ЗАНЯТИЕ 6
Ветвление процесса выполнения программ

Большинство приложений должно действовать по-разному в зависимости от ситуации или введенных пользователем данных. Чтобы позволить приложению реагировать по-другому, необходимы условные операторы, выполняющие разные сегменты кода в разных ситуациях.

На сегодняшнем занятии.

- Как заставить программу вести себя по-разному в определенных условиях.
- Как неоднократно выполнить раздел кода в цикле.
- Как лучше контролировать поток выполнения в цикле.

Условное выполнение с использованием конструкции `if...else`

Программы, которые вы видели и создавали до сих пор, имели последовательный порядок выполнения — сверху вниз. Все строки выполнялись, и ни одна не игнорировалась. Однако последовательное выполнение всех строк кода редко используется в приложениях.

Предположим, программа должна умножать два числа, если пользователь нажимает клавишу <м>, или суммировать их, если он нажимает другую клавишу.

Как можно заметить на рис. 6.1, не все участки кода выполняются при каждом запуске. Если пользователь нажимает клавишу <м>, выполняется код, умножающий два числа. Если он вводит нечто другое, выполняется код суммирования. Ни при какой ситуации обе последовательности кода не выполняются.

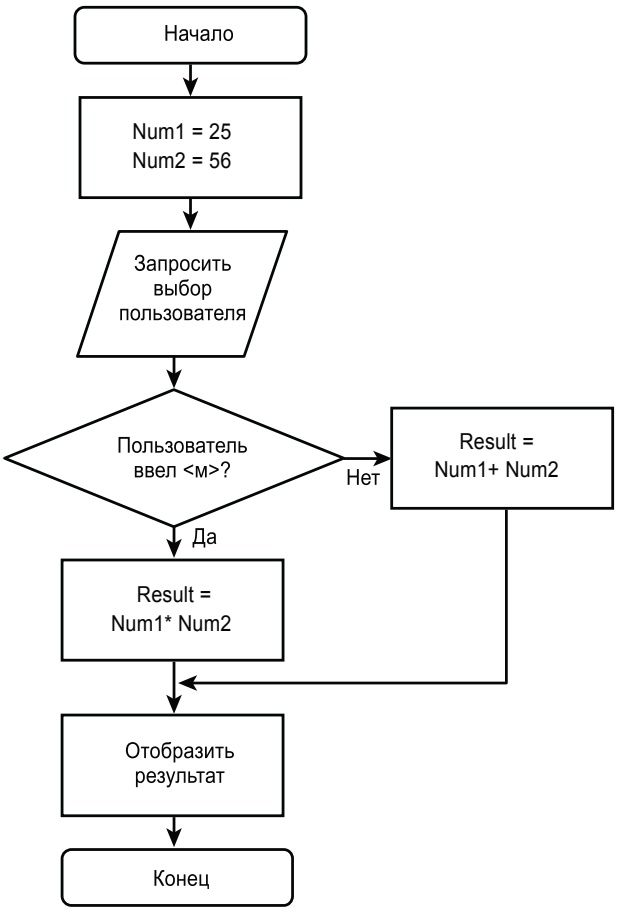

РИС. 6.1. Пример условного выполнения кода на основе пользовательского ввода

Условное программирование с использованием конструкции `if...else`

Условное выполнение кода реализовано в языке C++ на базе конструкции `if ... else`, синтаксис который выглядит следующим образом:

```
if (условное выражение)
    Сделать нечто, когда условное выражение возвращает true;
else // Необязательно
    Сделать нечто другое, когда условное выражение возвращает false;
```

Таким образом, конструкция `if ... else`, позволяющая программе умножать, если пользователь нажимает клавишу <м>, и добавлять в противном случае, выглядит следующим образом:

```
if (UserSelection == 'm')
     Result = Num1 * Num2; // умножение
else
     Result = Num1 + Num2; // сумма
```

ПРИМЕЧАНИЕ

В языке C++ результат `true` выражения по существу означает, что оно возвратило не значение `false`, которое является нулем. Таким образом, любое выражение, возвращающее любое ненулевое число, положительное или отрицательное, по существу рассматривается как возвращающее значение `true`, когда оно используется в условном выражении.

Проанализируем конструкцию в листинге 6.1, обрабатывающую условное выражение и позволяющую пользователю решить, следует ли умножить или просуммировать два числа.

ЛИСТИНГ 6.1. Умножение или сложение двух целых чисел на основе пользовательского ввода

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Enter two integers: " << endl;
 6:     int Num1 = 0, Num2 = 0;
 7:     cin >> Num1;
 8:     cin >> Num2;
 9:
10:     cout << "Enter \'m\' to multiply, anything else to add: ";
11:     char UserSelection = '\0';
12:     cin >> UserSelection;
13:
14:     int Result = 0;
15:     if (UserSelection == 'm')
16:         Result = Num1 * Num2;
17:     else
18:         Result = Num1 + Num2;
19:
```

```
20:     cout << "Result is: " << Result << endl;
21:
22:     return 0;
23: }
```

Результат

```
Enter two integers:
25
56
Enter 'm' to multiply, anything else to add: m
Result is: 1400
```

Следующий запуск:

```
Enter two integers:
25
56
Enter 'm' to multiply, anything else to add: a
Result is: 81
```

Анализ

Обратите внимание на использование оператора `if` в строке 15 и оператора `else` в строке 17. В строке 15 мы инструктируем компилятор выполнить умножение, если следующее за оператором `if` выражение `(UserSelection == 'm')` истинно (возвращает значение `true`), или выполнять сложение, если выражение ложно (возвращает значение `false`). Выражение `(UserSelection == 'm')` возвращает значение `true`, когда пользователь вводит символ `'m'` (с учетом регистра), и значение `false` в любом другом случае. Таким образом, эта простая программа моделирует блок-схему на рис. 6.1 и демонстрирует, как ваше приложение может вести себя по-другому при иной ситуации.

ПРИМЕЧАНИЕ

Часть `else` конструкции `if...else` является необязательной и может не использоваться в тех ситуациях, когда в противном случае не нужно делать ничего.

ВНИМАНИЕ!

Если бы строка 15 в листинге 6.1 выглядела так:

```
15:     if (UserSelection == 'm');
```

то конструкция `if` была бы бессмысленна, поскольку она завершилась бы в той же строке пустым оператором (точка с запятой). Будьте внимательны и избегайте такой ситуации, поскольку вы не получите от компилятора сообщения об ошибке, если за оператором `if` не последует часть `else`.

Некоторые хорошие компиляторы в такой ситуации предупреждают об этом сообщением “`empty control statement`” (пустой управляющий оператор).

Условное выполнение нескольких операторов

Если вы хотите выполнить несколько операторов в зависимости от условий, необходимо заключить их в блок операторов. По существу, это фигурные скобки ({ . . . }), включающие несколько операторов, которые будут выполняться как блок. Рассмотрим пример:

