означает обращение к самому первому ящику, 1 – к следующему и т. д.

Использование одномерного массива требует указания, как имени массива, так и правильного индекса для доступа к одному из его элементов. В зависимости от того, где находится ссылка на элемент массива, она может либо сохранять, либо извлекать его значение. Следует помнить простые правила:

- Присваивайте значение элементу массива до того, как вы обращаетесь к нему для извлечения данных. В противном случае вы получите неизвестно что.
- Используйте правильные значения индексов. Это имеет первостепенное значение, поскольку пытаться обратиться к массиву с неверным индексом -- это то же самое, что использовать неинициализированный указатель. Это может привести к странному поведению программы и GPF в Windows.

Рассмотрим простой пример обращения к массиву:

```
double nums[5];
for (int i = 0; i < 5; ++i)
{
        cout << “Введите число ” << i << “ :  ”;
        cin >> nums[i];
}
cout << “Вы ввели следующие числа: ” << endl;
for (i = 0; i < 5; ++i)
        cout << “nums[” << i << “] = ” << nums[i] << endl;
```

Делайте разумные проверки индексов при обращении к массивам, не предполагайте, что индексы всегда правильны!

Инициализация одномерных массивов

C++ предоставляет удобный механизм инициализации одномерных массивов. Вам нужно только задать список значений, которыми инициализируется массив, заключенный в фигурные скобки ({}). Список должен быть разделен запятыми и может занимать несколько строк. Если данных в списке меньше, чем размер массива, компилятор допишет в остальные элементы нули. В случае же, если данных больше, чем элементов массива, компилятор выдаст сообщение об ошибке.

В листинге 14 приведен исходный код программы, в котором данные задаются путем инициализации.

```
// C++ программа иллюстрирует использование одномерных массивов
#include <iostream.h>
const int MAX = 10;
int main()
{
        double array[MAX] = { 12.2, 45.4, 67.2, 12.2, 34.6,
                                87.4, 83.6, 12.3, 14.8, 55.5 };
        int num_elem = MAX;
        double sum = 0;
        for(int ix = 0; ix < num_elem; ++ix)
        {
                sum += array[ix];
                cout << “array[” << ix << “]:” << array[ix] << endl;
        }
        cout << endl << “Среднее:”  << sum / num_elem << endl;
        return 0;
}
```

Листинг 14.

Массив array объявляется и инициализируется в строке 6. Список величин, которыми инициализируется массив, заключен в скобки и разделен запятыми. Заканчивается он в строке 7. Следующий оператор объявляет переменную num_elem и инициализирует ее константой MAX.

Если размер массива в вашей программе определяется количеством элементов в списке инициализации, но у вас нет желания их пересчитывать, могу вас обрадовать: C++ может автоматически выделять массив размера, равного количеству элементов в списке инициализации. В этом случае при объявлении массива вам не нужно указывать в скобках размерность массива, Компилятор определит это число сам.

Многомерные массивы

В многомерных массивах каждое дополнительное измерение имеет свой параметр доступа, индекс. Двумерные массивы (или матрицы, если хотите), являются наиболее популярными многомерными массивами. Трехмерные менее популярны, и так далее.

Многомерный массив является множеством одномерных массивов.

Общая форма объявления двумерных и трехмерных массивов:

тип array[размер1][размер2];

тип array [размер1][размер2][размер3];

Как и в одномерных массивах, нижнее значение индекса по каждому измерению равно 0, а в скобках указывается количество элементов по каждому измерению.

Примеры

double matrixA[100][10];

char table[41] [22] [3];

int index[7] [12] ;

Важно знать, как C++ хранит элементы многомерных массивов в памяти.

Большинство компиляторов хранит элементы многомерного массива непрерывным списком, как один большой одномерный массив. При выполнении программы вычисляется местоположение элемента в этом одномерном массиве. Рассмотрим некоторые соглашения по поводу упорядочения измерений. Разберем шестимерный массив, случай редкий, но полезный в качестве примера.

	1	2	3	4	5	6	← номер измерения
M	[20]	[7]	[5]	[3]	[2]	[2]	

Размерности более высокого порядка →

Первый элемент массива M[0][0][0][0][0][0] находится и в оперативной памяти первым. Весь массив M занимает непрерывный блок памяти из 8400 элементов (20x7x5x3x2x2). Следующим в памяти располагается элемент с индексом 1 для размерности номер 6 (т. е. M[0][0][0][0][0][1]). Следующим идет элемент с индексом старшей размерности, увеличенным на 1, и так до

тех пор, пока не будет достигнуто верхнее значение индекса для старшей размерности. После этого увеличивается на 1 индекс размерности 5, а индекс размерности 6 сбрасывается в ноль. Таким образом располагаются в памяти все элементы массива. Вы можете сравнить этот способ хранения данных со спидометром, где цифры на колесиках показывают количество пройденных километров, – правое колесико вращается (изменяется) быстрее всех остальных, левое – самое медленное.

