На прошлой лекции

Три понятия: время жизни, область видимости и область действия - очень тесно связаны друг с другом.

Время жизни - обсудили (но вернемся к нему, когда будем обсуждать область действия)

Область видимости (scope) - подробно обсудили на прошлой лекции.

Теперь закончим обсуждение - понятие области действия.

Основные понятия ЯП - переменные

Область действия объектов (extent) - область программы, в которой объект данных (не обязательно именованный!!!) существует и доступен.

Понятие тесно связано с понятием времени жизни.

ОД - определяется в терминах областей программы, ВЖ - в терминах событий, происходящих при выполнении программы.

ОД объектов из динамической памяти - ДП (они, как правило, неименованные) - в общем случае - вся программа (статически не определимо).

А что с именованными объектами?

Основные понятия ЯП - область действия

ОД объектов из ДП (они, как правило, неименованные) - в общем случае - вся программа (статически не определимо).

А что с именованными объектами?

"Классические" императивные языки (С, Паскаль, Ада) - для именованных объектов область действия СОВПАДАЕТ с областью видимости (и, следовательно, время жизни определяется классом памяти).

Основные понятия ЯП - область действия

ОД объектов из ДП (они, как правило, неименованные) - в общем случае - вся программа (статически не определимо).

А что с именованными объектами?

"Классические" императивные языки (C, Паскаль, Ада) - для именованных объектов область действия СОВПАДАЕТ с областью видимости (и, следовательно, время жизни определяется классом памяти)

C - "некромантия" - за счет того, что адрес доступен не только для динамических объектов, но и для любых.

```
char * bar() {
   return (char*)calloc(1,1);
}
char * foo() {
  static char * s = "OK!!!";
  return s;
}
```

Основные понятия ЯП - область действия

С - "некромантия" - за счет того, что адрес доступен не только для динамических объектов, но и для любых.

```
char * bar() {
   return (char*)calloc(1,1);
}
char * foo() {
  static char * s = "OK!!!";
  return s;
}
char * necro () {
 char s[] = "OK!!!";
 return s;
}
char * s = necro(); // висячая ссылка!!! - ссылка есть, а объекта уже нет
```

Основные понятия ЯП - область действия

Область действия объектов (extent)

В современных ОИЯП появилось понятие "замыкания" (closure) из ФП

Иногда переводят как "захват"

Пример 1: C#

```
class Program
{
    static Func<int, int> PlusN(int N)
    {
        return x => x + N; // захват параметра N
    }

    static void Main(string[] args)
    {
        var plus1 = PlusN(1);
        var plus10 = PlusN(10);
        System.Console.WriteLine(plus10(5));
        System.Console.WriteLine(plus1(5));
    }
}
```

Основные понятия ЯП - область действия

Область действия объектов (extent)

Пример 2: C#

```
class Program
{
    static Func<int> Generate(int seed, int increment)
    {
        int next = seed;
        return () => next += increment; // захват параметра increment и локальной переменной next
    }

    static void Main(string[] args)
    {
        var g = Generate(1, 10);
        System.Console.WriteLine(g());   System.Console.WriteLine(g());   System.Console.WriteLine(g());

        var gg = Generate(5, -5);
        System.Console.WriteLine(gg());  System.Console.WriteLine(gg()); System.Console.WriteLine(gg());
    }
}
```

Основные понятия ЯП - переменные

Область действия объектов (extent)

Пример 3: C++

```
#include <functional>
#include <iostream>
std::function <int (int)> PlusN (int N) {
    return [N](int x) { return x+N; } // явный захват параметра N
}
int main() {
    auto plus1 = PlusN(1);
   auto plus10 = PlusN(10);
   std::cout << plus10(5) << std::endl << plus1(5) << std::endl;
   return 0;
}
```

Основные понятия ЯП - область действия

Область действия объектов (extent)

Пример 4: C++ ("некромантия")

```
std::function <int ()> Generate (int seed, int increment) {
  int next = seed;
  return [&next, increment]() { return next += increment; }
            // явный захват next по ссылке и параметра increment
}
int main () {
   auto gg = Generate(1,10);
   cout << gg() << endl << gg() << endl << gg() << endl;
   auto ggg = Generate(-5,5);
   cout << ggg() << endl << ggg() << endl << ggg() << endl;
   return 0;
}
```

