НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Дисциплина «Информатика (основы программирования)»

Методические указания

Тематическое занятие 2

Циклы, работа с циклическими конструкциями.

Содержание

Инкремент, декремент и операции с присваиванием

Операции инкремента и декремента

Одноместная (унарная) операция *инкрементирования* добавляет к своему операнду единицу. Она обозначается символами `++` и может использоваться как в префиксной (`++a`), так и в постфиксной (`a++`) формах (здесь переменная `a` имеет целочисленный тип). Различие заключается в порядке выполнения операции: в выражении `++a` значение переменной увеличивается на `1` *до того*, как она используется, а в выражении `a++` – *после того*.

Например, пусть `a=5`, тогда следующие операторы дают разный результат

```
b = ++a; /* эквивалентно a=a+1; b=a; в результате a=6, b=6 */
b = a++; /* эквивалентно b=a; a=a+1; в результате a=6, b=5 */
```

Аналогично, одноместная операция *декрементирования* обозначается символами `--` и вычитает из своего операнда единицу. Например, `--a` или `a--`.

Эти две операции применимы только к переменным, выражения наподобие `(a+b)++` недопустимы.

В выражениях приоритет операций инкремента и декремента такой же высокий, как у унарных `+` и `-`. Порядок ассоциирования также справа налево (←).

Описанные свойства префиксной и постфиксной форм записи данных операций удобно использовать, например, при работе с индексами элементов массива. (Примеры будут приведены в соответствующих разделах.)

Но главное преимущество заключается в том, что использование операций инкремента и декремента позволяет уменьшить количество выполняемых программой операций присваивания (см. комментарий к примеру). А это приводит к увеличению скорости выполнения программы. (*Количество операций присваивания* – один из ключевых показателей эффективности работы компьютерных программ.)

Побочные эффекты

Для операций *инкремента* и *декремента* необходимо соблюдать правило: ***не следует применять эти операции к переменной, которая входит в выражение более одного раза***. Например, не следует использовать такое выражение:

```
b = a + 2 * a++;
```

Весьма показателен следующий пример. Для выражения:

```
b = (3 * a++) + (2 * a++);
```

не понятно, в какой момент значение переменной `a` будет увеличено на `1` (когда это произойдет в первый раз, и когда – во второй). Данный вопрос не оговорен в стандарте языка С, и целиком зависит от разработчиков используемой версии компилятора. Результат выполнения такой операции не определен, он

называется ***«побочным эффектом»***. Подобных выражений необходимо тщательно избегать.

Операции с присваиванием

Выражения с операцией присваивания, в которых переменная из левой части повторяется в правой, можно записать в более компактной форме с помощью *операций с присваиванием*. Например:

`a = a + 3` можно записать так `a += 3`

Аналогично, для других операций с присваиванием запись вида

`ИмяПерем Опер= Выраж`

эквивалентна следующей записи

`ИмяПерем = (ИмяПерем) Опер (Выраж)`

здесь `Опер` – знак одной из операций: `+`, `-`, `*`, `/`, `%`. В этой записи следует обращать внимание на скобки – например, оператор

`a *= b + 2;` эквивалентен `a = a * (b + 2);`

Операции с присваиванием имеют такой же низкий приоритет, как и операция присваивания (=). Порядок ассоциирования тоже слева направо (→).

Помимо краткости записи, выражения с такими операциями более естественны для человеческого восприятия. Мы говорим «увеличить `a` на `3`», а не «извлечь `a`, прибавить `3` и поместить результат обратно в `a`».

Но основное преимущество в том, что операции с присваиванием выполняются быстрее, чем их эквивалентная форма. При их использовании компилятор генерирует оптимальный код, что способствует увеличению скорости выполнения программ.