Вот другой пример – расположение в памяти элементов трехмерного массива, M[3][2][2].

M[0][0][0] – начальный адрес памяти
M[0][0][1] – третье измерение заполнено
M[0][1][0]
M[0][1][1] – второе и третье измерение заполнены
M[1][0][0]
M[1][0][1] – третье измерение заполнено
M[1][1][0]
M[1][1][1] – второе и третье измерение заполнены
M[2][0][0]
M[2][0][1] – третье измерение заполнено
M[2][1][0]
M[2][1][1] – все размерности заполнены

C++ позволяет инициализировать как одномерные, так и многомерные массивы. Вам нужно задать величины для инициализации в том же порядке, в каком элементы массива располагаются в памяти. Теперь вы видите, как важно знать механизм хранения элементов многомерного массива в памяти, без этого знания вам было бы непонятно, в каком порядке задавать величины для инициализации массива. Рассмотрим исходный код программы представленный в листинге 15. Программа работает с двумерным массивом (матрицей), рассчитывает среднее значение по каждому столбцу.

```
// Пример работы в C++ с двумерным массивом.
#include <iostream.h>
const int MAX_COL = 3;
const int MAX_ROW = 3;
int main()
{
        double matrix[MAX_ROW][MAX_COL] =
                        {1, 2, 3,                       // строка №1
                         4, 5, 6,                       // строка №2
                         7, 8, 9                        // строка №3
                        };
        double sum, average;
        int rows = MAX_ROW, cols = MAX_COL;
        // вывод элементов матрицы
```

```
        cout << “Матрица :” << endl;
        for(int i = 0; i < rows; i++)
        {
                for(int j = 0; j < cols; j++)
                {
                        cout.width(4);
                        cout.precision(1);
                        cout << matrix[i][j] << “ ”;
                }
                cout << endl;
        }
        cout << endl;
// вычисляется среднее значение по каждому столбцу
        for (int j = 0; j < cols; j++)
        {
                sum =0.0;                       // инициализация sum
                for(int i = 0; i < rows; i++)
                sum += matrix[i][j];
                average = sum / rows;
cout << “Среднее значение столбца” << j << “ = ” << average << endl;
        }
        return 0;
}
```

Листинг 15.

Программа из листинга 15 объявляет матрицу matrix и инициализирует ее. Обратите внимание на то, что программа определяет константы MAX_COL MAX_ROW точно соответствующими размерам массива. Оператор объявления массива задает значения для строк матрицы. Функция присваивает начальные значения переменным rows и columns, равные константам MAX_ROW и MAX_COL соответственно. Переменные инициализируются по двум причинам. Во-первых, поскольку их не вводит пользователь, то надо же их как-то задать. А во-вторых, программа работает с матрицей фиксированного размера.

Вложенные циклы выводят элементы массива. Вторая пара вложенных for-циклов вычисляет средние значения для столбцов матрицы.

Указатели на массивы

C++ и его предшественник C поддерживают специальное применение имен массивов. Компилятор интерпретирует имя массива как адрес его первого элемента. Таким образом, если x – массив, то выражения &x[0] и x эквивалентны. В случае матрицы – назовем ее mat – выражения &mat[0][0] и mat также эквивалентны. Этот аспект C++ и C делает их чем-то вроде языков ассемблера высокого уровня. Можно сказать, что как только вы имеете адрес элемента данных, вы получаете его номер. Знание адреса переменной или

массива в памяти позволяет вам манипулировать их содержимым, используя указатели.

Программная переменная – это метка, маркирующая адрес памяти. Использование переменной в программе означает обращение к ассоциированной с ней ячейке памяти путем спецификации ее имени (или этикетки, если угодно). В этом смысле переменная становится именем, указывающим на ячейку памяти – т. е. указателем.

C++ позволяет использовать указатели для обращения к разным элементам массива. Когда вы вызываете элемент x[i] массива x, компилируемый код выполняет две задачи. Первая – получить базовый адрес массива, т.е. узнать, где находится первый элемент массива. Вторая – использовать i для вычисления смещения базового адреса массива. Смещение равно i, умноженному на размер базового типа массива.