```
if (условие)
{
    // блок при истинном условии
    Оператор 1;
    Оператор 2;
}
else
{
    // блок при ложном условии
    Оператор 3;
    Оператор 4;
}
```

Такие блоки называются также *составными операторам* (compound statement).

На занятии 4, "Массивы и строки", объяснялась опасность использования статических массивов и пересечения их границ. Чаще всего эта проблема проявляется в символьных массивах. При записи строки в символьный массив или при его копировании важно проверять, является ли массив достаточно большим, чтобы содержать все символы. Листинг 6.2 демонстрирует выполнение такой проверки, позволяющей избежать переполнения буфера.

ЛИСТИНГ 6.2. Проверка емкости перед копированием строки в символьный массив

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:     char Buffer[20] = {'\0'};
 7:
 8:     cout << "Enter a line of text: " << endl;
 9:     string LineEntered;
10:     getline (cin, LineEntered);
11:
12:     if (LineEntered.length() < 20)
13:     {
14:         strcpy(Buffer, LineEntered.c_str());
15:         cout << "Buffer contains: " << Buffer << endl;
16:     }
17:
18:     return 0;
19: }
```

Результат

```
Enter a line of text:
This fits buffer!
Buffer contains: This fits buffer!
```

Анализ

Обратите внимание, как в строке 12 проверяются длины строки и буфера перед копированием. Интересным в этой проверке является также присутствие блока операторов в строках 13–16 (называемого также ставным оператором).

ВНИМАНИЕ!

В конце строки `if(`*условие*`)` НЕТ точки с запятой. Это сделано намеренно и гарантирует, что оператор после оператора `if` выполнится в случае истинности условия.

Следующий фрагмент кода вполне корректен:

```
if(условие);
    оператор;
```

Но вы не получите ожидаемого результата, поскольку конструкция `if` завершается последующей точкой с запятой и вне зависимости от условия не делает ничего, зато следующий оператор выполняется всегда.

Вложенные операторы `if`

Нередки ситуации, когда необходимо проверить несколько разных условий, некоторые из которых зависят от результата оценки предыдущего условия. Язык C++ позволяет вкладывать операторы `if` для выполнения таких требований.

Вложенные операторы `if` имеют следующий синтаксис:

```
if (условие1)
{
    СделатьНечто1;
    if(условие2)
        СделатьНечто2;
    else
        СделатьНечтоДругое2;
}
else
    СделатьНечтоДругое1;
```

Считайте приложение, подобное представленному в листинге 6.1, где пользователь, нажимая клавишу <d> или <m>, может указать приложению, следует ли поделить или умножить значения. Далее, деление должно быть разрешено, только если делитель отличается от нуля. Поэтому в дополнение к проверке вводимой пользователем команды следует проверить, не является ли делитель нулем, когда пользователь требует деления. Для этого в листинге 6.3 используется вложенная конструкция `if`.

ЛИСТИНГ 6.3. Использование вложенных операторов `if` в приложении умножения или деления чисел

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Enter two numbers: " << endl;
```

```
 6:     float Num1 = 0, Num2 = 0;
 7:     cin >> Num1;
 8:     cin >> Num2;
 9:
10:     cout << "Enter 'd' to divide, anything else to multiply: ";
11:     char UserSelection = '\0';
12:     cin >> UserSelection;
13:
14:     if (UserSelection == 'd')
15:     {
16:         cout << "You want division!" << endl;
17:         if (Num2 != 0)
18:         {
19:             cout << "No div-by-zero, proceeding to calculate"
                     << endl;
20:             cout << Num1 << " / " << Num2 << " = " << Num1 / Num2
                     << endl;
21:         }
22:         else
23:             cout << "Division by zero is not allowed" << endl;
24:     }
25:     else
26:     {
27:         cout << "You want multiplication!" << endl;
28:         cout << Num1 << " x " << Num2 << " = " << Num1 * Num2
                 << endl;
29:     }
30:
31:     return 0;
32: }
```

Результат

```
Enter two numbers:
45
9
Enter 'd' to divide, anything else to multiply: m
You want multiplication!
45 x 9 = 405
```

Следующий запуск:

```
Enter two numbers:
22
7
Enter 'd' to divide, anything else to multiply: d
You want division!
No div-by-zero, proceeding to calculate
22 / 7 = 3.14286
```

Последний запуск:

```
Enter two numbers:
365
0
```

```
Enter 'd' to divide, anything else to multiply: d
You want division!
Division by zero is not allowed
```

Анализ

Результаты содержат вывод трех запусков программы с тремя разными наборами ввода. Как можно заметить, программа использовала различные пути выполнении кода для каждого из трех запусков. По сравнению с листингом 6.1 эта программа имеет довольно много изменений.

- Числа хранятся в переменных типа `float`, способных хранить десятичные числа, которые понадобятся при делении.
- Условие оператора `if` отличается от использованного в листинге 6.1. Вы больше не проверяете, нажал ли пользователь клавишу <м>; выражение `(UserSelection == 'd')` в строке 14 возвращает значение `true`, если пользователь ввел значение `d`. Если это так, то затребовано деление.
- С учетом того, что эта программа делит два числа и делитель вводится пользователем, важно удостовериться, что делитель не является ненулевым. Это осуществляется в строке 17 вложенным оператором `if`.

Таким образом, эта программа демонстрирует, как вложенные конструкции `if` могут оказаться очень полезными при решении различных задач, связанных с оценкой нескольких параметров.

СОВЕТ

Отступы со смещением, сделанные в коде, необязательны, но они существенно содействуют удобочитаемости вложенных конструкций `if`.

Обратите внимание: конструкции `if...else` можно группировать. Программа в листинге 6.4 запрашивает у пользователя день недели, а затем, используя групповую конструкцию `if...else`, сообщает, в честь чего назван этот день.

ЛИСТИНГ 6.4. Узнайте, в честь чего назван день недели

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     enum DaysOfWeek
 6:     {
 7:         Sunday = 0,
 8:         Monday,
 9:         Tuesday,
10:         Wednesday,
11:         Thursday,
12:         Friday,
13:         Saturday
14:     };
15:
```

```
16:     cout << "Find what days of the week are named after!" << endl;
17:     cout << "Enter a number for a day (Sunday = 0): ";
18:
19:     int Day = Sunday; // Инициализация днем Sunday
20:     cin >> Day;
21:
22:     if (Day == Sunday)
23:         cout << "Sunday was named after the Sun" << endl;
24:     else if (Day == Monday)
25:         cout << "Monday was named after the Moon" << endl;
26:     else if (Day == Tuesday)
27:         cout << "Tuesday was named after Mars" << endl;
28:     else if (Day == Wednesday)
29:         cout << "Wednesday was named after Mercury" << endl;
30:     else if (Day == Thursday)
31:         cout << "Thursday was named after Jupiter" << endl;
32:     else if (Day == Friday)
33:         cout << "Friday was named after Venus" << endl;
34:     else if (Day == Saturday)
35:         cout << "Saturday was named after Saturn" << endl;
36:     else
37:         cout << "Wrong input, execute again" << endl;
38:
39:     return 0;
40: }
```

Результат

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 5
Friday was named after Venus
```

Следующий запуск:

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 9
Wrong input, execute again
```

Анализ

Конструкция `if...else...if`, используемая в строках 22–37, проверяет ввод пользователя и создает соответствующий вывод. Вывод при втором запуске свидетельствует, что программа в состоянии указать пользователю, что он ввел номер вне диапазона 0–6, а следовательно, не соответствующий дню недели. Преимущество этой конструкции в том, что она отлично подходит для проверки множества взаимоисключающих условий, т.е. понедельник не может быть вторником, а недопустимый ввод не может быть никаким днем недели. Другим интересным моментом, на который стоит обратить внимание в этой программе, является использование перечисляемой константы `DaysOfWeek`, объявленной в строке 5 и используемой повсюду в операторе `if`. Можно было бы просто сравнивать пользовательский ввод с целочисленными значениями; так, например, 0 соответствовал бы воскресенью и т.д. Однако использование перечисляемой константы `Sunday` делает код более наглядным.

