4. ВЕТВЛЕНИЕ ПРОЦЕССА ВЫПОЛНЕНИЯ ПРОГРАММ

Оглавление

При запуске программы, центральный процессор (сокр. «ЦП») начинает выполнение кода с первой строки функции `main()`, выполняя определенное количество стэйтментов, а затем завершает выполнение при завершении блока `main()`. Последовательность стэйтментов, которые выполняет ЦП, называется **порядком выполнения программы** (или «потоком выполнения программы»).

Большинство программ, которые мы рассматривали до этого момента, были линейными с последовательным выполнением, то есть порядок выполнения у них один и тот же каждый раз: выполняются одни и те же стэйтменты, даже если значения, которые вводит пользователь, — меняются.

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.

§4.1 Операторы условного ветвления if/else

Условное выполнение кода реализовано на базе конструкции `if...else`, синтаксис которой выглядит следующим образом:

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

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

Проанализируем конструкцию в листинге 4.1, обрабатывающую условное выражение и позволяющую пользователю решить, следует ли умножить или просуммировать два числа.

Листинг 4.1.

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

Результат выполнения программы:

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

Обратите внимание на использование оператора `if` в строке 13 и оператора `else` в строке 15. В строке 13 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. 4.1 и демонстрирует, как ваше приложение может вести себя по-другому при иной ситуации.

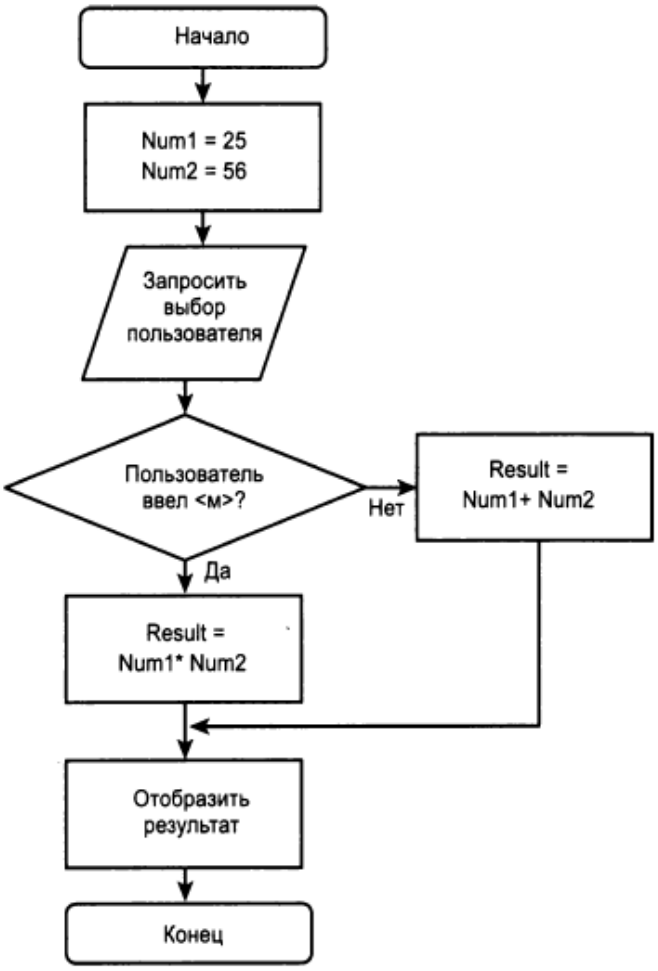

Рисунок 4.1 Пример условного выполнения кода на основе пользовательского ввода

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

Вложенные операторы `if` имеют следующий синтаксис:

```
if (условие 1)
{
    СделатьНечто 1;
    if (условие2)
        {СделатьНечто2;}
```

```
    else
        {СделатьНечтоДругое2;}
}
else
{
    СделатьНечтоДругое1;
}
```

Модифицируем программу из листинга 4.1, добавив операцию деления, которая выполняется при нажатии клавиши <d>, и проверку условия равенства делителя нулю. Для этого в листинге 4.2 используем вложенные условные конструкции.

