Лекция 2 – введение в синтаксис языка C++

Воронин Андрей Андреевич

Кафедра прикладной математики и информатики

30 сентября 2019 г.

Типы данных

- Целочисленные:
  1. **`char`**
  2. **`short int`**
  3. **`int`**
  4. **`long int`**
  5. **`long long int`**

  Могут быть беззнаковыми (**`unsigned`** или **`uint8_t`**)
  - $-2^{n-1} \ldots (2^{n-1} - 1)$ (n – число бит)
  - $0 \ldots (2^n - 1)$ (для **`unsigned`**)
- Числа с плавающей точкой:
  1. **`float`**, 4 байта, 7 значащих цифр.
  2. **`double`**, 8 байт, 15 значащих цифр.
- Логический тип данных **`bool`**.
- Пустой тип **`void`**.

Типы данных

- Целочисленные:
  1. **`int8_t`**
  2. **`int16_t`**
  3. **`int32_t`**
  4. **`int32_t`**
  5. **`int64_t`**

  Могут быть беззнаковыми (**`unsigned`** или **`uint8_t`**)
  - $-2^{n-1} \dots (2^{n-1} - 1)$ (n – число бит)
  - $0 \dots (2^n - 1)$ (для **`unsigned`**)
- Числа с плавающей точкой:
  1. **`float`**, 4 байта, 7 значащих цифр.
  2. **`double`**, 8 байт, 15 значащих цифр.
- Логический тип данных **`bool`**.
- Пустой тип **`void`**.

Литералы

- Целочисленные:
  1. `'a'` – код буквы `'a'`, тип **`char`**.
  2. `42` – все целые по умолчанию типа **`int`**.
  3. `123456789L` – суффикс `'L'` соответствует типу **`long`**.
  4. `1703U` – суффикс `'U'` соответствует типу **`unsigned int`**.
  5. `1703UL` – суффикс `'UL'` соответствует типу **`unsigned long`**.
- Числа с плавающей точкой:
  1. `3.14` – все числа с точкой по умолчанию **`double`**.
  2. `3.1415F` – суффикс `'F'` соответствует типу **`float`**.
  3. `3.0E8` – соответствует $3 \cdot 10^8$
- `true false` – значения типа **`bool`**.
- Строки задаются в двойных кавычках: `"Test string"`.

Переменные

- При определении переменной указывается её тип. При определении можно сразу задать начальное значение (инициализация).

```
int i = 10;
short j = 20;
bool b = false;

unsigned long l = 123321;

double x = 13.5, y = 3.1415;
float z;
```

- Нельзя оставлять переменные неинициализированными.
- Нельзя создать переменную пустого типа **void**.

Операции

- Оператор присваивания: **=**.
- Арифметические:
  1. бинарные: **+ - * / %**,
  2. унарные: **++ --**.
- Логические:
  1. бинарные: **&& ||**
  2. унарные: **!**.
- Сравнения: **== != > < >= <=**.
- Приведения типов: **(type)**.
- Сокращенные версии бинарных операторов: **+= -= *= /= %=**.

```
int i = 10;
i = (20 * 3) % 7;

int k = i++;
int l = --i;

bool b = !(k == 1);

b = (a == 0) || (1 / a <
↪   1);

double d = 3.1415;
float f = (int) d;

// d = d * (i + k);
d *= i + k;
```

Целочисленные типы в C++

Все целочисленные типы (кроме `char`) являются знаковыми. Беззнаковые версии типов определяется с ключевым словом `unsigned`, например: `unsigned short int`, `unsigned int` или `unsigned long int`. Для симметрии в языке предусмотрено явное указание того, что тип является знаковым — ключевое слово `signed` (используется редко). Более того, C++ допускает использование следующих сокращений:

- `unsigned` вместо `unsigned int`,
- `short` вместо `short int`,
- `long` вместо `long int`.