Условная обработка с использованием конструкции switch-case

Задача конструкции switch-case в том, чтобы сравнить результат некоего выражения с набором возможных констант и выполнить разные действия, соответствующие каждой из этих констант. Новые ключевые слова C++, которые используются в такой конструкции, — это switch, case, default и break.

Конструкция switch-case имеет следующий синтаксис:

```
switch(выражение)
{
case МеткаA:
    СделатьНечто;
    break;
case МеткаB:
    СделатьНечтоДругое;
    break;
// И так далее...
default:
    СделатьНечтоЕслиВыражениеНеСоответствуетНичемуВыше;
    break;
}
```

Код вычисляет результат выражения и сравнивает его на равенство с каждой из меток частей case ниже, каждая из которых должна быть константой, а затем выполняет код после этой метки. Когда результат выражения не соответствует метке *МеткаA*, он сравнивается с меткой *МеткаB*. Если значения совпадают, выполняется часть *СделатьНечтоДругое*. Выполнение продолжается до тех пор, пока не встретится оператор break. Вы впервые встречаете оператор break. Он означает выход из блока кода. Операторы break не обязательны; однако без них выполнение продолжится в коде следующей метки и так далее, до конца блока. Такого явления, как правило, желательно избегать. Часть default также является необязательной, она выполняется тогда, когда результат выражения не соответствует ни одной из меток в конструкции switch-case.

СОВЕТ

Конструкции switch-case хорошо подходят для использования с перечисляемыми константами. Ключевое слово enum было введено на занятии 3, “Использование переменных, объявление констант”.

Код листинга 6.5 является эквивалентом программы дней недели из листинга 6.4, но с использованием конструкции switch-case и перечисляемых констант.

ЛИСТИНГ 6.5. Узнайте, в честь чего назван день недели, с помощью конструкции switch-case

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     enum DaysOfWeek
```

```
 6:     {
 7:         Sunday = 0,
 8:         Monday,
 9:         Tuesday,
10:         Wednesday,
11:         Thursday,
12:         Friday,
13:         Saturday
14:     };
15:
16:     cout << "Find what days of the week are named after!" << endl;
17:     cout << "Enter a number for a day (Sunday = 0): ";
18:
19:     int Day = Sunday; // Инициализация днем Sunday
20:     cin >> Day;
21:
22:     switch(Day)
23:     {
24:     case Sunday:
25:         cout << "Sunday was named after the Sun" << endl;
26:         break;
27:
28:     case Monday:
29:         cout << "Monday was named after the Moon" << endl;
30:         break;
31:
32:     case Tuesday:
33:         cout << "Tuesday was named after Mars" << endl;
34:         break;
35:
36:     case Wednesday:
37:         cout << "Wednesday was named after Mercury" << endl;
38:         break;
39:
40:     case Thursday:
41:         cout << "Thursday was named after Jupiter" << endl;
42:         break;
43:
44:     case Friday:
45:         cout << "Friday was named after Venus" << endl;
46:         break;
47:
48:     case Saturday:
49:         cout << "Saturday was named after Saturn" << endl;
50:         break;
51:
52:     default:
53:         cout << "Wrong input, execute again" << endl;
54:         break;
55:     }
56:
57:     return 0;
58: }
```

Результат

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 5
Friday was named after Venus
```

Следующий запуск:

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 9
Wrong input, execute again
```

Анализ

Строки 22–55 содержат конструкцию `switch-case`, осуществляющую различный вывод в зависимости от того, введено ли пользователем целое число, содержавшееся в переменной `Day`. Когда пользователь вводит число 5, приложение проверяет выражение `Day` оператора `switch`, которое составляет 5, и сравнивает с четырьмя метками, являющимися перечисляемыми константами от `Sunday` (значение 0) до `Thursday` (значение 4), пропуская код ниже каждого из них, поскольку ни один из них не равен 5. По достижении метки `Friday`, значение 5 которой равно выражению `Day` оператора `switch`, выполняется содержащийся ниже ее код, пока не встретится оператор `break`, предписывающий выйти из конструкции `switch`. При следующем запуске, когда вводится недопустимое значение, достигается часть `default` и выполнение кода ниже его отображает сообщение с просьбой повторить ввод.

В этой программе конструкция `switch-case` используется для получения такого же вывода, что и созданного конструкцией `if...else...if` в листинге 6.4. Все же версия с конструкцией `switch-case` выглядит немного более структурированной и, возможно, лучше подходящей к ситуациям, когда необходимо сделать больше, чем просто вывести строку на экран (в этом случае вы заключали бы код ниже меток в фигурные скобки, создавая блоки).

Троичный условный оператор (?:)

Язык C++ предоставляет интересный и мощный оператор, называемый *троичным условным оператором* (conditional operator), который подобен сжатой конструкции `if...else`.

Троичный условный оператор, называемый также *условным оператором* и *троичным оператором*, использует три операнда:

```
(логическое условное выражение) ? выражение1 при true : выражение2 при false;
```

Такой оператор применим при компактном сравнении двух чисел, как показано ниже.