Приоритет операций и порядок ассоциирования

Сводная таблица всех операций в языке С в порядке приоритета:

Операции	***Ассоциирование***	***Приоритет***
`() [] -> .`	→ слева направо	высокий (15)
`! ~ ++ -- + - * & (тип) sizeof`	← **справа налево**	(14)
`* / %`	→ слева направо	(13)
`+ -`	→ слева направо	(12)
`<< >>`	→ слева направо	(11)
`< <= > >=`	→ слева направо	(10)
`== !=`	→ слева направо	(9)
`&`	→ слева направо	(8)
`^`	→ слева направо	(7)
`|`	→ слева направо	(6)
`&&`	→ *слева направо*	(5)
`||`	→ *слева направо*	(4)
`?:`	← **справа налево**	(3)
`= += -= *= /= %= &= ^= |= <<= >>=`	← **справа налево**	(2)
`,`	→ слева направо	низкий (1)

Операторы цикла

Виды циклических конструкций

Цикл предназначен для выполнения повторяющихся действий. Цикл содержит группу операторов (*тело цикла*), выполнение которых повторяется необходимое количество раз. Каждое такое повторение называется *итерацией* цикла.

Существуют циклы с явно заданным числом повторений (с *параметром* или *счетчиком*). И циклы с неизвестным числом повторений (*итеративные*), которые проводят проверку заданного *условия* до или после выполнения тела цикла. Последние называются циклы с *предусловием* и циклы с *постусловием*, соответственно.

Цикл с предусловием `while`

Синтаксис оператора с предусловием содержит ключевое слово `while` (что означает «**пока**»), выражение условия и выполняемый оператор :

```
while (Выражение)
   Оператор
```

он похож на сокращенную форму оператора условия `if`. Результат условного `Выражения` принимает числовое значение.

Оператор цикла `while` (так же, как оператор `if`) вначале проверяет значение `Выражения`. Если оно истинно (принимает ***ненулевое*** значение), то выполняется тело цикла, содержащее один `Оператор`. Затем проверка `Выражения` повторяется еще раз. Так продолжается до тех пор, пока `Выражения` не станет ложным (примет нулевое значение).

Если в теле цикла требуется использовать несколько операторов:

```
while (Выражение) {
   Оператор1
   Оператор2
   ...
   ОператорN
}
```

! Замечание: ***Перед вызовом*** *любого оператора цикла необходимо провести подготовку – назначить* ***начальные значения*** *всем переменным, которые изменяются в теле цикла.*

Число повторений (итераций) оператора `while`, как правило, заранее неизвестно.

Для завершения цикла в его теле **обязательно** должны содержаться операторы, оказывающие **влияние на истинность** условного `Выражения`. Иначе цикл будет выполняться бесконечно – произойдет *зацикливание* программы.

Предотвратить зацикливание позволяет правило для итеративных циклов: ***переменная, которая участвует в*** `Выражении`***, обязательно должна изменяться в теле цикла.***

Пример. Суммирование целых чисел от 1 до 20:

```
#include <stdio.h>
int main(void) {
   int i, sum;  /* i – текущее число, sum – сумма */
   sum = 0;
   i = 1;  /* установка начальных значений */
   while (i<=20) {  /* проверка условного выражения цикла */
      sum += i;  /* накапливается сумма (sum=sum+i) */
      i++;  /* переход к следующему числу, и изменение условия */
   }
   printf("Сумма чисел от 1 до 20 равна %d\n", sum);
   return 0;
}
```

Цикл с постусловием do-while

Синтаксис оператора с постусловием содержит ключевые слова `do` и `while` (что означает «**выполнить**» и «**пока**»):

```
do {
   Оператор1
   Оператор2
   ...
   ОператорN
} while (Выражение);
```

Если в теле цикла содержится только один оператор, то фигурные скобки ставить не обязательно. Однако чтобы визуально не перепутать окончание цикла `do-while` с началом цикла с предусловием `while`, фигурные скобки желательно ставить всегда.

Оператор `do-while` вначале выполняет тело цикла, а затем проверяет значение `Выражения`. Далее аналогично циклу `while`. Если `Выражение` истинно (принимает ***ненулевое*** значение), то тело цикла повторяется еще раз. Так продолжается до тех пор, пока `Выражение` не станет ложным (примет нулевое значение).