Операции инкремента и декремента:

Операции инкремента и декремента

```
int a = 10; // a = 10
int b = ++a; // префиксный инкремент возвращает новое
↪  значение => b = 11 и a = 11
int c = a++; // постфиксный инкремент возвращает
↪  старое значение => c = 11 и a = 12
```

Преобразования типов в операторах

Выражениям так же как и значениям в C++ приписывается некоторые типы. Например, если `a` и `b` — это переменные типа `int`, то выражения `(a + b)`, `(a - b)`, `(a * b)` и `(a / b)` тоже будут иметь тип `int`. Важно всегда понимать, какой тип у выражения, которое вы написали в программе. Давайте проиллюстрируем это на следующем примере:

```
int a = 20;
int b = 50;
double d = a / b; // d = 0, оба аргумента
 ↪  целочисленные, а значит деление целочисленное
```

Для того чтобы исправить эту ситуацию хотя бы один из аргументов оператора деления должен иметь типа double. Этого можно добиться при помощи уже известного нам оператора приведения типов:

```
double d = (double)a / b; // d = 0.4
```

Почему это сработало? Дело в том, что операторы для встроенных типов C++ *всегда* работают с *одинаковыми типами* аргументов. Если аргументы имеют разные типы, то происходит преобразование типов (promotion).

Правило преобразования встроенных типов в операторах

Рассмотрим выражение `(a + b)`, где вместо `'+'` может стоять любой другой подходящий оператор.

1. Если один из аргументов имеет числовой тип с плавающей точкой, то второй аргумент приводится к этому типу (например, при сложении `double` и `int` значение типа `int` приводится к `double`).
2. Если оба аргумента имеют числовой тип с плавающей точкой, то выбирается наибольший из этих типов (например, при сложении `double` и `float` значение типа `float` приводится к `double`).
3. Если оба аргумента целочисленные, но их типы меньше `int`, то оба аргумента приводятся к типу `int` (например, при сложении двух значений типа `char` они оба сначала приводятся к `int`).
4. Если оба аргумента целочисленные, то аргумент с меньшим типом приводится к типу второго аргумента (например, при сложении `long` и `int` значение типа `int` приводится к `long`).
5. Если оба аргумента целочисленные и имеют тип одного размера, то предпочтение отдаётся беззнаковому типу (например, при сложении `int` и `unsigned int` значение типа `int` приводится к `unsigned int`).

Следствия неявных преобразований типов

1. Следите за тем, какие типы участвуют в выражении, от этого может зависеть его значение.
2. Не стоит использовать целочисленные типы меньше `int` в арифметических выражениях, они всё равно будут приведены к `int`.
3. **Не стоит смешивать `unsigned` и `signed`** типы в одном выражении, это может привести к неприятным последствиям.

Для иллюстрации последнего следствия давайте рассмотрим следующий пример:

```
unsigned from = 100;
unsigned to = 0;
for (int i = from; i >= to; --i) {  ....  }
```

Инструкции

- Выполнение состоит из последовательности *инструкций*.
- Инструкции выполняются одна за другой.
- Порядок вычислений внутри инструкций неопределен.

```
/* unspecified behavior */
int i = 10;
i = (i+=6) + (i * 4);
```

- Блоки имеют вложенную область видимости:

```
int k = 10;
{
    int k = 5 * i; // не видна за пределами блока
    i = (k += 5) + 5;
}
k = k + 1;
```

Условные операторы

- Оператор **if**:

```
int d = b * b - 4 * a * c;
if (d > 0)
{
    roots = 2;
}
else if (d == 0)
{
    roots = 1;
}
else
{
    roots = 0;
}
```

- Тернарный условный оператор:

```
int roots = 0;
if (d >= 0)
    roots = (d > 0) ? 2 : 1;
```

Циклы

- Цикл **while**:

```
int squares = 0;
int k = 0;
while (k < 0)
{
    squares += k * k;
    k = k + 1;
}
```

- Цикл **for**:

```
for (int k = 0; k < 10; k = k + 1)
{
    squares += k * k;
}
```

- Для выхода из цикла используется оператор **break**.