Основные позиции при рассмотрении ЯП

- Технологическая
- Авторская
- Реализаторская
- Семиотическая
- Социальная

Схема рассмотрения ЯП

- Базис
  - скалярный
  - структурный
- Средства развития (создание новых абстракций)
- Средства защиты (поддержка новых абстракций)

Скалярные базисные типы данных в ОИЯП

Классификация скалярных (примитивных, простых) типов данных

- Арифметические
  - целочисленные
  - вещественные
- Логические
- Символьные
- Порядковые
  - перечисления
  - диапазоны
- Указатели/ссылки

Скалярные базисные типы данных в ОИЯП

Классификация скалярных (примитивных, простых) типов данных

- Арифметические
  - целочисленные
  - вещественные
- Логические
- Символьные
- Порядковые
  - перечисления
  - диапазоны
- Указатели/ссылки

Замечание 1: функциональные (подпрограммные) ТД - при рассмотрении средств развития

Замечание 2: вспомним известные нам ОИЯП (Паскаль, C, C++, ..., Java, C#, Python,...)

Скалярные базисные типы данных в ОИЯП

Классификация скалярных (примитивных, простых) типов данных

- Арифметические
  - целочисленные
  - вещественные
- Логические
- Символьные
- Порядковые
  - перечисления
  - диапазоны
- Указатели/ссылки

Замечание 1: функциональные (подпрограммные) ТД - при рассмотрении средств развития

Замечание 2: вспомним известные нам ОИЯП (Паскаль, C, C++, ..., Java, C#, Python,...)

Что новенького с 1969 года?

Принципиально - ничего, только детали....

Но эти детали - базисные => важные

Скалярные базисные типы данных в ОИЯП

Классификация скалярных (примитивных, простых) типов данных

- Арифметические
  - целочисленные
  - вещественные
- Логические
- Символьные
- Порядковые
  - перечисления
  - диапазоны
- Указатели/ссылки

Какие интересные типы отсутствуют?

Скалярные базисные типы данных в ОИЯП

Классификация скалярных (примитивных, простых) типов данных

- Арифметические
  - целочисленные
  - вещественные
- Логические
- Символьные
- Порядковые
  - перечисления
  - диапазоны
- Указатели/ссылки

Какие интересные типы отсутствуют?

Дата-время

Денежные (особые правила - нельзя свести к вещественным)

???

Скалярные базисные типы данных в ОИЯП

Классификация скалярных (примитивных, простых) типов данных

- Арифметические
  - целочисленные
  - вещественные
- Логические
- Символьные
- Порядковые
  - перечисления
  - диапазоны
- Указатели/ссылки

Какие интересные типы отсутствуют?

Дата-время

Денежные (особые правила - нельзя свести к вещественным)

Некоторые языки (Visual Basic) имеют эти типы в базисе (чем это плохо?), но общая концепция - все специфические типы (special) - в библиотеки (стандартные, либо 3-rd party)

Арифметические типы данных - целые

Почему 2 вида? - разные представления

JavaScript - единый ТД - Number. Фактически - 8-байтный double.

Просто для начинающих и непрограммистов.

Проблема - "7.0000000002 землекопа"

Некоторые ЯП - только integer. Отсутствие полноты - не страшно для "научных" и "игровых".

Арифметические типы данных - целые

Вопросы, связанные с целочисленными типами

- полнота (насколько полно учтена номенклатура машинных типов);
- наличие (или отсутствие) беззнаковых типов;
- надежность (какие ошибки могут возникать при выполнении операций с целыми значениями, как их предотвратить);
- отображение в машинные типы (размер значения, диапазоны значений);
- набор операций.

Арифметические типы данных - целые

Полнота - насколько полно учтена номенклатура машинных типов.

Два аспекта:

- нужен ли беззнаковый тип?
- номенклатура размеров

Минимальный вариант - только один целый тип - научные языки. Какой именно вариант целого типа выбрать?

"Родной" целый тип - Паскаль, универсальный "безразмерный" тип с эмуляцией длинной арифметики - Python, Lisp, Mathematica.....