Рассмотрим исходный код представленный в листинге 16. Программа демонстрирует использование указателя на одномерный массив и рассчитывает среднее значение данных, находящихся в массиве

```
#include <iostream.h>
const int MAX = 30;
int main()
{
	double x[MAX];
//объявить указатель и инициализировать базовым адресом массива x
	double *realPtr = x;
	double sum = 0.0, mean;
	int n, count;
// получить количество позиций для данных
	do
	{
		cout << “Введите число элементов от 2 до ” << MAX;
		cin >> n;
		cout << endl;
	} while (n < 2 && n > MAX);
	// предложить ввести данные
	for (int i = 0; i < n; i++)
	{
		cout << “x[” << i << “]: ”;
//использовать форму *(x+i) для записи данных в x[i]
		cin >> *(x + i);
	}
	count = n;
	for (i = 0; i < n; i++)
//использовать форму * (realPtr+i) для обращения к x[i]
		sum += *(realPtr + i);
	mean = sum / count;
```

```
        cout << endl << “Среднее значение =  ” << mean << endl;
        return 0;
}
```

Листинг 16.

Программа в листинге 16 объявляет массив x типа double с MAX элементами. Кроме того, программа объявляет указатель realPtr и инициализирует его, используя имя массива x. Таким образом, указатель realPtr хранит адрес элемента x[0], являющегося первым элементом массива x.

Программа использует указатель *(x+i) во входном операторе. Таким образом, идентификатор x работает как указатель на массив x. Выражение *(x+i) ссылается на элемент с номером i массива x, точно так же, как это делает выражение x[i].

Программа использует указатель realPtr в цикле for. Выражение *(realPtr + i) эквивалентно выражению *(x+i), которое, в свою очередь, эквивалентно x[i]. Таким образом, цикл for использует указатель realPtr со значением смещения i для обращения к элементам массива x.

Метод инкремента/декремента указателя

Программа из листинга 6 хранила в указателе realPtr один и тот же адрес. Применяя арифметику указателей и используя индекс i цикла for, можно написать программу, которая увеличивает смещение при обращении к элементам массива x. C++ предоставляет вам и другой метод, позволяя обращаться последовательно к элементам массива без использования точного значения смещения. Метод прост и основан на применении к указателю операции инкремента или декремента. Инициализируйте указатель базовым адрес массива и затем используйте операцию ++ для доступа к следующему элементу массива. В листинге 17 приведена модифицированная версия предыдущей программы, использующая метод инкремента указателя.

```
#include <iostream.h>
const int MAX = 30;
int main ()
{
        double x[MAX];
        double *realPtr = x;
        double sum = 0.0, mean;
        int n, count;
        do
        {
                cout << “Введите число элементов от 2 до ” << MAX;
                cin >> n;
                cout << endl;
```

```
        } while(n < 2 && n > MAX);
        cin >> n;
        cout << endl;
        for (int i = 0; i < n; i++)
        {
                cout << “x[” << i << “]: ”;
//инкрементировать указатель realPtr после получения ссылки
                cin >> *realPtr++;
        }
//восстановить адрес, используя арифметику указателей
        realPtr -= n;
        count = n;
        for (i = 0; i < n; i++)
//инкрементировать указатель realPtr после получения ссылки
                sum += * (realPtr++);
        mean = sum / count;
        cout << endl << “Среднее значение = ” << mean << endl;
        return 0;
}
```

Листинг 17.

Программа в листинге 17 инициализирует указатель realPtr базовым адресом массива x. Программа использует указатель realPtr в операторе ввода данных с клавиатуры. Этот оператор использует выражения *realPtr++ для записи ваших входных данных в текущий доступный элемент массива x и последующей установки указателя на следующий элемент массива x. После завершения входного цикла указатель realPtr указывает на ячейку за последним элементом массива x. Чтобы восстановить указатель на базовый адрес массива x, программа использует оператор присваивания. Этот оператор, используя арифметику указателей, уменьшает текущий адрес в указателе realPtr на величину n*sizeof(double) и восстанавливает тем самым адрес, соответствующий элементу массива x[0]. Программа использует тот же самый метод инкремента для расчета суммы данных во втором цикле for.

Большинство языков программирования поддерживают статические массивы. Но многие языки, в том реле C++, работают и с динамическими массивами.

C++ позволяет вам определять массивы в качестве параметров функции. C++ позволяет задавать массивы-параметры точно или в общем виде: можно укать размер массива при объявлении параметра или объявить параметр с пустыми скобками.

Массив-параметр фиксированного размера

Общая форма объявления в качестве параметра массива фиксированного размера:

type parameterName[arraySize]
Пример:
int minArray(int arr[100]);
void sort (unsigned dayNum[7]);

Массив-параметр неопределенной длины

Общая форма объявления в качестве параметра массива неопределенного размера (открытого массива) такова:

type parameterName [ ]
Пример:
int minArray(int arr[], int num_elem);
void sort (unsigned dayNum[], int num_elem);

Обратите внимание на то, что в последних примерах появился дополнительный параметр, через который передается количество элементов массива. Так как функции неизвестна размерность массива при объявлении, она должна быть указана через дополнительный параметр.