Цикл do-while

В C++ существует вариация цикла `while`, которая называется `do-while`. В отличие от обычного `while` в `do-while` условие проверяется не до, а *после* итерации. Т.е. такой цикл всегда имеет хотя бы одну итерацию.

```
int i = 10;
int sum = 0;
while (i < 10) {
    sum += i;
}
// sum = 0
```

```
int i = 10;
int sum = 0;
do {
    sum += i;
} while(i < 10);
// sum = 10
```

Управление циклами

Ранее был упомянут оператор **break**, который используется для выхода из цикла. Рассмотрим его действие на следующем примере.

```
int a = 323;
int b = 2;
while ( b <= a ) {
    if ( a % b == 0 )
    break; // выйти из цикла
    b = b + 1;
}
```

В данном случае b будет равен 17, т.к. $323 = 17 \times 19$.

Ещё один оператор, который можно использовать с циклами — это оператор **continue**. Оператор **continue** прерывает текущую итерацию цикла и переходит к следующей.

```
int sum = 0;
for ( int i = 1; i <= 100; ++i ) {
    if ( (i % 17 == 0) || (i % 19 == 0) )
    continue; // перейти к следующей итерации
    sum += i;
}
```

Потоки ввода и вывода

Вывод различных типов в C++

```
#include <iostream>

int main()
{
    int i = 42;
    double d = 3.14;
    const char *s = "C-style string";

    std::cout << "This is an integer " << i << "\n";
    std::cout << "This is a double " << d << "\n";
    std::cout << "This is a \"" << s << "\"\n";

    return 0;
}
```

Потоки ввода и вывода

Вывод и ввод в C++

```
#include <iostream>

int main()
{
    int i = 42;
    double d = 3.14;

    std::cout << "Enter an integer and a double:\n";
    std::cin >> i >> d;
    std::cout << "Your input is " << i << ", " << d <<
    ↪   "\n";

    return 0;
}
```

Посимвольный ввод

```
char c = '\0';
while (cin.get(c)) { // на каждой итерации считываем
 ↪  один символ в переменную c
    /* здесь можно пользоваться значением прочитанным
     ↪  в переменную c */
    if (c != 'a')
    cout << c; // выводим символ, если он не равен 'a'
}
```

Функции

- В сигнатуре функции указывается тип возвращаемого значений и типы параметров.
- Ключевое слово return возвращает значение.

```
double square ( double x ) {
    return x * x ;
}
```

- Переменные, определённые внутри функций, — *локальные*.
- Функция может возвращать **void**
- Параметры передаются по значению (копируются).

```
void strange ( double x , double y ) {
    x = y ;
}
```

Макросы

- Макросами в C++ называют инструкции препроцессора.
- Препроцессор C++ является самостоятельным языком, работающим с произвольными строками.
- Макросы можно использовать для определения функций:

```
int max1 ( int x , int y ) {
    return x > y ? x : y ;
}
#define max2 (x , y ) x > y ? x : y
a = b + max2 (c , d ); // b + c > d ? c : d;
```

- Препроцессор “не знает” про синтаксис C++

Макросы

- Параметры макросов нужно оборачивать в скобки:

```
#define max3 (x , y ) (( x ) > ( y ) ? (x ) : ( y
↪  ))
```

- Это не избавляет от всех проблем:

```
int a = 1;
int b = 1;
int c = max3 (++ a , b );
// c = ((++a) > (b) ? (++a) : (b))
```

- Определять функции через макросы — плохая идея.
- Макросы можно использовать для условной компиляции:

```
#ifdef DEBUG
// дополнительные проверки
#endif
```

Макросы

Предположим, что в программе определён макрос sqr.

```
#define sqr(x) x * x
```

Какое значение будет иметь следующее выражение?

```
sqr(3 + 0)
```

Ввод-вывод

- Будем использовать библиотеку <iostream>

```
#include <iostream>
using namespace std ;
```

- Ввод:

```
int a = 0;
int b = 0;
cin >> a >> b ;
```

- Вывод:

```
cout << " a + b = " << ( a + b ) << endl;
```

Простая программа

```
#include <iostream>
using namespace std ;
int main ()
{
    int a = 0;
    int b = 0;
    cout << " Enter a and b : " ;
    cin >> a >> b ;
    cout << " a + b = " << ( a + b ) << endl ;
    return 0;
}
```