```
int Max = (Num1 > Num2)? Num1 : Num2;
// Max содержит большее число из Num1 и Num2
```

В листинге 6.6 показана условная обработка с использованием оператора `:?`.

ЛИСТИНГ 6.6. Использование условного оператора (`?:`) для поиска большего из двух чисел

```
0: #include <iostream>
1: using namespace std;
```

```
 2:
 3: int main()
 4: {
 5:     cout << "Enter two numbers" << endl;
 6:     int Num1 = 0, Num2 = 0;
 7:     cin >> Num1;
 8:     cin >> Num2;
 9:
10:     int Max = (Num1 > Num2)? Num1 : Num2;
11:     cout << "The greater of " << Num1 << " and " \
12:          << Num2 << " is: " << Max << endl;
13:
14:     return 0;
15: }
```

Результат

```
Enter two numbers
365
-1
The greater of 365 and -1 is: 365
```

Анализ

Интерес представляет код строки 10. Он содержит очень компактное выражение, принимающее решение о том, какое из двух введенных чисел больше. Используя конструкцию `if...else`, эту строку можно было бы переписать следующим образом:

```
int Max = 0;
if (Num1 > Num2)
    Max = Num1;
else
    Max = Num2;
```

Таким образом, троичный условный оператор сэкономил несколько строк кода! Однако экономия строк кода не должна быть приоритетом. Одни программисты предпочитают троичные условные операторы, другие нет. Главное, чтобы пути выполнения кода были легко понятны.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте константы и перечисления для выражений оператора `switch`, чтобы сделать код читабельным	**Не добавляйте** два оператора `case` с одинаковой меткой — это не имеет смысла и не будет компилироваться
Используйте раздел `default` оператора `switch`, только если не уверены в его совершенной ненужности	**Не усложняйте** свои операторы `case`, отказавшись от оператора `break` и разрешив последовательное выполнение, поскольку последующее перемещение оператора `case` может нарушить порядок выполнения кода
Проверяйте, не забыли ли вы включить оператор `break` в конец каждого оператора `case`	**Не используйте** в операторах `:?` сложные условия или выражения

Выполнение кода в циклах

Недавно вы узнали, как заставить программу вести себя по-разному, когда переменные содержат разные значения. Например, код листинга 6.2 осуществлял умножение, когда пользователь нажал клавишу <m>, а в противном случае — суммирование. Но что если пользователь не хочет, чтобы программа закончила выполнение? Что если он хочет выполнить еще одну операцию суммирования или умножения, или возможно еще пять? Вот когда необходимо повторное выполнение уже существующего кода.

Вот когда программе необходим цикл.

Рудиментарный цикл с использованием оператора `goto`

Как и подразумевает название оператора `goto`, он приказывает указателю команд продолжать исполнение с определенной точки в коде. Вы можете использовать его для перехода назад и повторного выполнения определенных операторов. Синтаксис оператора `goto` таков:

```
SomeFunction()
{
    JumpToPoint: // Называется меткой
    CodeThatRepeats;
    goto JumpToPoint;
}
```

Вы объявляете метку `JumpToPoint` и используете оператор `goto` для повторного исполнения кода с этого момента, как показано в листинге 6.7. Если вы не вызываете оператор `goto` с учетом условия, способного вернуть значение `false` при определенных обстоятельствах, или если повторяемый код содержит оператор `return`, выполняемый при определенных условиях, то часть кода между командой `goto` и меткой будет повторяться бесконечно и не позволит программе завершиться.

ЛИСТИНГ 6.7. Запрос пользователю, не хочет ли он повторить вычисления, используя оператор `goto`

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     JumpToPoint:
 6:     int Num1 = 0, Num2 = 0;
 7:
 8:     cout << "Enter two integers: " << endl;
 9:     cin >> Num1;
10:     cin >> Num2;
11:
12:     cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
13:     cout << Num1 << " + " << Num2 << " = " << Num1 + Num2 << endl;
14:
15:     cout << "Do you wish to perform another operation (y/n)?"
             << endl;
16:     char Repeat = 'y';
```

```
17:     cin >> Repeat;
18:
19:     if (Repeat == 'y')
20:         goto JumpToPoint;
21:
22:     cout << "Goodbye!" << endl;
23:
24:     return 0;
25: }
```

Результат

```
Enter two integers:
56
25
56 x 25 = 1400
56 + 25 = 81
Do you wish to perform another operation (y/n)?
y
Enter two integers:
95
-47
95 x -47 = -4465
95 + -47 = 48
Do you wish to perform another operation (y/n)?
n
Goodbye!
```

Анализ

Обратите внимание на основное различие между кодом листинга 6.7 и листинга 6.1: последнему требуются два запуска, чтобы позволить пользователю ввести новый набор чисел и увидеть результат их сложения или умножения. Код листинга 6.7 делает это при одном запуске, циклически повторяя запрос пользователю, желает ли он выполнить другую операцию. Код, фактически обеспечивающий это повторение, находится в строке 20, где вызывается оператор `goto`, если пользователь вводит символ `'y'`. В результате использования оператора `goto` в строке 20 программа переходит к метке `JumpToPoint`, объявленной в строке 5, что фактически перезапускает программу.

ВНИМАНИЕ!

Использование оператора `goto` не рекомендуется для создания циклов, поскольку его массовое применение может привести к непредсказуемой последовательности выполнения кода, когда выполнение может переходить с одной строки на другую без всякого очевидного порядка или последовательности, оставляя также переменные в непредсказуемых состояниях.

Тяжелый случай программы, использующей операторы `goto`, называется *запутанная программа* (spaghetti code). Объясняемые на следующих страницах циклы `while`, `do...while` и `for` вполне позволяют избежать использования оператора `goto`.

Единственная причина упоминания здесь оператора `goto` в том, чтобы объяснить код, который его использует.

Цикл while

Ключевое слово while языка C++ позволяет получить то, что оператор goto делал в листинге 6.7, но правильным способом. Синтаксис таков:

```
while(выражение)
{
    // Результат выражения = true
    БлокОператоров;
}
```

Выполнение блока операторов повторяется до тех пор, пока *выражение* возвращает значение true. Очень важно, чтобы в коде была ситуация, где *выражение* возвращает значение false, иначе цикл while никогда не завершится.

Листинг 6.8 является эквивалентом листинга 6.7, позволяющим пользователю повторять цикл вычисления, но с использованием цикла while вместо оператора goto.

ЛИСТИНГ 6.8. Использование цикла while позволяет пользователю повторно выполнять вычисления

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     char UserSelection = 'm'; // Исходное значение
 6:
 7:     while (UserSelection != 'x')
 8:     {
 9:         cout << "Enter the two integers: " << endl;
10:         int Num1 = 0, Num2 = 0;
11:         cin >> Num1;
12:         cin >> Num2;
13:
14:         cout << Num1 << " x " << Num2 << " = " << Num1 * Num2
                 << endl;
15:         cout << Num1 << " + " << Num2 << " = " << Num1 + Num2
                 << endl;
16:
17:         cout << "Press x to exit(x) or any other key to recalculate"
                 << endl;
18:         cin >> UserSelection;
19:     }
20:
21:     cout << "Goodbye!" << endl;
22:
23:     return 0;
24: }
```

Результат

