Целочисленные типы

Введение в целочисленные типы

Один из целочисленных типов – это тип int. Начнём с проблемы, которая может возникнуть при работе с ним. Возьмем для примера задачу «Средняя температура. Дан набор наблюдений за температурой, в виде вектора t (значения 8, 7 и 3). Нужно найти среднее арифметическое значение температуры за все дни и затем вывести номера дней, в которые значение температуры было больше, чем среднее арифметическое. Должно получиться (8+7+3)/3 = 6.

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

int main() {
    vector<int> t = {8, 7, 3}; // вектор с наблюдениями
    int sum = 0; // переменная с суммой
    for (int x : t) { // проитерировались по вектору и нашли суммарную температуру
        sum += x;
    }
    int avg = sum / t.size(); // получили среднюю температуру
    cout << avg << endl;
    return 0;
} // вывод программы будем писать последним комментарием листинга
// 6
```

Но в этой задаче есть ограничение: гарантируется, что все значения температуры не отрицательные. Если в таком решении в исходном векторе будут отрицательные значения температуры, например, –8, –7 и 3 (ответ (–8–7+3)/3 = –4), код работать не будет:

```
int main () {
    vector<int> t = {–8, –7, 3}; // сумму -4
    ...
    int avg = sum / t.size(); // t.size() не умеет хранить отрицательные числа
    cout << avg << endl;
    return 0;
}
// 1431655761
```

Это не –4. На самом деле мы от незнания неаккуратно использовали другой целочисленный тип языка C++. Он возникает в t.size() – это специальный тип, который не умеет хранить отрицательные числа. Размер контейнера отрицательным быть не может, и это беззнаковый тип. Какая еще бывает проблема с целочисленными типами? Очень простой пример:

```
int main () {
    int x = 2'000'000'000'; // для читаемости разбиваем на разряды кавычками
    cout << x << " "; // выводим само число
    x = x * 2;
    cout << x << " "; // выводим число, умноженное на 2
    return 0;
}
// 2000000000 -294967296
```

Итак, запускаем код и видим, что 4 миллиарда в переменную типа int не поместилось.

Особенности целочисленных типов языка C++:

1. В языке C++ память для целочисленных типов ограничена. Если вам не нужны целые числа размером больше 2 миллиардов, язык C++ для вас выделит вот ровно столько памяти, сколько достаточно для хранения числа размером 2 миллиарда. Соответственно, у целочисленных типов языка C++ ограниченный диапазон значений.
2. Возможны так же проблемы с беззнаковыми типами. Если бы в задаче 1 допускались отрицательные значения температуры, это то, что некоторые целочисленные типы языка C++ беззнаковые. Тем самым вы, сможете хранить больше положительных значений, но не сможете хранить отрицательные.

Виды целочисленных типов:

- int – стандартный целочисленный тип.
    1. auto x = 1; – как и любая комбинация цифр имеет тип int;
    2. Эффективен: операции с ним напрямую транслировались в инструкции процессора;
    3. В зависимости от архитектуры имеет размер 4 или 32 бита, и диапазон его значений от −231 до (231−1).
- unsigned int (unsigned) – беззнаковый аналог int.
    1. Диапазон его значений от 0 до (232−1). Занимает 4 байта.
- size_t – тип для представления размеров.
    1. Результат вызова size() для контейнера;
    2. 4 байта (до (232−1)) или 8 байт (до (264−1)). Зависит от разрядности системы.
- Типы с известным размером из модуля #include<cstdint>.

1. int32_t – знаковый, всегда 32 бита (от −231 до (231−1));
2. uint32_t – беззнаковый, всегда 32 бита (от 0 до (232−1));
3. int8_t и uint8_t всегда 8 бит; int16_t и uint16_t всегда 16 бит; int64_t и uint64_t всегда 64 бита.

Тип	Размер	Минимум	Максимум	Стоит ли выбрать его?
int	4 (обычно)	-2^{31}	$2^{31}-1$	по умолчанию
unsigned int	4 (обычно)	0	$2^{32}-1$	только положительные
size_t	4 или 8	0	$2^{32}-1$ или $2^{64}-1$	размер контейнеров
int8_t	1	-2^{7}	$2^{7}-1$	сильно экономить память
int16_t	2	-2^{15}	$2^{15}-1$	экономить память
int32_t	4	-2^{31}	$2^{31}-1$	нужно ровно 32 бита
int64_t	8	-2^{63}	$2^{63}-1$	недостаточно int

Узнаём размеры и ограничения типов:

Как узнать размеры типа? Очень просто:

```
cout << sizeof(int16_t) << " ";                 // размер типа в байтах. Вызывается от переменной
cout << sizeof(int) << endl;
 // 2 4
```

Узнаём ограничения типов:

```
#include <iostream>
#include <vector>
#include <limits> // подключаем для получения информации о типе
using namespace std;
int main() {
    cout << sizeof(int16_t) << " ";
    cout << numeric_limits<int>::min() <<" " <<numeric_limits<int>::max() << endl;
    return 0;
}
// 4 -2147483648 2147483647
```

Преобразования целочисленных типов

Начнём с эксперимента. Прибавим 1 к максимальному значению типа int.