Современные индустриальные ЯП - из соображений эффективности:

полная номенклатура размеров типов: 1-2-4-8 байтов (а 16 и выше?)

Арифметические типы данных - целые

Беззнаковые типы - частный и очень важный случай проблемы полноты

Зачем они вообще? Две основные причины:

- "бедность" (короткое маш.слово) (seek - lseek) - сейчас малоактуально
- поддерживаются ЛЮБОЙ машинной архитектурой (почему?) => для полноты

Арифметические типы данных - целые

Беззнаковые типы - частный и очень важный случай проблемы полноты

Зачем они вообще? Две основные причины:

- "бедность" (короткое маш.слово) (seek - lseek) - сейчас малоактуально
- поддерживаются ЛЮБОЙ машинной архитектурой (почему?) => для полноты

Почему это вообще вопрос? Добавить и все....

Надежность - проблема смешения знаковых и беззнаковых чисел.

Оберон и Java - нет беззнакового типа данных - нет этой проблемы

Но: C,C++,C#, Go, Swift,

Арифметические типы данных - целые

Надежность - какие ошибки могут возникать при выполнении операций с целыми значениями, как их предотвратить.

- переполнение
- смешение знаковой и беззнаковой арифметики

Арифметические типы данных - целые

Надежность - какие ошибки могут возникать при выполнении операций с целыми значениями, как их предотвратить.

- переполнение

int cnt = 0;

for (unsigned i = 255; i >= 0; i--) cnt++;

Как контролировать? Маш. архитектуры дают возможность (CF-OF в x86), но не поддерживают аппаратные прерывания по этому поводу (как с делением на 0).

Эффективность vs надежность

C - эффективность

Ряд других - на выбор программиста:

C#: checked-unchecked блоки

```
checked {
   for (uint i = 256; i >= 0; i--) cnt++;
}
```

Арифметические типы данных - целые

Надежность - какие ошибки могут возникать при выполнении операций с целыми значениями, как их предотвратить.

- переполнение

int cnt = 0;

for (unsigned i = 255; i >= 0; i--) cnt++;

Как контролировать? Маш. архитектуры дают возможность (CF-OF в x86), но не поддерживают аппаратные прерывания по этому поводу (как с делением на 0) - так как это не всегда ошибка (hash - вычисления по модулю....)

Эффективность vs надежность

C - эффективность

Ряд других - на выбор программиста:

C#: checked-unchecked блоки

```
checked {
   for (uint i = 256; i >= 0; i--) cnt++;
}
```

Rust - debug - контроль, release - отключение (нет контроля - в C всегда так)

Арифметические типы данных - целые

Надежность - какие ошибки могут возникать при выполнении операций с целыми значениями, как их предотвратить.

Смешение знаковой и беззнаковой арифметики:

C# (требует явных преобразований - под ответственность программиста):

short i = -1;

ushort ui = (ushort)i; // 65535 - для C/C++ по стандарту - неопределено

int k = 256;

sbyte sb = (sbyte)k; // 0 - то же самое

В C/C++ - даже преобразовывать не надо - само сломается.

Как избежать? Либо полностью исключить (Java), либо запретить сужающие и небезопасные преобразования (C#, Go, Rust).

C и C++ - ЛЮБЫЕ преобразования между арифметическими типами допустимы (могут быть предупреждения).

Насколько это опасно?

Современные ЯП - запрещают небезопасные преобразования (знаковые <-> беззнаковые одного размера, сужающие преобразования). Безопасные - расширения (integer promotions).

Арифметические типы данных - целые

Отображение в машинные типы - размер значения, диапазоны значений (связано с представлением, в т.ч. отрицательных чисел).

Не путать с полнотой. Полнота - возможность представления любого машинного типа в языке. Зачем? - эффективность использования компьютера. Если неважно => эмуляция (Питон - см. выше).

"Сцилла и Харибда" - с одной стороны, максимально использовать машинные ТД, с другой стороны, не зависеть от представления в конкретных архитектурах.

Как именно?

Арифметические типы данных - целые

Отображение в машинные типы - размер значения, диапазоны значений (связано с представлением, в т.ч. отрицательных чисел).