```
Enter the two integers:
56
```

```
25
56 x 25 = 1400
56 + 25 = 81
Press x to exit(x) or any other key to recalculate
r
Enter the two integers:
365
-5
365 x -5 = -1825
365 + -5 = 360
Press x to exit(x) or any other key to recalculate
x
Goodbye!
```

Анализ

Цикл `while` в строках 7–19 содержит бо́льшую часть логики этой программы. Обратите внимание, что цикл `while` проверяет выражение `(UserSelection! = 'x')` и продолжает выполнение, только если оно возвращает значение `true`. Для обеспечения первого запуска символьная переменная `UserSelection` инициализируется в строке 5 значением `'ì'`. Это должно было быть любым значением, кроме `'x'` (иначе условие не будет выполняться с самого первого цикла и приложение закончит выполнение, не позволив пользователю ничего сделать). Первый запуск очень прост — у пользователя в строке 17 спрашивают, не желает ли он выполнить вычисления снова. Строка 18 принимает сделанный пользователем выбор; здесь вы изменяете значение выражения, которое проверяет цикл `while`, давая программе шанс продолжить выполнение или завершить его. По завершении первого цикла выполнение возвращается к проверке выражения оператора `while` в строке 7 и цикл повторяется, если пользователь ввел значение, отличное от `'x'`. Если пользователь нажал клавишу <x>, то выражение в строке 7 вернет значение `false` и цикл `while` окончится. Приложение после этого также закончит работу, вежливо попрощавшись.

ПРИМЕЧАНИЕ

Цикл также называется *итерацией* (iteration). Выражения, задействовавшие циклы `while`, `do...while` и `f_or`, также называют итерационными выражениями.

Цикл do...while

Бывают ситуации (как в листинге 6.8), когда определенному сегменту кода, повторяемому в цикле, необходимо гарантировать выполнение по крайней мере однажды. Для этого используется цикл `do...while`.

Его синтаксис таков:

```
do
{
    БлокОператоров; // выполняется как минимум раз
} while(условие);   // закончить цикл, если условие ложно
```

Обратите внимание, что строка, содержащая часть `while(`*условие*`)`, заканчивается точкой с запятой. Это является отличием от цикла `while`, в котором точка с запятой фактически завершила бы цикл в первой строке, приведя к пустому оператору.

ЛИСТИНГ 6.9. Использование цикла `do...while` для повторного выполнения блока кода

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     char UserSelection = 'x'; // Исходное значение
 6:     do
 7:     {
 8:         cout << "Enter the two integers: " << endl;
 9:         int Num1 = 0, Num2 = 0;
10:         cin >> Num1;
11:         cin >> Num2;
12:
13:         cout << Num1 << " x " << Num2 << " = " << Num1 * Num2
                 << endl;
14:         cout << Num1 << " + " << Num2 << " = " << Num1 + Num2
                 << endl;
15:
16:         cout << "Press x to exit(x) or any other key to recalculate"
                 << endl;
17:         cin >> UserSelection;
18:     } while (UserSelection != 'x');
19:
20:     cout << "Goodbye!" << endl;
21:
22:     return 0;
23: }
```

Результат

```
Enter the two integers:
654
-25
654 x -25 = -16350
654 + -25 = 629
Press x to exit(x) or any other key to recalculate
m
Enter the two integers:
909
101
909 x 101 = 91809
909 + 101 = 1010
Press x to exit(x) or any other key to recalculate
x
Goodbye!
```

Анализ

Эта программа очень похожа по действию и выводу на предыдущую. Действительно, единственное различие — в ключевом слове `do` в строке 6 и использовании цикла `while` в строке 18. Выполнение кода происходит последовательно, одна строка за другой, пока в строке 18 не встретится оператор `while`, проверяющий выражение `(UserSelection != 'x')`. Когда оно возвращает значение `true` (т.е. пользователь не нажал клавишу <x> для выхода), цикл повторяется. Когда выражение возвращает значение `false` (т.е. пользователь нажал клавишу <x>), выполнение покидает цикл и продолжается до окончания приложения, включая вывод прощания.

Цикл `for`

Цикл `for` немного сложней и обладает выражением инициализации, выполняемым только однажды (обычно для инициализации счетчика), условия выхода (как правило, использующего этот счетчик) и выполняемого в конце каждого цикла действия (обычно инкремента или изменения этого счетчика).

Синтаксис цикла `for` таков:

```
for (выражение инициализации, выполняемое только раз;
     условие выхода, проверяемое в начале каждого цикла;
     выражение цикла, выполняемое в конце каждого цикла)
{
    БлокОператоров;
}
```

Цикл `for` — это средство, позволяющее программисту определить счетчик с исходным значением, проверить его значение в условии выхода в начале каждого цикла и изменить значение счетчика в конце цикла.

В листинге 6.10 показан эффективный способ доступа к элементам массива при помощи цикла `for`.

ЛИСТИНГ 6.10. Использование цикла `for` для ввода и отображения элементов статического массива

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     const int ARRAY_LENGTH = 5;
 6:     int MyInts[ARRAY_LENGTH] = {0};
 7:
 8:     cout << "Populate array of " << ARRAY_LENGTH << " integers"
           << endl;
 9:
10:     for (int ArrayIndex = 0; ArrayIndex < ARRAY_LENGTH; ++ArrayIndex)
11:     {
12:        cout << "Enter an integer for element " << ArrayIndex << ": ";
13:        cin >> MyInts[ArrayIndex];
14:     }
15:
```

```
16:     cout << "Displaying contents of the array: " << endl;
17:
18:     for (int ArrayIndex = 0; ArrayIndex < ARRAY_LENGTH; ++ArrayIndex)
19:         cout << "Element " << ArrayIndex << " = "
                 << MyInts[ArrayIndex] << endl;
20:
21:     return 0;
22: }
```

Результат

```
Populate array of 5 integers
Enter an integer for element 0: 365
Enter an integer for element 1: 31
Enter an integer for element 2: 24
Enter an integer for element 3: -59
Enter an integer for element 4: 65536
Displaying contents of the array:
Element 0 = 365
Element 1 = 31
Element 2 = 24
Element 3 = -59
Element 4 = 65536
```

Анализ

Листинг 6.10 содержит два цикла `for` в строках 10 и 18. Первый помогает ввести элементы в массив целых чисел, а второй — отобразить их. Синтаксис обоих циклов `for` идентичен. Оба объявляют индексную переменную `ArrayIndex` для доступа к элементам массива. Значение этой переменной увеличивается в конце каждого цикла; поэтому на следующей итерации цикла она позволяет обратиться к следующему элементу. Среднее выражение в цикле `for` — это условие выхода. Оно проверяет, находится ли значение переменной `ArrayIndex`, увеличенное в конце каждого цикла, все еще в пределах границ массива (сравнивая его со значением `ARRAY_LENGTH`). Этим гарантируется, что цикл `for` никогда не превысит длину массива.

ПРИМЕЧАНИЕ

Такая переменная, как `ArrayIndex`, из листинга 6.10, которая позволяет обращаться к элементам коллекции (например, массива), называется *итератором* (iterator).

Область видимости итератора, объявленного в пределах конструкции `for`, ограничивается этой конструкцией. Таким образом, во втором цикле `for` листинга 6.10 эта переменная, которая была объявлена повторно, фактически является новой переменной.

Однако применение инициализации, условия выхода и выражения, выполняемого в конце каждого цикла, является необязательным. Вполне возможно получить цикл `for` без некоторых или любого из них, как показано в листинге 6.11.

ЛИСТИНГ 6.11. Использование цикла `for` без выражения цикла для повторения вычислений до просьбы пользователя

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     // без выражения цикла (третье выражение пропущено)
 6:     for(char UserSelection = 'm'; (UserSelection != 'x'); )
 7:     {
 8:         cout << "Enter the two integers: " << endl;
 9:         int Num1 = 0, Num2 = 0;
10:         cin >> Num1;
11:         cin >> Num2;
12:
13:         cout << Num1 << " x " << Num2 << " = " << Num1 * Num2
                 << endl;
14:         cout << Num1 << " + " << Num2 << " = " << Num1 + Num2
                 << endl;
15:
16:         cout << "Press x to exit or any other key to recalculate"
                 << endl;
17:         cin >> UserSelection;
18:     }
19:
20:     cout << "Goodbye!" << endl;
21:
22:     return 0;
23: }
```