Многомерные массивы – параметры функции

C++ позволяет вводить в качестве параметров функции многомерные массивы. Как и в случае одномерных массивов, вы можете точно указать размер массива либо задать массив неопределенной длины. В последнем случае вы можете оставить неопределенным размер только по одному измерению, а именно по первому. Если вы хотите определить в качестве параметра массив фиксированной длины, вы должны определить размер по каждому измерению.

Массив-параметр фиксированного размера

Общая форма объявления в качестве параметра массива фиксированного размера:

тип имяПараметра[dimlSize][dim2Size]…
Пример:
int minMatrix(int intMat[100][20], int rows, int cols);
void sort (unsigned mat[23][55], int rows, int cols, int collndex);

Массив-параметр неопределенной длины

Общая форма объявления в качестве параметра массива неопределенной длины (открытого массива) такова:

type parameterName[ ] [dim2Size]…
Пример:
int minMatrix(int intMat[ ][20], int rows, int cols);
void sort (unsigned mat[ ][55], int rows, int cols, int collndex);

Сортировка массивов

Сортировка и поиск – чрезвычайно распространенные операции над массивами. Сортировка массива обычно означает перестановку его

элементов в порядке возрастания. Для определения старшинства элементов может использоваться все или часть значения элемента. Поиск данных в упорядоченном массиве намного проще, чем в неупорядоченном.

Специалисты в вычислительных науках провели много времени и усилий в изучении и создании методов сортировки массивов. Существуют следующие методы сортировки массивов, включая быструю сортировку, сортировку Шелла-Метцнера, динамическую сортировку и сотовую сортировку. Метод быстрой сортировки наиболее скоростной, но имеет некоторые накладные расходы. Шелла-Метцнера и сотовый методы не требуют дополнительных расходов.

5. Понятие записи. Задачи обработки записей

C++ поддерживает структуры (записи), элементы которых могут быть данными предопределенных типов или другими структурами.

Структура позволяет определять новый тип, который логически объединяет несколько полей, или элементов.

Общий синтаксис объявления структуры:

```
struct меткаСтруктуры
{
        < список элементов >
};
```

Примеры

```
struct point
{
        double x;
        double y;
}
struct rect
{
        point upperLeftCorner;
        point lowerRightCorner;
};
struct circle
{
        point center;
        double radius;
};
```

Определив тип struct, вы можете использовать этот тип для объявления переменных. Вот пример объявления, использующий структуру, которая была объявлена выше:

```
point pi, p2, p3;
```

После объявления самой структуры можно сразу объявить структурные переменные:

```
struct point
{
        double x;
        double y;
} pi, p2, p3;
```

Непомеченные (анонимные) структуры позволяют объявлять структурные переменные без определения имени соответствующей структуры.

C++ позволяет объявлять и инициализировать структурные переменные, например:

```
point pt = { 1.0, -8.3 };
```

Для доступа к элементам структуры используют операцию-точку, например:

```
p1.x = 12.45;
p1.y = 34.56;
p2.x = 23.4 / p1.x;
p2.y = 0.98 * pi.y;
```

Этот способ инициализации структуры использовался в старом С. В С++ имеется лучший способ, основанный на использовании конструкторов. Возможность такой инициализации обусловлена тем, что структуры на самом деле являются специальными формами классов. Классы и конструкторы мы с вами будем изучать позже.

В листинге 18 показан исходный код программы использующий структуру. Программа предлагает вам указать две пары координат, определяющих прямоугольник. Прямоугольник определяется координатами x и y верхнего левого и нижнего правого углов. После этого программа рассчитывает и выводит площадь прямоугольника.

```
// Программа на С++, демонстрирующая структурные типы
#include <iostream.h>
#include <math.h>
struct point
{
        double x;
        double y;
};
struct rect
{
        point ulc;              // верхний левый угол
        point lre;              // нижний правый угол
        double area;
};
int main()
{
        rect r;
        double length, width;
        cout << “Введите координаты ВЛУ прямоугольника: ”;
        cin >> r.ulc.x >> r.ulc.y;
        cout << “Введите координаты ВЛУ прямоугольника: ”;
        cin >> r.lrc.x >> r.lrc.y;
        length = fabs(r.ulc.x - r.lrc.x);
        width = fabs(r.ulc.y - r.lrc.y);
        r.area = length * width;
        cout << “Площадь прямоугольника = ” << r.area << endl;
        return 0;
}
```

Листинг 18.