Число итераций цикла `do-while`, как правило, также заранее не известно. Но тело цикла обязательно будет выполнено **хотя бы один раз.** А приведенное выше правило для итеративных циклов позволяет избежать зацикливания.

Тот же пример суммирования чисел:

```
#include <stdio.h>
int main(void) {
   int i, sum;  /* i – текущее число, sum – сумма */
   sum = 0;
   i = 1;  /* установка начальных значений */
   do {
      sum += i;  /* накапливается сумма (sum=sum+i) */
      i++;  /* переход к следующему числу, и изменение условия */
   } while (i<=20);  /* проверка условного выражения цикла */
   printf("Сумма чисел от 1 до 20 равна %d\n", sum);
   return 0;
}
```

Цикл со счетчиком `for`

В операторе цикла `for` может использоваться *переменная-параметр* (*счетчик*), которая на каждой итерации цикла изменяет свое значение согласно заданным правилам.

Синтаксис оператора содержит ключевое слово `for` (что означает «**для**»):

```
for (ВыражИниц; ВыражУсл; ВыражИзмен)
   Оператор
```

Эта конструкция эквивалентна следующей:

```
ВыражИниц;
while (ВыражУсл) {
   Оператор;
   ВыражИзмен;
}
```

В общем виде принцип работы цикла `for` следующий. Вначале один раз выполняется `ВыражИниц` – инициализация счетчика цикла, т.е. выражение с присваиванием начального значения некоторой переменной-параметру цикла. Затем проводится проверка `ВыражУсл` – условного выражения цикла. Если оно истинно (принимает ***ненулевое*** значение), то выполняется `Оператор` тела цикла. В конце итерации проводится изменение счетчика цикла согласно выражению `ВыражИзмен` (например, приращение счетчика на заданную величину). После этого выполняется следующая итерация цикла `for`, т.е. следующая проверка условного выражения.

Так продолжается до тех пор, пока `ВыражУсл` не станет ложным (примет нулевое значение). Следует заметить, что после этой последней проверки изменение счетчика не произойдет, а цикл сразу окончится.

Если в теле цикла требуется использовать несколько операторов:

```
for (ВыражИниц; ВыражУсл; ВыражИзмен) {
   Оператор1
   Оператор2
   ...
   ОператорN
}
```

Как правило, оператор `for` используется для организации циклов с числом повторений, которое известно заранее, к моменту вызова цикла. Желательно использовать переменную-параметр (счетчик) целочисленного типа.

Тот же пример суммирования чисел:

```
#include <stdio.h>
int main(void) {
   int i, sum;  /* i – текущее число (счетчик), sum – сумма */
   sum = 0;  /* установка начального значения суммы */
   for (i=1; i<=20; i++)  /* управление циклом */
      sum += i;  /* накапливается сумма (sum=sum+i) */
   printf("Сумма чисел от 1 до 20 равна %d\n", sum);
   return 0;
}
```

Значение счетчика цикла может быть как угодно изменено в теле самого цикла. Причем при выходе из цикла счетчик сохраняет свое значение.

```
int i, k=0;
for (i=1; i<=10; i+=2) {
   i += 3;  /* такое действие допускается */
   printf("k=%d i=%d\n", ++k, i);
}   /* две итерации: «k=1 i=4» и «k=2 i=9» */
printf("Result: i=%d\n", i);  /* результат: «i=11» */
```

Зацикливание (бесконечный цикл)

Любое из трех выражений цикла `for` можно опустить, при этом точки с запятыми должны оставаться на своих местах. Если опустить выражения `ВыражИниц` и `ВыражИзмен`, то соответствующие операции не будут выполняться. Если же опустить проверку условия `ВыражУсл`, то по умолчанию считается, что условие продолжения цикла всегда истинно, и такая конструкция станет *бесконечным циклом* (*зациклится*):

```
for ( ; ; ) {      /* зацикливание */
   ...
}
```

Такой цикл должен прерываться другими способами, например с помощью оператора `break`.