```
#include <iostream>
#include <vector>
#include <limits> // подключаем для получения информации о типе
using namespace std;
 int main() {
    cout << numeric_limits<int>::max() + 1 << " ";
    cout << numeric_limits<int>::min() – 1 << endl;
    return 0;
}
// -2147483648 2147483647
```

Получилось, что max + 1 = min, а min - 1 = max. Теперь попробуем вычислить среднее арифметическое 1000000000+2000000000.

```
int x = 2'000'000000;
int y = 1'000'000000;
cout << (x + y) / 2 << endl;
// -647483648
```

Хоть их среднее вмещается в int, но программа сначала сложила x и y и получила число, не уместившееся в тип int, и только после этого поделила его на 2. Если в процессе случается переполнение, то и с результатом будет не то, что мы ожидаем. Теперь попробуем поработать с беззнаковыми типами:

```
int x = 2'000'000000;
unsigned int y = x; // сохраняем в переменную беззнакового типа 2000000000
unsigned int z = -z;
cout << x << " " << y << " " << -x << " " << z << endl;
// 2000000000 2000000000 -2000000000 2294967296
```

Если значение умещается даже в int, то проблем не будет. Но если записать отрицательное число в unsigned, мы получим не то, что ожидали. Возвращаясь к задаче о средней температуре, посмотрим, в чём была проблема:

```
vector<int> t = {-8, -7, 3};
int sum = 0; // знаковое
for (int x : t){
    sum += x;
} int avg = sum / t.size(); // sum / t.size(); уже беззнаковое, т.к. t.size() беззнаковое
cout << avg << endl;
```

Правила выведения общего типа:

1. Перед сравнениями и арифметическими операциями числа приводятся к общему типу;
2. Все типы размера меньше int приводятся к int;
3. Из двух типов выбирается больший по размеру;
4. Если размер одинаковый, выбирается беззнаковый.

Примеры:

Слева	Операция	Справа	Общий тип	Комментарий
int	/	size_t	size_t	больший размер
int32_t	+	int8_t	int32_t (int)	тоже больший размер
int8_t	*	uint8_t	int	все меньшие приводятся к int
int32_t	<	uint32_t	uint32_t	знаковый к беззнаковому

Для определения типа в самой программе можно просто вызвать ошибку компиляции и посмотреть лог ошибки. Изменим одну строчку:

```
int avg = (sum / t.size()) + vector<int>{}                    // прибавили пустой вектор;
```

И получим ошибку, в логе которой указано, что наша переменная avg имеет тип size_t. Теперь попробуем сравнить знаковое и беззнаковое число:

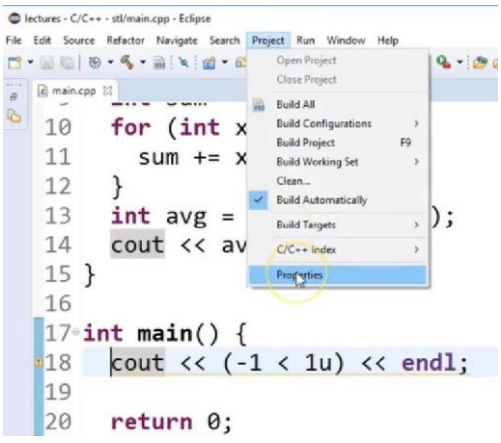
```
int main () {
   int x = –1;
   unsigned y = 1;
   cout << (x < y) << " ";
   cout << (–1 < 1u) << endl;  // суффикс u делает 1 типом unsigned по умолчанию
   return 0;
}
// 0 0
```

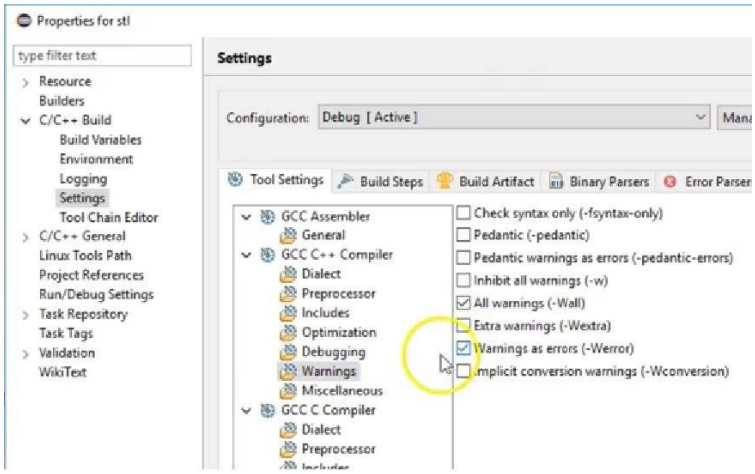
Как было сказано ранее, при операции между знаковым и беззнаковым типом обе переменные приводятся к беззнаковому. -1, приведённая к беззнаковому, становится очень большим числом, большим 1. Суффикс u также приводит 1 к unsigned, а операция < теперь сравнивает unsigned -1 и 1. Причём в данном случае компилятор предупреждает нас о, возможно, неправильном сравнении.

Безопасное использование целочисленных типов

Настроим компилятор:

Попросим компилятор считать каждый warning (предупреждение) ошибкой. Project → Properties → C/C++ → Build → Settings → GCC C++ Compiler → Warnings и отмечаем Warnings as errors.





После этого ещё раз компилируем код. И теперь каждое предупреждение считается ошибкой, которую надо исправить. Это одно из правил хорошо кода.