Программа в листинге 18 включает заголовочные файлы IOSTREAM.H MATH.H. В начале программы объявляется структура point, определяющая два элемента типа double – x и y. Эта структура моделирует точку на плоскости. Далее объявляется структура rect, которая моделирует прямоугольник. Структура содержит два элемента типа point, ulc и lrc, и элемент area типа double. Элементы ulc и lrc представляют координаты верхнего левого и нижнего правого углов, определяющих прямоугольник. Элемент area хранит площадь прямоугольника.

Функция main объявляет переменную r типа rect и переменные length и width типа double. Затем получает координаты прямоугольника. Заметьте, как первая операция-точка обеспечивает доступ к элементам rect, а затем дополнительная точка используется с элементами ulc и lre, чтобы обеспечить доступ к элементам point. Length и width вычисляются и их значения используются для определения площади прямоугольника.

Указатели на структуры

C++ поддерживает объявление и использование указателей на структуры. Для присвоения адреса структурной переменной указателю того же типа используется тот же синтаксис, что и в случае простых переменных. После того, как указатель получит адрес структурной переменной, для обращения к элементам структуры нужно использовать операцию ->.

Обращение к элементам структуры

Общий синтаксис для доступа к элементам структуры с помощью указателя имеет вид:

structPtr -> aMember

Пример:

```
struct point
{
        double x;
        double y;
};
point p;
point* ptr = &p;
ptr->x = 23.3;
ptr->y = ptr->x + 12.3;
```

В листинге 19 представлен исходный код программы использующей указатели на структуры.PTR4.CPP. Программа предлагает ввести четыре набора координат для определения четырех прямоугольников. Каждый прямоугольник определяется координатами x и y верхнего левого и нижнего правого углов. Программа вычисляет площадь каждого прямоугольника,

сортирует прямоугольники по площади и отображает прямоугольники в порядке, соответствующем их площадям.

```
//Программа демонстрирующая указатели на структурные типы
#include <iostream.h>
#include <stdio.h>
#include <math.h>
const int MAX_RECT = 4;
struct point
{
        double x;
        double y;
};
struct rect
{
        point ulc;                      // верхний левый угол
        point lre;                      // нижний правый угол
        double area;
        int id;
};
typedef rect rectArr[MAX_RECT];
int main()
{
        rectArr r;
        rect temp;
        rect* pr = r;
        rect* pr2;
        double length, width;
        for(int i = 0; i < MAX RECT; i++, pr++)
        {
cout << “Введите коор. X, Y ВЛУ прямоугольника № ” << i + 1 << “: ”;
                cin >> pr->ulc.x >> pr->ulc.y;
cout << “Введите коор. X, Y НПУ прямоугольника № ” << i + 1 << “: ”;
                cin >> pr->lrc.x >> pr->lrc.y;
                pr->id = i;
                length = fabs(pr->ulc.x - pr->lrc.x);
                width = fabs(pr->ulc. y - pr->lrc.y);
                pr->area = length * width;
        }
        pr -= MAX_RECT;                 // восстановить указатель
//сортировать прямоугольники по площадям
        for(int i = 0; i < (MAX_RECT - 1); i++, pr++)
        {
                pr2 = pr + 1;       // инициализировать указатель pr2
                for(int j = i + 1; j < MAX_RECT;j++, pr2++)
```

```
                    if(pr->area > pr2->area)
                    {
                        temp = *pr;
                        *pr = *pr2;
                        *pr2 = temp;
                    }
        }
        pr -= MAX_RECT - 1;          // восстановить указатель
//отобразить прямоугольники, сортированные по площадям
        for(int i = 0; i < MAX_RECT; i++, pr++)
cout << “Пр-ник № ” << pr->id + 1 << “ – площадь ” << pr->area << endl;
        return 0;
}
```

Листинг 19.

Программа в листинге 19 объявляет указатели pr и pr2. Эти указатели обращаются к структурам типа rect. Программа инициализирует указатель pr базовым адресом массива r.

Первый цикл for использует указатель pr для обращения к элементам массива r. Инкрементная часть цикла содержит выражение pr++, которое использует арифметику указателя для подготовки указателя pr для доступа к следующему элементу массива r. Входные операторы используют указатель pr для доступа к элементам ulc и lre.