Сравнение операторов цикла

Большинство задач можно решить, используя любой из операторов цикла, но в некоторых случаях предпочтительнее использовать один из них.

Самым универсальным из операторов цикла является цикл с предусловием `while`, т.к. операторы в его теле могут быть не выполнены ни разу. Доказано, что любая программа может быть написана, используя только итеративную конструкцию `while` и оператор условия `if`.

Цикл `for` следует предпочесть тогда, когда есть простая инициализация и простые правила изменения счетчика цикла, поскольку в этом случае все управляющие элементы цикла удобно сгруппированы вместе в его заголовке.

Обобщение рассмотренного примера суммирования чисел от 1 до 20:

```
sum = 0;
i = 1;
while (i<=20) {
   sum += i;
   i++;
}
```

```
sum = 0;
i =1;
do {
   sum += i;
   i++;
} while (i<=20);
```

```
sum = 0;
for (i=1; i<=20; i++)
   sum += i;
```

Примеры

Вычисление факториала n!

```
#include <stdio.h>
#include <locale.h>
int main(void) {
   int      n, i;  /* n – число, i – счетчик */
   long int fact;  /* fact – значение факториала */
   setlocale(LC_ALL,"");
   printf("Введите число n: ");   scanf("%d", &n);
   fact = 1;  /* установка начального значения факториала */
   for (i=1; i<=n; i++)  /* управление циклом */
      fact *= i;  /* накапливается значение (fact=fact*i) */
   printf("Факториал n!=%d\n", fact);
   return 0;
}
```

Далее в примерах при использовании национальных стандартов будем опускать подключение заголовочного файла `<locale.h>` и вызов функции `setlocale()`.

Вычисление НОД двух чисел

Наибольший общий делитель (НОД) двух чисел – это наибольшее целое число, на которое оба числа делятся без остатка.

Алгоритм Евклида основан на следующем свойстве целых чисел. Пусть целые неотрицательные числа a и b одновременно не равны нулю и $a \geq b$. Тогда, если $b=0$, то $НОД(a,b)=a$. Если $b \neq 0$, то для чисел a, b и r, где r – остаток от деления a на b, выполняется равенство $НОД(a,b)=НОД(b,r)$. Действительно, $r = a \bmod b = a - (a \operatorname{div} b) \cdot b$, где mod и div – остаток и частное от целочисленного деления соответственно. Если какое-то число делит нацело и a и b, то из приведенного равенства следует, что оно делит нацело и числа r и b.

Реализация алгоритма Евклида:

```
#include <stdio.h>
int main(void) {
   long int a, b;
   printf("Введите два числа, не равные нулю: ");
   scanf("%d %d", &a, &b);
   do {
      if (a>b)  a %= b;
      else      b %= a;
   } while (a!=0 && b!=0);
   printf("НОД =%d\n", a+b);
   return 0;
}
```

Вложенный цикл

Вложенность

В теле любого оператора цикла (*внешнего*) могут находиться другие операторы цикла (*внутренние*). Все операторы внутреннего цикла должны полностью располагаться в теле внешнего цикла.

Пример вложенного цикла `for`:

```
for (ВыражИниц1; ВыражУсл1; ВыражИзмен1) {
   Оператор
   Оператор
   for (ВыражИниц2; ВыражУсл2; ВыражИзмен2) {
      Оператор
      Оператор
      Оператор
   }
   Оператор
   Оператор
}
```

Внутренний цикл выполняется полностью на каждой итерации внешнего цикла.

Пример. Вывод на экран таблицы умножения:

```
#include <stdio.h>
int main(void) {
   int i, j;   /* счетчики */
   for (i=1; i<=9; i++) {   /* внешний цикл */
      for (j=1; j<=9; j++)   /* внутренний цикл */
         printf("%d*%d=%2d  ", i, j, i*j);
      printf("\n");
      }
   return 0;
}
```

Задачи на перебор всех вариантов

Старинная задача о покупке скота. Сколько можно купить быков, коров и телят (бык стоит 10 рублей, корова – 5 рублей, теленок – 0,5 рубля), если на 100 рублей надо купить 100 голов скота.