```
int main () {
    vector<int> x = {4, 5};
    for (int i = 0; i < x.size(); ++i) {
        cout << i << " " x[i] << endl;
    } return 0;
}
// error: "i < x.size()"... comparison between signed and unsigned"
```

Есть два способа это исправить: объявить i типом size_t или явно привести x.size() к типу int с помощью static_cast<int>(x.size()).

```
int main () {
    vector<int> x = {4, 5};
    for (int i = 0; i < static_cast<int>(x.size()); ++i) {
        cout << i << " " x[i] << " ";
    } return 0;
}
// 0 4 1 5
```

Исправляем задачу о температуре:

В задаче о температуре тоже приводим t.size() к знаковому с помощью оператора static_cast:

```
int main () {
    vector<int> t = {−8, −7, 3};                    // сумма -4
    ...
    int avg = sum / static_cast<int>(t.size());   // явно привели типы
    cout << avg << endl;
    return 0;
}
// -4
```

Предупреждений и ошибок не было. Всё, задача средней температуры для положительных и отрицательных значений решена! Таким образом если у нас где-то могут быть проблемы с беззнаковыми типами, мы либо следуем семантике и помним про опасности, либо приводим всё к знаковым с помощью static_cast.

Ещё примеры опасностей с беззнаковыми типами:

Переберём в векторе все элементы кроме последнего:

```
int main() {
    vector<int> v; // пустой вектор
    for (size_t i = 0; i < v.size() − 1; ++i) {    // v.size() - беззнаковый 0
        cout << v[i] << endl;
    } return 0;
}
```

После компиляции код падает. Вычтя из v.size() 1 мы получили максимальное значение типа size_t и вышли из своей памяти. Чтобы такого не произошло, мы перенесём единицу в другую часть сложения: Теперь на пустом векторе у нас все компилируется и вывод пустой. А на непустом выводит все элементы, кроме последнего. Напишем программу вывода элементов вектора в обратном порядке:

```
int main() {
    vector<int> v; // пустой вектор
    for (size_t i = 0; i + 1 < v.size(); ++i) { // v.size() - беззнаковый 0
        cout << v[i] << endl;
    } return 0;
}
```

Теперь на пустом векторе у нас все компилируется и вывод пустой. А на непустом выводит все элементы, кроме последнего. Напишем программу вывода элементов вектора в обратном порядке:

```
int main() {
    vector<int> v = {1, 4, 5};
    for (size_t i = v.size() - 1; i >= 0; --i) {
        cout << v[i] << endl;
    }
    return 0;
}
```

На пустом векторе, очевидно, будет ошибка. Но даже на не пустом он сначала 5, 4, 1, а затем очень много чисел и программа падает. Это произошло из-за того, что i >= 0 выполняется всегда и мы входим в бесконечный цикл. От этой проблемы мы избавимся «заменой переменной» для итерации:

```
for (size_t k = v.size() - 1; k > 0 ; --k) {
    size_t i = k - 1; // теперь
    cout << v[i] << endl;
}
```

Теперь всё работает нормально. В итоге, проблем с беззнаковыми типами помогают избежать:

- Предупреждения компилятора;
- Внимательность при вычитании из беззнаковых;
- Приведение к знаковым типам с помощью static_cast.

Условный оператор и циклы

Условный оператор if

Условный оператор if позволяет указать операции, которые должны выполниться при соблюдении некоторого условия, либо не выполниться, если это условие неверно.

Синтаксис оператора if в простейшем случае имеет вид:

```
if ( <условие> )
    <команда если верно>
```

Пусть пользователь вводит с консоли два числа, которые потом сравниваются между собой.