Не путать с полнотой. Полнота - возможность представления любого машинного типа в языке. Зачем? - эффективность использования компьютера. Если неважно => эмуляция (Питон - см. выше).

"Сцилла и Харибда" - с одной стороны, максимально использовать машинные ТД, с другой стороны, не зависеть от представления в конкретных архитектурах.

Как именно?

Пример 1. Язык Ада - весьма абстрактная схема => полностью машинно-независимая, но не ограничивающая с точки зрения как эффективности, так и надежности (см. выше).

Целые типы в языке Ада

Основная идея (весьма общая - не только для целых типов) - есть понятие наследования нового типа из существующего, и есть понятие типа - подтипа. Новый тип наследует полностью множество операций, и ,возможно ограниченное, множество значений. Подтип - наследует все то же самое, но не является отдельным типом.

Разные типы полностью НЕСОВМЕСТИМЫ друг с другом (не присваиваются, не смешиваются в выражениях), подтипы совместимы со своим главным типом, разные подтипы одного типа совместимы между собой (возможно, с контролем диапазона значений).

Есть конкретный стандартный (предопределенный) тип Integer, диапазон значений которого определяется реализацией:

Integer'First .. Integer'Last.

Также есть два стандартных ПОДТИПА:

subtype Natural is Integer range 0.. Integer'Last;

subtype Positive is Integer range 1.. Integer'Last;

Пример ограниченных подтипов. При присваивании значения переменной ограниченного (под)типа и при вычислениях проверяется, что значение входит в диапазон типа (подтипа) значения контролируется.

Стандартное исключение - Constraint_Error

Целые типы в языке Ада

Кроме обычных целых есть и т.н. модульные типы:

type T is Integer mod 2**16;

Все операции выполняются по модулю => нет нужды проверять отдельно. При подборе значения модуля, отвечающего машинным беззнаковым типам, все операции реализуются максимально эффективно.

К целочисленному набору операций добавлены побитовые операции and, or, xor, not.

Целые типы в языке Ада

Примеры (из стандарта LRM 2012 http://www.ada-auth.org/standards/12rm/RM-Final.pdf):

```
type Page_Num  is range 1 .. 2_000;
type Line_Size is range 1 .. Max_Line_Size;
subtype Small_Int   is Integer   range -10 .. 10;
subtype Column_Ptr  is Line_Size range 1 .. 10;
subtype Buffer_Size is Integer   range 0 .. Max;
type Byte        is mod 256; -- an unsigned byte
type Hash_Index  is mod 97;  -- modulus is prime
```

Целые типы в языке Ада

Итак:

- беззнаковые типы ОК
- с надежностью - тоже ОК
- полнота?

Компилятор должен подбирать "наилучшие" машинные типы.

Есть требования к минимальной реализации:

мин. диапазон Integer: -2**15+1 .. 2**15-1 (уже ограничение на архитектуру)

Для 32-битных машин - должен быть стандартный тип Long_Integer с мин. диапазоном -2**31 + 1 .. 2 ** 31 - 1

Все ограничения и требования сформулированы так, чтобы допускать различные представления отрицательных чисел (с явным знаком, с дополнением до 1 и 2)

В целом - эффективно, но сложновато, особенно для компилятора.

Арифметические типы данных - целые

Отображение в машинные типы - размер значения, диапазоны значений (связано с представлением, в т.ч. отрицательных чисел).

"Сцилла и Харибда" - с одной стороны, максимально использовать машинные ТД, с другой стороны, не зависеть от представления в конкретных архитектурах.

Как именно?

Пример 2: язык С.

Стандарт - задает полную номенклатуру типов данных и операций над ними + минимальные требования к реализации, включая некоторые зависящие от реализации ограничения (limits.h)

Что не задается? Не специфицируются отображения в конкретные типы и размеры, за одним исключением - unsigned char - это беззнаковый байт (зато не сказано, что же такое байт....!!!)

Стандарт требует, чтобы sizeof (unsigned char) = 1 и максимальное значение unsigned char = 2**CHAR_BIT-1 (это константа из limits.h должна бы называться UCHAR_BIT, но так уж сложилось...)