Обозначим: b – количество быков, k – количество коров, t– количество телят. Тогда условие задачи можно записать в виде системы из двух уравнений: 10b+5k+0,5t=100 и b+k+t=100. Преобразуем первое: 20b+10k+t=200.

Запишем ограничения. На 100 рублей можно купить:

- не более 10 быков, $0 \le b \le 10$;
- не более 20 коров, $0 \le k \le 20$;
- не более 200 телят, $0 \le t \le 200$.

```
#include <stdio.h>
int main(void) {
   int b, k, t;
   for (b=0; b<=10; b++)
      for (k=0; k<=20; k++)
         for (t=1; t<=200; t++)
            if ((20*b+10*k+t==200)&&( b+k+t==100))
               printf("быков %d, коров %d, телят %d\n ",b,k,t);
   return 0;
}
```

В данной программе значение переменной `b` изменяется 11 раз, `k` – 21 раз, `t` – 201 раз. Таким образом, условие в операторе `if` проверяется 11×21×201=46431 раз. Но если известно количество быков и коров, то количество телят можно вычислить по формуле t=100–(b+k). Цикл по переменной `t` исключается:

```
#include <stdio.h>
int main(void) {
   int b, k, t;
   for (b=0; b<=10; b++)
      for (k=0; k<=20; k++) {
         t=100-(b+k);
         if (20*b+10*k+t==200)
            printf("быков %d, коров %d, телят %d\n",b,k,t);
         }
   return 0;
}
```

При этом решении условие проверяется 11×21=231 раз, уменьшается процессорное время, необходимое на обработку программы. В данной задаче количество проверок можно еще уменьшить.

Оптимизация программы

При вычислениях часто используются циклы, содержащие одинаковые операции для каждого слагаемого. Например, общий множитель:

```
sum=0;
for (i=1; i<=1000; i++)
   sum+=a*i;
```

Другая форма записи этого цикла содержит всего одно умножение, вместо 1000, поэтому более экономна:

```
sum=0;
for (i=1; i<=1000; i++)
   sum+=i;
sum*=a;
```

Внутри цикла могут встречаться выражения, фрагменты которых никак не зависят от переменной-счетчика этого цикла.

```
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    for (k=1; k<=p; k++)
       x=a*i*j+k;
```

Здесь слагаемое `a*i*j` в выражении для вычисления `x` не зависит от переменной `k`, поэтому для уменьшения количества вычислений можно использовать вспомогательную переменную `y`.

```
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++) {
     y=a*i*j;
     for (k=1; k<=p; k++)
        x=y+k;
     }
```

Иногда ***производительность программы целиком зависит от того, как запрограммирован цикл***.

На практике, однако, чтобы не внести новые ошибки, для устранения которых потребуется время, ***к оптимизации уже существующей и отлаженной программы следует прибегать, только если в этом есть реальная необходимость***.

Операторы перехода

Оператор `break`

Оператор `break` осуществляет принудительный выход из циклов `for`, `while` и `do-while`, аналогично выходу из оператора `switch`.

Когда внутри цикла встречается оператор `break`, выполнение цикла немедленно прекращается (без проверки каких-либо условий) и управление передается оператору, следующему за ним. Например:

```
int i;
for (i=1; i<10; i++) {
   if (i==5)
      break;
   printf("%d ",i);
}
printf("\nПоследнее значение i=%d\n",i);
```

Здесь на пятой итерации цикла `for` выполнение цикла прерывается. На экран будет выведено:

```
1 2 3 4
Последнее значение i=5
```

Если оператор `break` находится внутри вложенного цикла, то прекращается выполнение только этого внутреннего цикла.

Оператор `continue`

Оператор `continue` прерывает текущую итерацию цикла и осуществляет переход к следующей итерации. При этом все операторы до конца тела цикла пропускаются.

В цикле `for` оператор `continue` вызывает выполнение операторов приращения и проверки условия цикла. В циклах `while` и `do-while` – передает управление операторам проверки условий цикла.