```
int a, b;
cin >> a >> b;
if (a == b)
    cout << "equal";
```

Если ввести два одинаковых числа, программа выводит «equal», иначе — ничего не выводит.

Оператор else позволяет указать утверждение, которое будет выполнено в случае, если условие не верно. Оператор else всегда идет в паре с оператором if и имеет следующий синтаксис:

```
if ( <условие> )
    <команда если верно> else
    <команда если неверно>
```

В результате, программу можно дополнить следующим образом.

```
int a, b;
cin >> a >> b;
if (a == b)
    cout << "equal" << endl;
else
    cout << "not equal" << endl;
```

Если ввести два одинаковых числа, программа выводит «equal», иначе — «not equal».

Если необходимо выполнить больше одной операции при выполнении условия, нужно использовать фигурные скобки:

```
if ( <условие> ) {
    ...
}
```

Например, можно вывести значения чисел: оба значения, если числа различны, и одно, если совпадают.

```
int a, b;

cin >> a >> b;

if (a == b) {
    cout << "equal" << endl;
    cout << a;
} else {
    cout << "not equal" << endl;
    cout << a << " " << b;
}
```

Здесь endl (end of line) — оператор, который делает перенос строки.

При работе с оператором if следует иметь в виду следующую особенность. Пусть дан такой код:

```
int a = -1;

if (a >= 0)
    if (a > 0)
        cout << "positive";
else
    cout << "negative";
```

Из-за отступов могло показаться, что оператор else относится к внешнему if, а на самом деле в такой записи он относится к внутреннему if. В C++, в отличие от Python, отступы не определяют вложенность. В итоге программа ничего не выводила в консоль.

Если явно расставить скобки, получится:

```
int a = -1;

if (a >= b) {
    if (a > 0)
        cout << "positive";
} else {
    cout << "negative";
```

```
}
```

В данном случае, как и ожидается, выведено «negative».

Из последнего примера можно сделать вывод, что следует всегда явно расставлять фигурные скобки, даже если выполнить необходимо всего одну команду.

Цикл while

Цикл while может быть полезен, если необходимо выполнять некоторые условия много раз, пока истинно некоторое условие.

```
while ( <условие> )
    <команда>
```

Пусть пользователь вводит число n. Требуется подсчитать сумму чисел от 1 до n.

```
int n = 5;
int sum = 0;
int i = 1;
while (i <= n) {
    sum += i;
    i += 1;
}
cout << sum;
```

Аналогом цикла while является так называемый цикл do-while, который имеет следующий синтаксис:

```
do {
    <команда>
} while ( <условие> );
```

Следующая программа является интерактивной игрой, в которой пользователь пытается угадать загаданное число.

```
int a = 5;
int b;

do {
    cout << "Guess the number: ";
    cin >> b;
} while (a != b);
```

```
cout << "You are right!";
```

Цикл for

Цикл for используется для перебора набора значений. В качестве набора значений можно использовать некоторые типы контейнеров:

vector

```
vector<int> a = {1, 4, 6, 8, 10};

int sum = 0;
for (auto i : a) {
   sum += i;
} cout << sum;
```

map

```
map<string, int> b = {{"a", 1}, {"b", 2}, {"c", 3}};

int sum = 0;
string concat;
for (auto i : b) {
   concat += i.first;
    sum += i.second;
}

cout << concat << endl;

 cout << sum;
```

string

```
string a = "asdfasdfasdf";

int i = 0; for (auto c : a) {
   if (c == 'a') {
      cout << i << endl;
   }
   ++i;
}
```

Простой цикл for позволяет создавать цикл с индексом:

```
string a = "asdfasdfasdf";

for (int i = 0; i < a.size(); ++i) {
   if (a[i] == 'a') {
      cout << i << endl;
```

```
    }
}
```

С помощью оператора **break** можно прервать выполнение цикла:

```
string a = "sdfasdfasdf";

for (int i = 0; i < a.size(); ++i) {
    if (a[i] == 'a') {
        cout << i << endl; break;
    }
} cout << "Yes";
```

```
3
Yes
```