Листинг 4.2.

```
#include <iostream>
using namespace std;
int main()
{
cout << "Enter two numbers: " << endl;
float Num1 = 0, Num2 = 0;
cin >> Num1;
cin >> Num2;
cout << "Enter ' d' to divide, anything else to multiply: ";
char UserSelection = '\0';
cin >> UserSelection;
if (UserSelection == 'd')
{
cout << "You want division!" << endl;
if (Num2 != 0)
{
cout << "No div-by-zero, proceeding to calculate" << endl;
cout << Num1 << " / " << Num2 << " = " << Num1 / Num2 << endl;
}
else
cout << "Division by zero is not allowed" << endl;
}
else
{
cout << "You want multiplication!" << endl;
cout << Num1 << " x " << Num2 << " = " << Num1 * Num2 << endl;
}
```

```
28  return 0;
29  }
```

Результат выполнения программы:

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

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 4.1 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 4.1. Вы больше не проверяете, нажал ли пользователь клавишу <m>; выражение `(UserSelection == 'd')` в строке 12 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 15 вложенным оператором `if`.

Конструкции `if...else` можно группировать. Пример группировки представлен в листинге 4.3.

Листинг 4.3.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      cout << "Enter an integer: ";
6      int a;
7      cin >> a;
8      cout << "Enter another integer: ";
9      int b;
10     cin >> b;
11     if (a > 0 && b > 0)
12         cout << "Both numbers are positive\n";
13     else if (a > 0 || b > 0)
14         cout << "One of the numbers is positive\n";
15     else
16         cout << "Neither number is positive\n";
17     return 0;
18 }
```

Совокупность этих операторов — `else if` означает, что если не выполнилось предыдущее условие, то проверить данное. Если ни одно из условий не верно, то выполняется тело оператора `else`.

Если после оператора `if`, `else` или их связки `else if` должна выполняться только одна команда, то фигурные скобки можно не ставить.

§4.2 Оператор множественного выбора

Итак, мы рассмотрели оператор с одиночным выбором `if` и оператор с двойным выбором `if else`, но еще имеется оператор множественного выбора `switch`.

Задача конструкции `switch-case` в том, чтобы сравнить результат некоего выражения с набором возможных констант и выполнить разные действия, соответствующие каждой из этих констант.

Конструкция `switch-case` имеет следующий синтаксис:

```
switch (/*переменная или выражение*/)
{
    case /*константное выражение 1*/:
    {
     /*группа операторов 1*/;
     break;
    }
    case /*константное выражение 2*/:
    {
         /*группа операторов 2*/;
         break;
    }
//...
    default:
    {
         /*группа операторов 3*/;
     }
}
```

Сначала пишем ключевое слово `switch` за которым следует выражение, с которым мы хотим работать. Обычно это выражение представляет собой только одну переменную, но это могут быть и сложные выражения. Единственное ограничение к этому выражению — оно должно быть интегрального типа данных (т.е. типа char, short, int, long, long long или enum). Переменные типа с плавающей точкой или неинтегральные типы использоваться не могут.

После выражения `switch` мы объявляем блок. Внутри блока мы используем **лейблы** для определения всех значений, которые мы хотим проверять на соответствие выражению. Существуют два типа лейблов: `case` и `default`.

Константное выражение, находящееся после ключевого слова `case`, проверяется на равенство с выражением, находящимся возле ключевого слова `switch`. Если они совпадают, то тогда выполняется код под соответствующим кейсом. Когда результат

выражения не соответствует метке /*константное выражение1*/, он сравнивается с меткой /*константное выражение2*/. Если значения совпадают, выполняется часть /*группа операторов 2*/.

Выполнение продолжается до тех пор, пока не встретится 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 (лейбл по умолчанию) также является необязательной, она выполняется тогда, когда результат выражения не соответствует ни одной из меток в конструкции switch-case.

Рассмотрим пример выполнения оператора множественного выбора (листинг 4.4).

Листинг 4.4.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int x;
6  cin >> x;
7  switch(x)
8  {
9  case 1:
10    cout << 1 << '\n';
11    break;
12 case 2:
13    cout << 2 << '\n';
14    break;
15 case 3:
16    cout << 3 << '\n';
17    break;
18 case 4:
19    cout << 4 << '\n';
20    break;
21 default:
22    cout << 5 << '\n';
23    break;
24 }
25 }
```

Предположим, пользователь в процессе выполнения программы ввел значение «3». Тогда выполнение программы переходит к лейблу, расположенному в строке 14 и выполняются стэйтменты, соответствующие данному лейблу (строка 15). Оператор `break` прекращает выполнение блока `switch`.

Одна из самых каверзных вещей в `switch` — это последовательность выполнения кода. Когда кейс совпал (или выполняется `default`), то выполнение начинается с первого стэйтмента, который находится после соответствующего кейса и продолжается до тех пор, пока не будет выполнено одно из следующих условий завершения:

1. Достигнут конец блока switch;
2. Выполняется оператор return;
3. Выполняется оператор goto;
4. Выполняется оператор break.