Результат

```
Enter the two integers:
56
25
56 x 25 = 1400
56 + 25 = 81
Press x to exit or any other key to recalculate
m
Enter the two integers:
789
-36
789 x -36 = -28404
789 + -36 = 753
Press x to exit or any other key to recalculate
x
Goodbye!
```

Анализ

Это идентично коду листинга 6.8, который использовал цикл `while`; единственное отличие в использовании цикла `for` в строке 8. Самое интересное в этом цикле `for` то, что он содержит только выражение инициализации и условие выхода, без возможности изменять значения переменной в конце каждого цикла.

ПРИМЕЧАНИЕ

В пределах выражения инициализации цикла `for` можно инициализировать несколько переменных. Цикл `for` в листинге 6.11 при инициализации нескольких переменных выглядел бы следующим образом:

```
for (int Index = 0, AnotherInt = 5; Index < ARRAY_LENGTH;
++Index, --AnotherInt)
```

Обратите внимание на новую переменную `AnotherInt`, которая инициализируется значением 5.

В выражении цикла, выполняемом на каждой итерации, вполне можно осуществлять и декремент.

Изменение поведения цикла с использованием операторов `continue` и `break`

В некоторых случаях (особенно в сложных циклах с большим количеством параметров в условии) вы можете не суметь грамотно сформулировать условие выхода из цикла, тогда вам придется изменять поведение программы уже в пределах цикла. В этом могут помочь операторы `continue` и `break`.

Оператор `continue` позволяет возобновить выполнение с вершины цикла. Он просто пропускает код, расположенный в блоке цикла после него. Таким образом, результат выполнения оператора `continue` в цикле `while`, `do...while` или `for` сводится к переходу к условию выхода из цикла и повторному входу в блок цикла, если условие истинно.

ПРИМЕЧАНИЕ

В случае применения оператора `continue` в цикле `for` перед повторной проверкой условия выхода выполняется выражение цикла (третье выражение в операторе `for`, которое обычно увеличивает значение счетчика).

Оператор `break` осуществляет выход из блока цикла, фактически завершая цикл, в котором он был вызван.

ВНИМАНИЕ!

Обычно программисты ожидают, что пока условия цикла выполняются, выполняется и весь код в цикле. Операторы `continue` и `break` изменяют это поведение и могут привести к интуитивно непонятному коду.

Поэтому операторы `continue` и `break` следует использовать только тогда, когда вы на самом деле не можете сообразить, как правильно и эффективно организовать цикл, не используя их.

Следующий далее листинг 6.12 демонстрирует использование оператора `continue` для запроса у пользователя повторного ввода обрабатываемых чисел и оператора `break` для выхода из цикла.

Циклы, которые не заканчиваются никогда, т.е. бесконечные циклы

Помните, что у циклов `while`, `do...while` и `for` есть условия, результат `false` вычисления которых приводит к завершению цикла. Если вы зададите условие, которое всегда возвращает значение `true`, цикл никогда не закончится.

Бесконечный цикл `while` выглядит следующим образом:

```
while(true) // выражение while зафиксировано в true
{
    СделатьНечтоНеоднократно;
}
```

Бесконечный цикл `do...while` выглядит так:

```
do
{
    СделатьНечтоНеоднократно;
} while(true); // выражение do...while никогда не вернет false
```

Бесконечный цикл `for` можно создать следующим образом:

```
for (;;) // нет условия выхода, значит, цикл for бесконечный
{
    СделатьНечтоНеоднократно;
}
```

Как ни странно, но у таких циклов действительно есть цель. Представьте операционную систему, которая должна непрерывно проверять, подключено ли устройство USB к порту USB. Это действие должно выполняться регулярно, пока запущена операционная система. Такие случаи гарантируют популярность бесконечных циклов.

Контроль бесконечных циклов

Для выхода из бесконечного цикла (скажем, перед завершением работы операционной системы в предыдущем примере) вы можете вставить оператор `break` (как правило, в пределах блока `if(`*условие*`)`).

Вот пример использования оператора `break` для контроля бесконечного цикла `while`:

```
while(true) // выражение while зафиксировано в true
{
    СделатьНечтоНеоднократно;
    if(выражение)
        break; // выход из цикла, когда выражение возвращает true
}
```

Использование оператора `break` в бесконечном цикле `do..while`:

```
do
{
    СделатьНечтоНеоднократно;
    if(выражение)
        break; // выход из цикла, когда выражение возвращает true
} while(true); // выражение do...while никогда не вернет false
```

Использование оператора `break` в бесконечном цикле `for`:

```
for (;;) // нет условия выхода, значит, цикл for бесконечный
{
    СделатьНечтоНеоднократно;
    if(выражение)
        break; // выход из цикла, когда выражение возвращает true
}
```

Листинг 6.12 демонстрирует использование операторов `continue` и `break` для контроля критерия выхода из бесконечного цикла.

ЛИСТИНГ 6.12. Использование оператора `continue` для перезапуска и оператора `break` для выхода из бесконечного цикла `for`

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     for(;;) // бесконечный цикл
 6:     {
 7:         cout << "Enter two integers: " << endl;
 8:         int Num1 = 0, Num2 = 0;
 9:         cin >> Num1;
10:         cin >> Num2;
11:
12:         cout << "Do you wish to correct the numbers? (y/n): ";
13:         char ChangeNumbers = '\0';
14:         cin >> ChangeNumbers;
15:
16:         if (ChangeNumbers == 'y')
17:             continue; // перезапуск цикла!
18:
19:         cout << Num1 << " x " << Num2 << " = " << Num1 * Num2
                 << endl;
20:         cout << Num1 << " + " << Num2 << " = " << Num1 + Num2
                 << endl;
21:
22:         cout << "Press x to exit or any other key to recalculate"
                 << endl;
23:         char UserSelection = '\0';
24:         cin >> UserSelection;
25:
26:         if (UserSelection == 'x')
27:             break; // выход из бесконечного цикла
28:     }
29:
30:     cout << "Goodbye!" << endl;
31:
32:     return 0;
33: }
```

Результат