В предыдущем примере для цикла `for` заменим `break` на `continue`.

```
int i;
for (i=1; i<10; i++) {
   if (i==5)
      continue;
   printf("%d ",i);
   }
printf("\nПоследнее значение i=%d\n",i);
```

Тогда на пятой итерации значение переменной `i=5` не будет распечатано:

```
1 2 3 4 6 7 8 9
Последнее значение i=10
```

Если `continue` находится внутри вложенного цикла, то прекращается выполнение итерации только этого внутреннего цикла.

! Замечание: *Операторы* **`break`** *и* **`continue`** *нарушают нормы структурного программирования, результаты их работы могут быть достигнуты другими средствами. Поэтому избегать их использования считается хорошим стилем программирования. С другой стороны, принудительное прерывание работы циклов может приводить к ускорению работы программы. Таким образом, операторы* **`break`** *и* **`continue`** *нужно применять только там, где это действительно необходимо.*

Оператор `goto`

Оператор `goto` осуществляет безусловный переход к метке. При этом в программе должен присутствовать оператор с присвоенной ему меткой:

```
goto Метка;
...
Метка: Оператор
```

Правила именования меток те же, что и для любых идентификаторов.

! Замечание: *Использование оператора* **`goto`** *следует* ***избегать****, поскольку оно нарушает нормы структурного программирования.*

Пожалуй, единственный случай, когда *использование* `goto` ***опытным программистом (!)*** *может быть оправдано – это* ***принудительный выход из вложенного набора циклов при возникновении ошибки***. Например:

```
while (...) {
   for (...) {
      for (...) {
         Операторы
         if (Ошибка)
            goto label;  /* переход к метке */
      }
      Операторы
   }
   Операторы
}
Операторы
label: УстранениеОшибки;
```

Переполнение и системно-зависимые константы

Границы целочисленных типов

Поскольку в языке С размер памяти, которую занимает каждый из типов данных, зависит от реализации компилятора, библиотек и аппаратной части, то и границы диапазонов допустимых значений могут быть разными.

В заголовочном файле `<limits.h>` определены константы, описывающие размеры **целочисленных** типов данных независимо от конкретной реализации. Вот некоторые из них:

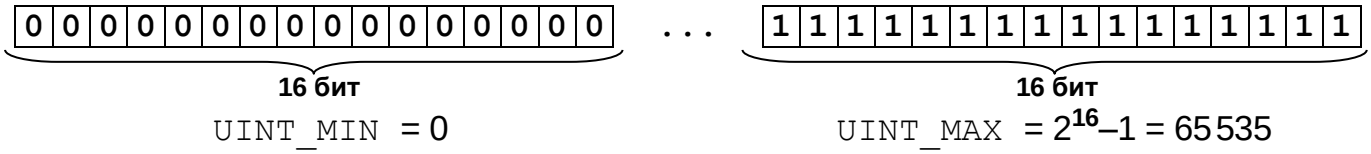
Константа	***Описание***
`INT_MAX`	максимальное значение `int`
`INT_MIN`	минимальное значение `int`
`LONG_MAX`	максимальное значение `long int`
`LONG_MIN`	минимальное значение `long int`
`UINT_MAX`	максимальное значение `unsigned int`
`ULONG_MAX`	максимальное значение `unsigned long int`

Данные константы удобно использовать в программах для отслеживания возможного ***переполнения***.

Переполнение

Рассмотрим упрощенную схему хранения переменной в памяти компьютера.

Если для хранения переменных типа `unsigned int` используется 2 байта, тогда говорят, что тип `int` является ***16-битным*** (в каждом байте – по 8 бит: 8×2=16). Значит, всего с его помощью можно закодировать $\mathbf{2^{16}}$ различных значений. Для данного типа это – целые числа из диапазона $[0; 2^{16}-1] = [0; 65\,535]$, которые имеют двоичные коды:



`UINT_MIN` = 0 `UINT_MAX` = $2^{16}-1$ = 65 535

В общем случае, если для переменной типа `unsigned int` используется n байт (т.е. тип ***8×n***-битный), всего можно закодировать $\mathbf{2^{8n}}$ различных значений из диапазона $[0; 2^{8n}-1]$, тогда: `UINT_MIN` = 0, `UINT_MAX` = $2^{8n}-1$.