Оператор pr -= MAX_RECT восстанавливает адрес, хранящийся в указателе pr, смещая его на MAX_RECT элементов (это MAX_RECT * sizeof(double) байт). Вложенные циклы используют указатели pr и pr2. Выходной цикл for увеличивает на единицу адрес в указателе pr перед следующей итерацией. Оператор pr2 = pr + 1 присваивает pr+1 указателю pr2. Этот оператор предоставляет указателю pr2 доступ к (1+1)-му элементу массива r. Внутренний цикл for увеличивает на единицу указатель pr2 перед следующей итерацией. Таким образом, вложенные циклы for используют указатели pr и pr2 для доступа к элементам массива r. Оператор if использует указатели pr и pr2 для доступа к элементу area при сравнении площадей разных прямоугольников. Следующие обменивают элементы массива r, на которые ссылаются указатели pr и pr2. Обратите внимание, что операторы используют *pr и *pr2 для доступа ко всему элементу массива r. Оператор pr -= MAX_RECT - 1 восстанавливает адрес в указателе pr вычитанием MAX_RECT-1. Последний оператор for использует также указатель pr для получения и вывода значений id и area различных элементов массива r.

Программа иллюстрирует возможность манипуляции массивом с помощью одних указателей. Они допускают большую гибкость в обращении с массивом.

Указатели и динамическая память

Представленные ранее программы создают пространство для своих переменных во время компиляции. Когда программа начинает выполняться, переменным уже отведены их места в памяти. Однако существует много приложений, в которых необходимо создавать новые переменные и динамически распределять для них память во время выполнения программы. Создатели C++ ввели в него новые операции – new и delete, отсутствовавшие ранее в C и предназначенные для того, чтобы динамически распределять и перераспределять память. Несмотря на то, что для программирования в стиле C пока еще применяются такие функции динамической памяти, как malloc, calloc и free, вы должны использовать операции new и delete. По сравнению с указанными функциями эти операции лучше контролируют тип создаваемых динамических данных. К тому же стандартные функции C для создания и удаления динамической памяти не работают с конструкторами и деструкторами, которые мы будем изучать в дальнейшем.

Операции new и delete

Общий синтаксис использования операций new и delete для создания динамической памяти таков:

указатель = new тип;

delete указатель;

Операция new возвращает адрес динамически распределенной переменной. Операция delete освобождает динамически распределенную память, на которую ссылается указатель. Если динамическое распределение с помощью операции new потерпело неудачу, она выбрасывает исключение типа xalloc (объявленное в заголовочном файле EXCEPT.H). Поэтому, если у вас есть основания для беспокойства, вы должны убедиться, что полностью охватили распределение вашей динамической памяти блоками try.

Пример

```
try
{
int *pint;
pint = new int;
*pint = 33;
cout << “Указатель pint сохранен ” << *pint << endl;
delete pint;
}
catch(xalloc&)
{
cout << “Невозможно выделить память” << endl;
}
```

Динамический массив

Для распределения и удаления распределения динамического массива используется следующий общий синтаксис:

указательМассива = new тип[размерМассива];
delete [ ] указательМассива;

Операция new[ ] возвращает адрес динамически распределенного массива. Если распределение потерпело неудачу, операция выбрасывает исключение xalloc. Операция delete[ ] удаляет динамически распределенный массив, на который ссылается указатель.

Пример

```
try {
        const int MAX = 10;
        int *pint;
        pint = new int[MAX];
        for(int i = 0; i < MAX; i++)
                pint[i] = i * i;
        for(i = 0; i < MAX; i++)
                cout << *(pint + i) « endl;
        delete [ ] pint;
        }
catch(xalloc&)
{
cout << “Невозможно выделить память” << endl;
}
```

Все время поддерживайте связь с динамическими переменными и массивами. Не забывайте удалять динамические переменные и массивы после того, как вы перестали ими пользоваться. В противном случае возможна “утечка памяти”, когда кажется, что память в вашей системе становится все меньше без видимых причин. Это значит, что где-то распределяется память, которая затем не освобождается.

Использование указателей для создания и доступа к динамическим данным может быть проиллюстрировано программой в листинге 20. Эта программа вычисляет среднее значение данных в массиве. Сначала она запрашивает, сколько данных вы собираетесь ввести, и проверяет полученное значение. Затем она предлагает вам ввести данные, вычисляет среднее значение данных в массиве и выводит его.

```
// программа на C++, демонстрирующая управление
// динамическими данными с помощью указателей
#include <except.h>
#include <iostream.h>
const int MAX = 30;
int main()
{
        double* x;
        double sum = 0, mean;
```

```
        int *n, count;
        try
        {
                n = new int;
        }
        catch(xalloc&)
        {
                return 1;
        }
        do // получить количество позиций для данных
        {
                cout << “Введите число данных от 2 до ” << MAX << “: ”;
                cin >> *n;
                cout << endl;
        } while(*n < 2 && *n > MAX);
// создать динамический массив в точности нужного размера
        try
        {
                x = new double [*n];
        }
        catch(xalloc&)
        {
                delete n;
                return 1;
        }
// предложить пользователю ввести данные
        for(int i = 0; i < *n; i++)
        {
                cout << “X[” << i << “]: ”;
                cin >> x[i];
        }
// вычислить сумму элементов
        count = *n;                     // число суммируемых элементов
        for(int i = 0; i < *n; i++)
                sum += *(x + i);
        mean = sum / count;             // вычислить среднее значение
        cout << endl << “Среднее = ” << mean << endl << endl;
// удалить распределенную память
        delete n;
        delete[] x;
        return 0;
}
```

Листинг 20.