```
Enter two integers:
560
25
Do you wish to correct the numbers? (y/n): y
Enter two integers:
56
25
Do you wish to correct the numbers? (y/n): n
56 x 25 = 1400
56 + 25 = 81
Press x to exit or any other key to recalculate
r
Enter two integers:
95
-1
Do you wish to correct the numbers? (y/n): n
95 x -1 = -95
95 + -1 = 94
Press x to exit or any other key to recalculate
x
Goodbye!
```

Анализ

Цикл `for` в строке 5 отличается от такового в листинге 6.11 тем, что он бесконечный, в цикле отсутствует условие выхода, проверяемое на каждой итерации. Другими словами, без исполнения оператора `break` этот цикл (а следовательно, это приложение) никогда не завершится. Обратите внимание на вывод, который отличается от представленного до сих пор, — он позволяет пользователю поправить введенные числа, прежде чем программа перейдет к вычислению суммы и произведения. Эта логика реализуется в строках 16 и 17 с использованием оператора `continue`, выполняемого при определенном условии. Когда пользователь нажимает клавишу <y> в ответ на запрос, хочет ли он исправить числа, условие в строке 16 возвращает значение `true`, а следовательно, выполняется оператор `continue`. Оператор `continue` возвращает выполнение к началу цикла, и пользователя снова спрашивают, не желает ли он ввести два целых числа. Аналогично в конце цикла, когда в ответ на предложение выйти из программы пользователь вводит символ `'x'`, условие в строке 26 выполняется и выполняется следующий далее оператор `b_reak`, заканчивающий цикл.

ПРИМЕЧАНИЕ

Для создания бесконечного цикла в листинге 6.12 использован пустой оператор `for(;;)`. Вы можете заменить его оператором `while(true)` или `do... while(true);` и получить тот же результат, хотя и будет использован цикл другого типа.

РЕКОМЕНДУЕТСЯ	РЕКОМЕНДУЕТСЯ
Используйте цикл `do...while`, когда код в цикле должен быть выполнен по крайней мере один раз **Используйте** циклы `while`, `do...while` и `for` с хорошо продуманным условием выхода **Используйте** отступы в блоке кода, содержащегося в цикле, чтобы улучшить его удобочитаемость	**Не используйте** оператор `goto` **Не используйте** операторы `continue` и `break` без крайней необходимости **Не используйте** бесконечные циклы с оператором `break`, если этого можно избежать

Программирование вложенных циклов

Как вы уже видели в начале этого занятия, вложенные операторы `if` позволяют вложить один цикл в другой. Предположим, есть два массива целых чисел. Программа поиска произведения каждого элемента массива `Array1` и каждого элемента массива `Array2` будет проще, если использовать вложенный цикл. Первый цикл перебирает элементы массива `Array1`, а второй цикл, внутри первого, перебирает элементы массива `Array2`.

Листинг 6.13 демонстрирует, как можно вложить один цикл в другой.

ЛИСТИНГ 6.13. Использование вложенных циклов для умножения каждого элемента одного массива на каждый элемент другого

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     const int ARRAY1_LEN = 3;
 6:     const int ARRAY2_LEN = 2;
 7:
 8:     int MyInts1[ARRAY1_LEN] = {35, -3, 0};
 9:     int MyInts2[ARRAY2_LEN] = {20, -1};
10:
11:     cout << "Multiplying each int in MyInts1 by each in MyInts2:"
          << endl;
12:
13:     for(int Array1Index = 0; Array1Index < ARRAY1_LEN; ++Array1Index)
14:         for(int Array2Index = 0;
                Array2Index < ARRAY2_LEN; ++Array2Index)
15:             cout << MyInts1[Array1Index] << " x "
                  << MyInts2[Array2Index] \
16:                 << " = " << MyInts1[Array1Index] * MyInts2[Array2Index]
                    << endl;
17:
18:     return 0;
19: }
```

Результат

```
Multiplying each int in MyInts1 by each in MyInts2:
35 x 20 = 700
35 x -1 = -35
-3 x 20 = -60
-3 x -1 = 3
0 x 20 = 0
0 x -1 = 0
```

Анализ

Рассматриваемые вложенные циклы `for` находятся в строках 13 и 14. Первый цикл `for` перебирает массив `MyInts1`, а второй — массив `MyInts2`. Первый цикл `for` запускает второй в пределах каждой итерации. Второй цикл `for` перебирает все элементы массива `MyInts2`, причем при каждой итерации он умножает этот элемент на элемент, проиндексированный переменной `Array1Index` из первого, внешнего, цикла. Так, для каждого элемента массива `MyInts1` второй цикл переберет все элементы массива `MyInts2`, в результате первый элемент массива `MyInts1` (со смещением 0) перемножается со всеми элементами массива `MyInts2`. Затем второй элемент массива `MyInts1` перемножается со всеми элементами массива `MyInts2`. И наконец, третий элемент массива `MyInts1` перемножается со всеми элементами массива `MyInts2`.

ПРИМЕЧАНИЕ

Для удобства (и чтобы не отвлекаться от циклов) содержимое массивов в листинге 6.13 инициализируется. В предыдущих примерах, например в листинге 6.10, показано, как позволить пользователю ввести числа в целочисленный массив.

Использование вложенных циклов для перебора многомерного массива

На занятии 4 вы узнали о многомерных массивах. В листинге 4.3 происходит обращение к элементам двумерного массива из трех рядов и трех столбцов. Там обращение к каждому элементу в каждом ряду осуществлялось индивидуально, и не было никакой автоматизации. Если бы массив стал большим или его размерностей стало больше, то для доступа к его элементам понадобилось бы много больше кода. Однако использование циклов может все это изменить, как показано в листинге 6.14.

ЛИСТИНГ 6.14. Использование вложенных циклов для перебора элементов двумерного массива

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     const int MAX_ROWS = 3;
6:     const int MAX_COLS = 4;
7:
```

```
 8:     // Двумерный массив целых чисел
 9:     int MyInts[MAX_ROWS][MAX_COLS] = { {34, -1, 879, 22},
10:                                        {24, 365, -101, -1},
11:                                        {-20, 40, 90, 97} };
12:
13:     // перебор всех рядов массива
14:     for (int Row = 0; Row < MAX_ROWS; ++Row)
15:     {
16:         // перебор всех чисел в каждом ряду (столбцов)
17:         for (int Column = 0; Column < MAX_COLS; ++Column)
18:         {
19:             cout << "Integer[" << Row << "][" << Column \
20:                  << "] = " << MyInts[Row][Column] << endl;
21:         }
22:     }
23:
24:     return 0;
25: }
```

Результат