Пусть в программе объявлена переменная `a` описанного 16-битного типа, значение которой задает пользователь. Допустим, значение этой переменной требуется увеличить на 10 000:

```
unsigned int a;
printf("Введите число a: ");
scanf("%d", &a);
a += 10000;
```

Если пользователь введет значение `a` больше, чем 55 535, то корректно выполнить операцию «сложение с присваиванием» (+=) будет невозможно – для хранения нового значения переменной 16 бит уже недостаточно.

При попытке выполнить сложение произойдет ***переполнение*** памяти, выделенной для хранения переменной `a`. В ней будет содержаться некорректное значение.

Отслеживание переполнения

Чтобы избежать переполнения в описанном примере перед выполнением операции `+=` необходимо провести проверку, не выйдет ли результат за границы диапазона возможных значений. Переполнения не произойдет, если выполняется неравенство:

$$a+10000 \le 65535,$$

где 65535=`UINT_MAX`

Казалось бы, в программе достаточно выполнить проверку:

```
if (a+10000 <= UINT_MAX)  a += 10000;    /* НЕВЕРНО! */
```

но здесь переполнение все равно произойдет при проверке условного выражения оператора `if` в операции сложения `a+10000`. Затем полученное некорректное значение будет сравниваться с константой `UINT_MAX`. Такая проверка была бы бессмысленна.

Поэтому, обычно ***для проверки возможного переполнения при выполнении арифметической операции используют операцию, обратную к ней***. Проверочное неравенство можно представить так:

$$a \le 65535 - 10000,$$

тогда код:

```
if (a <= UINT_MAX-10000)  a += 10000;
```

не приведет к переполнению при проверке условного выражения.

Другой пример. Пусть для переменной типа `int`, значение которой введено пользователем, необходимо проверить, приведет ли к переполнению умножение этой переменной на `2`:

```
#include <stdio.h>
#include <limits.h>
int main(void) {
   int n;
   printf("Макс.значение типа int = %d\n", INT_MAX);
   printf("Введите число n: ");
   scanf("%d", &n);
   if (n >= INT_MAX/2 + 1)
      printf("Увеличение числа n более, чем в 2 раза
              приведет к переполнению типа int.\n");
   return 0;
}
```

Упражнения

Упражнение 2.1

Составить программу, которая запрашивает у пользователя два целых числа и выводит на экран наибольшее из них. В случае если числа равны, программа не выводит ничего.

Упражнение 2.2

Составить программу, которая запрашивает у пользователя целое число и, если это число положительное и четное, выводит на экран соответствующее сообщение.

Упражнение 2.3

Составить программу, которая запрашивает у пользователя два целых числа. Если одно из чисел делит другое нацело (без остатка), то программа выводит на экран результат целочисленного деления, в противном случае программа выводит их сумму.

Упражнение 2.4

Составить программу, которая с помощью цикла определяет, является ли простым число, введенное пользователем. Простым называется натуральное число, которое делится только на 1 и на само себя.

Упражнение 2.5

Составить программу, которая запрашивает у пользователя целые числа (до тех пор, пока пользователь не введет число 0) и вычисляет их сумму.

Упражнение 2.6

Составить программу, которая вычисляет степень числа n^k, натуральные числа ***n*** и ***k*** вводятся пользователем.

Упражнение 2.7

Составить программу, которая находит все простые числа в диапазоне от 1 до 1000. Простым называется натуральное число, которое делится только на 1 и на само себя.

Упражнение 2.8

Составить программу, которая запрашивает у пользователя число *n*, и выводит на экран *n* строк в следующем виде:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
. . .
```

Упражнение 2.9

Составить программу, которая запрашивает у пользователя два числа типа `int` (в том числе отрицательные) и вычисляет их сумму (произведение) с предотвращением возможного переполнения.