Обратите внимание, если ни одного из этих условий завершения не будет, то выполняться будут все кейсы после того кейса, который совпал с выражением `switch`. Например:

Листинг 4.5.

```
1  switch (2)
2  {
3     case 1: // Не совпадает!
4         cout << 1 << '\n'; // пропускается
5     case 2: // Совпало!
6         cout << 2 << '\n'; // выполнение кода начинается здесь
7     case 3:
8         cout << 3 << '\n'; // это также выполнится
9     case 4:
10        cout << 4 << '\n'; // и это
11    default:
12        cout << 5 << '\n'; // и это
13 }
```

Результат:

2

3

4

5

Когда выполнение переходит из одного кейса в следующий, то это называется **fall-through**.

§4.3 Условный тернарный оператор

Условный (тернарный) оператор (обозначается как ?:) является единственным тернарным оператором, который работает с 3-мя операндами. Из-за этого его часто называют просто «тернарный оператор». Он предоставляет сокращенный способ (альтернативу) ветвления `if/else`.

Синтаксис оператора имеет следующий вид:

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

В стилистике `if/else` данный оператор можно записать в следующем виде:

```
if(логическое условное выражение)
{выражение1 при true;}
else
{выражение2 при false;}
```

Условный тернарный оператор — это удобное упрощение ветвления `if/else`, особенно при присваивании результата переменной или возврате определенного значения, так как данный оператор вычисляется как выражение.

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
```

При использовании тернарного оператора часто возникает необходимость использовать в качестве `выражения1` и `выражения2` сразу блоки выражений:

```
if(z > 5) { cout << x; x += 3; }
else { cout << y; y += 4; }
```

Пытаемся сократить вышеописанную условную конструкцию до:

```
z > 5 ? cout << x, x += 3 : cout << y, y += 4;
```

Но если для `выражения1` использование списка выражений корректно, то для `выражения2` мы сталкиваемся с проблемой приоритетов операций: тернарный оператор

имеет больший приоритет, чем оператор запятая и вышеуказанное выражение интерпретируется следующим образом:

```
(z > 5 ? cout << x, x += 3 : cout << y), y += 4;
```

Данную проблему всегда можно решить расстановкой дополнительных скобок:

```
z > 5 ? cout << x, x += 3 : (cout << y, y += 4);
```

§4.4 Оператор goto

Оператор **goto** — это оператор управления потоком выполнения программ, который заставляет центральный процессор выполнить переход из одного участка кода в другой (осуществить прыжок). Другой участок кода идентифицируется с помощью лейбла. Например:

Листинг 4.6.

```
1  #include <iostream>
2  #include <cmath> // для функции sqrt()
3  using namespace std;
4  int main()
5  {
6      double z;
7      tryAgain: // это лейбл
8      cout << "Enter a non-negative number: ";
9      cin >> z;
10     if (z < 0.0)
11         goto tryAgain; // а это оператор goto
12     cout << "The sqrt of " << z << " is " << sqrt(z) << endl;
13     return 0;
14 }
```

В этой программе пользователю предлагается ввести неотрицательное число. Однако, если пользователь введет отрицательное число, программа, используя оператор `goto`, выполнит переход обратно к лейблу `tryAgain`. Затем пользователю снова нужно будет ввести число. Таким образом, мы можем постоянно запрашивать у пользователя ввод числа, пока он не введет корректное число.

Оператор `goto` и соответствующий лейбл должны находиться в одной и той же функции, так как они имеют область видимости функции.

Существуют некоторые ограничения на использование операторов `goto`. Например, вы не сможете перепрыгнуть вперед через переменную, которая инициализирована в том же блоке, что и `goto`:

Листинг 4.7.

```
1  int main()
2  {
3  goto skip; // прыжок вперед недопустим
4  int z = 7;
5  skip: // лейбл
6  z += 4; // какое значение будет в этой переменной?
7  return 0;
8  }
```

В целом, программисты избегают использования оператора `goto`. Основная проблема с ним заключается в том, что он позволяет программисту управлять выполнением кода так, что точка выполнения может прыгать по коду произвольно. А это, в свою очередь, создает то, что опытные программисты называют «спагетти-кодом».

Спагетти-код — это код, порядок выполнения которого напоминает тарелку со спагетти (всё запутано и закручено), что крайне затрудняет следование порядку и понимание логики выполнения такого кода.