```
Integer[0][0] = 34
Integer[0][1] = -1
Integer[0][2] = 879
Integer[0][3] = 22
Integer[1][0] = 24
Integer[1][1] = 365
Integer[1][2] = -101
Integer[1][3] = -1
Integer[2][0] = -20
Integer[2][1] = 40
Integer[2][2] = 90
Integer[2][3] = 97
```

Анализ

Строки 14–22 содержат два цикла `for`, необходимых для перебора двумерного массива целых чисел и получения доступа к его элементам. В действительности двумерный массив — это массив массивов целых чисел. Обратите внимание, что первый цикл `for` обращается к рядам (каждый из которых является массивом целых чисел), а второй — к его столбцам, т.е. осуществляет доступ к каждому элементу в этом массиве.

ПРИМЕЧАНИЕ

Скобки в листинге 6.14 вокруг вложенного цикла `for` использованы только для удобочитаемости. Эти вложенные циклы прекрасно сработают и без фигурных скобок, поскольку оператор цикла — это только один оператор, а не составной оператор, который требует использования фигурных скобок.

Использование вложенных циклов для вычисления чисел Фибоначчи

Знаменитая прогрессия Фибоначчи — это ряд чисел, начинающихся с 0 и 1, где каждое последующее число — сумма предыдущих двух. Таким образом, прогрессия Фибоначчи начинается с такой последовательности:

0, 1, 1, 2, 3, 5, 8, … и так далее

В листинге 6.15 показано, как получить прогрессию Фибоначчи из любого желаемого количества чисел, ограниченного только размером целочисленной переменной, хранящей последнее число.

ЛИСТИНГ 6.15. Использование вложенных циклов для вычисления чисел прогрессии Фибоначчи

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     const int NumsToCal = 5;
 6:     cout << "This program will calculate " << NumsToCal \
 7:          << " Fibonacci Numbers at a time" << endl;
 8:
 9:     int Num1 = 0, Num2 = 1;
10:     char WantMore = '\0';
11:     cout << Num1 << " " << Num2 << " ";
12:
13:     do
14:     {
15:         for (int Index = 0; Index < NumsToCal; ++Index)
16:         {
17:             cout << Num1 + Num2 << " ";
18:
19:             int Num2Temp = Num2;
20:             Num2 = Num1 + Num2;
21:             Num1 = Num2Temp;
22:         }
23:
24:         cout << endl << "Do you want more numbers (y/n)? ";
25:         cin >> WantMore;
26:     }while (WantMore == 'y');
27:
28:     cout << "Goodbye!" << endl;
29:
30:     return 0;
31: }
```

Результат

```
This program will calculate 5 Fibonacci Numbers at a time
0 1 1 2 3 5 8
Do you want more numbers (y/n)? y
```

```
13 21 34 55 89
Do you want more numbers (y/n)? y
144 233 377 610 987
Do you want more numbers (y/n)? y
1597 2584 4181 6765 10946
Do you want more numbers (y/n)? n
Goodbye!
```

Анализ

Внешний цикл `do...while` в строке 13 является основным, он запрашивает у пользователя, хочет ли он получить следующие числа. Внутренний цикл `for` в строке 15 решает задачу вычисления и отображения за один раз пяти следующих чисел Фибоначчи. В строке 19 значение переменной `Num2` присваивается временной переменной, чтобы использовать его затем в строке 21. Обратите внимание, что без сохранения этого временного значения, переменной `Num1` было бы присвоено значение, измененное в строке 20, что было бы неправильно. Благодаря этим трем строкам цикл повторяется с новыми значениями в переменных `Num1` и `Num2`, если пользователь нажмет клавишу <y>.

Резюме

На этом занятии вы узнали, что можно писать код, выполняющийся не только сверху вниз; операторы условного выполнения кода позволяют создавать альтернативные пути выполнения и повторять блоки кода в цикле. Теперь вы знаете, как использовать конструкцию `if...else` и оператор `switch-case`, чтобы справиться с различными ситуациями, когда переменные содержат различные значения.

Для объяснения концепции циклов был представлен оператор `goto`, однако сразу было сделано предупреждение не использовать его в связи с возможностью создания запутанного кода. Вы изучили циклы языка C++, использующие конструкции `while`, `do...while` и `for`, и узнали, как заставить циклы выполнять итерации бесконечно, чтобы создать бесконечные циклы, и использовать операторы `continue` и `break` для их контроля.

Вопросы и ответы

- **Что будет, если я пропущу оператор `break` в конструкции `switch-case`?**

  Оператор `break` позволяет выйти из конструкции `switch`. Без него продолжится выполнение следующих операторов в частях `case`.

- **Как выйти из бесконечного цикла?**

  Для выхода из цикла используется оператор `break`. Оператор `return` позволяет выйти также из блока функции.

- **Мой цикл `while` выглядит как `while(Integer)`. Цикл будет продолжаться, пока значением переменной `Integer` не станет -1, не так ли?**

  В идеале выражение выхода из цикла `while` должно возвращать логическое значение `true` или `false`, однако как `false` интерпретируется также значение 0. Любое другое значение рассматривается как `true`. Поскольку -1 — это не нуль, условие выхода из цикла `while` возвращает значение `true`, и цикл продолжает выполняться. Если вы хотите, чтобы цикл был выполнен только для положительных чисел, перепишите

выражение как `while(Integer>0)`. Это правило истинно для всех условных операторов и циклов.

- **Эквивалентны ли пустой цикл `while` и оператор `for(;;)`?**
  Нет, оператор `while` всегда нуждается в последующем условии выхода.
- **Я изменил код `do...while(exp);` на `while(exp);` копированием и вставкой. Не возникнет ли каких-нибудь проблем?**
  Да, и большие! Код `while(exp);` — вполне допустимый, хоть и пустой цикл `while`, поскольку перед точкой с запятой нет никаких операторов, даже если за ним следует блок операторов. Блок кода перед первым в вопросе циклом выполняется как минимум однажды. Будьте внимательны при копировании и вставке кода.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Зачем беспокоиться об отступах кода в блоках операторов, вложенных циклов операторов `if`, если код вполне нормально компилируется и без них?
2. Вы можете быстро реализовать переход, используя оператор `goto`. Почему следует избегать его применения?
3. Возможно ли написать цикл `for`, где значение счетчика уменьшается? Как бы он выглядел?
4. В чем проблема со следующим циклом?
```
for (int Counter=0; Counter==10; ++Counter)
    cout << Counter << " ";
```

Упражнения

1. Напишите цикл `for` для доступа к элементам массива в обратном порядке.
2. Напишите вложенный цикл, эквивалентный использованному в листинге 6.13, но добавляющий элементы в два массива в обратном порядке.
3. Напишите программу, которая, подобно листингу 6.15, отображает числа Фибоначчи, но спрашивает пользователя, сколько чисел он хочет вычислить.
4. Напишите конструкцию `switch-case`, которая сообщает, есть ли в радуге такой цвет или нет. Используйте перечисляемую константу.
5. **Отладка:** Что не так с этим кодом?
```
for (int Counter=0; Counter=10; ++Counter)
    cout << Counter << " ";
```

6. **Отладка:** Что не так с этим кодом?

```
int LoopCounter = 0;
while(LoopCounter <5);
{
    cout << LoopCounter << " ";
    LoopCounter++;
}
```

7. **Отладка:** Что не так с этим кодом?

```
cout << "Enter a number between 0 and 4" << endl;
int Input = 0;
cin >> Input;
switch (Input)
{
case 0:
case 1:
case 2:
case 3:
case 4:
    cout << "Valid input" << endl;
default:
    cout << "Invalid input" << endl;
}
```