Программа в листинге 20 использует для динамического распределения два указателя. Строка 8 объявляет первый указатель, используемый для распределения и доступа к динамическому массиву. Строка 10 объявляет второй указатель, создающий динамическую переменную.

В строке 11 использована операция new, распределяющая память для динамической переменной int. Оператор присваивает возвращенный адрес динамических данных указателю n. Оператор catch определяет, потерпело ли неудачу динамическое распределение. Если да, то функция main завершает работу и возвращает код завершения 1 (признак ошибки). На самом деле все это носит несколько надуманный характер, потому что случай распределения всего одной целой переменной чрезвычайно редок.

Цикл do-while ввести количество позиций для данных. Оператор потокового ввода запоминает входные данные в динамической переменной. Оператор использует *n для обращения к переменной. Предложение while также использует *n для доступа к значению динамической переменной. Вообще все операторы программы используют для доступа к количеству позиций данных ссылку указателя *n.

Далее с помощью операции new создает динамический массив типа double с указанным количеством элементов. Эта особенность демонстрирует преимущества использования динамического распределения, позволяющего создать массив в точности нужного размера. Оператор catch определяет, было ли динамическое распределение успешным. Если нет, то операторы освобождают динамическую переменную, на которую ссылается указатель n, и завершают выполнение функции с возвратом значения 1.

Цикл for предлагает ввести значения для динамического массива. Оператор потокового ввода помещает входные данные в i-й элемент динамического массива. Обратите внимание, что оператор использует выражение x[i] для доступа к требуемому элементу. Эта форма напоминает статический массив. C++ обрабатывает выражение x[i] так же, как *(x +1). На самом деле программа использует последнюю форму во втором операторе цикла for, который обращается к элементам динамического массива с помощью *(x + 1).

Последние операторы функции main удаляют динамическую переменную и массив.

Дальние указатели

Для архитектуры таких процессоров, как семейство Intel 80x86, характерно сегментирование памяти. Каждый сегмент имеет объем 64 Кбайта. Использование сегментов имеет свои преимущества и недостатки. Эта структура памяти поддерживает два типа указателей: ближние указатели и дальние указатели.

Внутри сегмента для доступа к данным этого сегмента вы можете использовать ближние указатели. Указатели хранят только адрес смещения в сегменте и поэтому занимают мало места. Напротив, дальние указатели

хранят адреса и сегмента, и смещения, поэтому им требуется больше памяти. 16-битовые приложения Windows используют дальние указатели.

Чтобы объявить дальний указатель, вставьте ключевое слово far (иногда _far) между типом и именем указателя.

Важно отметить, что дальние указатели совершенно непригодны для переноса на другие системы. То есть они употребляются только в приложениях, написанных для IBM PC и совместимых с ними компьютерах, и только при работе в 16-битном режиме, например, в MS DOS или Windows 3.1. Если вы пишете программы для Windows NT, Windows 95 или других 32-битных версий Windows, вы можете полностью игнорировать дальние указатели. На самом деле, ключевые слова far и _far в 32-битном режиме запрещены. Этот 32-битный режим иногда называют плоской моделью.

Необходимо также отметить, что внесение различия между ближними и дальними указателями действительно имеет смысл только при компиляции в чем-то отличном от большой (Large) модели памяти. Лучше всего при написании программ в 16-битном варианте всегда компилировать их в большой модели и никогда не отходить от нее. Экономия памяти, которую вы могли бы получить от использования других моделей памяти, минимальна, если учесть потенциальные ошибки и ужасы смешивания и преобразования ближних и дальних указателей.

Заключение

Вы можете использовать оператор typedef, чтобы создавать типы-псевдонимы существующих типов и определять типы массивов. Оператор typedef имеет следующий общий синтаксис:

typedef известныйТип новыйТип;

Перечислимые типы данных позволяют объявлять уникальные идентификаторы, которые представляют набор логически взаимосвязанных констант. Общий синтаксис объявления перечислимого типа таков:

enum перечислТип {<список перечисляемых идентификаторов>}

Структуры позволяют определять новые типы, которые логически группируют несколько полей или элементов. Общий синтаксис объявления структур имеет вид

```
struct меткаСтруктуры
{
<список элементов>
};
```

Объединения – это форма структур с вариантами. Общий синтаксис объединения таков:

```
union меткаОбъединения
{
        тип1 элемент1;
        тип2 элемент2;
        …
        типn элементn;
};
```

Ссылочные переменные – это псевдонимы переменных, на которые они ссылаются. Чтобы объявить ссылочную переменную, поместите & после типа данных ссылочной переменной или слева от имени переменной.

Указатели – это переменные, которые хранят адреса других переменных или иных данных. C++ использует указатели для обеспечения гибкости и эффективности при манипуляции данными и системными ресурсами.

Указатели существующих переменных используют операцию & для получения адресов этих переменных. Снабженные этими адресами указатели обеспечивают доступ к данным, находящимся в ассоциированных с указателями переменных. Для обращения к переменной используйте операцию *, за которым следует имя указателя.

Массивы – это группы значений, хранящиеся в списке с единственным именем; они полезны для хранения групп данных одинакового типа.

Указатели обращаются к элементам массива, если этим указателям присваивается базовый адрес массива. C++ рассматривает имя массива как эквивалент указателя базового адреса. Например, имя массива a считается эквивалентом как &a[0]. Указатели можно использовать для

последовательного просмотра элементов массива при записи и/или извлечении их значений.

Указатели на структуры манипулируют структурами и обращаются к их элементам. В C++ имеется операция ->, позволяющая указателю обращаться к элементам структуры.

Указатели могут создавать динамические данные и обращаться к ним, используя операции new и delete. Эти операции позволяют создавать динамические переменные и массивы. Операция new присваивает адрес динамических данных указателю, используемому при создании. Операция delete восстанавливает пространство динамических данных, после того как информация становится ненужной.

Дальние указатели – это указатели, которые хранят как адрес сегмента, так и адрес смещения элементов данных.

Ближние указатели хранят только адреса смещения. Дальние указатели требуют больше памяти, чем ближние.

При объявлении одномерных массивов им можно присвоить начальные значения. Список инициализации должен быть заключен в фигурные скобки, а элементы в нем должны быть разделены запятыми. Можно при инициализации задать данных меньше, чем размер массива. В этом случае компилятор автоматически присвоит нулевые значения тем элементам, которые вы не инициализировали. И вдобавок, если вы не укажете размерность инициализируемого массива, она будет определена по количеству элементов в списке инициализации.

Объявление одномерных массивов в качестве параметров функции возможно в двух формах: массив-параметр фиксированной размерности и массив-параметр неопределенной длины (открытый массив). При объявлении параметром массива фиксированной размерности указывается размер массива. В этом случае передаваемые функции аргументы должны соответствовать параметру по типу и размеру. Массив-параметр неопределенной длины объявляется с пустыми скобками, означающими, что аргумент может быть любого размера.

Сортировка массива – важная операция. В результате сортировки элементы массива распределяются в порядке возрастания или убывания. Осуществлять поиск в сортированном массиве намного проще, чем в несортированном. Для сортировки массивов можно использовать эффективную встроенную функцию быстрой сортировки qsort.

Поиск в массиве означает нахождение в массиве элемента, совпадающего с заданным значением. Методы поиска делятся на две группы: для упорядоченных и неупорядоченных массивов. Метод линейного поиска применяется для неупорядоченных массивов, а метод двоичного поиска – для сортированных массивов.

При объявлении многомерных массивов вам нужно указать тип массива, его имя и размер (заключенный в свою пару скобок) по каждому измерению. Нижнее значение индекса для любого измерения равно 0.

Верхнее значение индекса по любому измерению равно количеству элементов по этому измерению минус единица.

Для того чтобы обратиться к многомерному массиву, вам нужно задать его имя и правильные значения индексов. Каждый индекс должен быть заключен в свою пару скобок.

При объявлении многомерных массивов им можно присвоить начальные значения. Список инициализации должен быть заключен в фигурные скобки, а элементы в нем должны быть разделены запятыми. Можно при инициализации задать данных меньше, чем размер массива. В этом случае компилятор автоматически присвоит нулевые значения тем элементам, для которых вы не указали начальные значения

Объявление многомерных массивов в качестве параметров функции возможно в двух формах: массив-параметр фиксированной размерности массив-параметр неопределенной длины по первому измерению. При объявлении параметром массива фиксированной размерности указываете размер массива по каждому измерению. В этом случае передаваемые функции аргументы должны соответствовать по типу и размеру параметру. Массив-параметр неопределенной длины объявляется с пустыми скобками для первого измерения, означающими, что передаваемый аргумент может быть любого размера по первому измерению. По другим измерениям размеры аргумента и параметра должны совпадать.