Арифметические типы данных - целые

Отображение в машинные типы - размер значения, диапазоны значений (связано с представлением, в т.ч. отрицательных чисел).

"Сцилла и Харибда" - с одной стороны, максимально использовать машинные ТД, с другой стороны, не зависеть от представления в конкретных архитектурах.

Как именно?

Пример 2: язык С.

Стандарт - задает полную номенклатуру типов данных и операций над ними + минимальные требования к реализации, включая некоторые зависящие от реализации ограничения (limits.h)

Что не задается? Не специфицируются отображения в конкретные типы и размеры, за одним исключением - unsigned char - это беззнаковый байт (зато не сказано, что же такое байт....!!!)

А что такое байт? Может ли он быть равным не восьми битам, другими словами есть ли архитектуры, в которых CHAR_BIT != 8? Да, есть!

Целые типы в языке C (C++)

Главный целый тип - int.

Компилятор должен отображать его В ТОЧНОСТИ в "родной" целый тип для каждой реализации (для гарантии того, что операции над int - максимально эффективны).

Про все остальное - только минимальные требования:

char - signed | unsigned - зависит от реализации (хотя любая реализация должна допускать настройку умолчания)

Пример: архитектура x86 - CBW и CWD операции - расширение знакового бита AL(AX) на AH(DX). В IA-32 добавились общие команды для этой операции (integer promotion для знаковых и беззнаковых чисел):

MOVSX r16/32, r/m8 или r/m16 - src копируется в dst с распространением знака

MOVSX r16/32, r/m8 или r/m16 - src копируется в dst с заполнением нулями

Минимально допустимый диапазон значений (а заодно номенклатура типов):

signed char: -127..127 unsigned char: 0..2**8 - 1

signed short: -2**15+1..2**15-1 unsigned short: 0..2**16-1

signed int: как с short unsigned int: как с unsigned short

signed long: -2**31+1..2**31-1 unsigned long: 0.. 2**32−1

signed long long: -2**63+1..2**63-1 unsigned long long: 0..2**64−1

Целые типы в языке C (C++)

Размеры:

sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long) <= sizeof (long long)

Каждый из знаков - "выстрадан" - ослабить нельзя (есть реализации практически для любой комбинации неравенств, вроде бы даже для sizeof (char) == sizeof (long) == 4).

Программист ОБЯЗАН знать и понимать детали отображения стандартных типов данных на машинные типы для используемой реализации (не умеешь - не берись).

Если важна точность отображения, то в помощь программисту (C99) добавлен заголовок stdint.h с объявлениями, фиксирующими точное отображение типов для каждой реализации (дополнение к limits.h)

Целые типы в языке C (C++)

stdint.h (в C++ - конечно cstdint) - набор typedef и констант

intmax_t	uintmax_t	Максимальный целый тип
int8_t	uint8_t	Типы, содержащие точное количество битов (8,16,32,64)
int16_t	uint16_t	Знаковые - с дополнением до 2
int32_t	uint32_t	Нет лишних битов
int64_t	uint64_t	Могут отсутствовать (если архитектура не поддерживает)
Int_least8_t	uint_least8_t	Типы, содержащие по меньшей мере нужное количество битов (8,16,32,64)
int_least16_t	uint_least16_t	Могут содержать лишние биты
int_least32_t	uint_least32_t	Типов меньшего размера с нужным количеством не может быть
int_least64_t	uint_least64_t	
Int_fast8_t	uint_fast8_t	Типы, содержащие по меньшей мере нужное количество битов (8,16,32,64)
int_fast16_t	uint_fast16_t	Но при этом самые быстрые для нужного размера
int_fast32_t	uint_fast32_t	
int_fast64_t	uint_fast64_t	

Целые типы в языке C (C++)

stdint.h (в C++ - конечно cstdint) - набор typedef и констант

Отдельно отметим целый тип, в который можно преобразовать (туда и обратно) без потерь void * (не всегда определен).

intptr_t и uintptr_t

Еще - набор предельных значений:

INTMAX_MIN, INTMAX_MAX, INT8_MIN, INT8_MAX, ... (для каждой константы есть минимальные требования, например, для INT8_MIN это точное значение -128== -2**7 (не -127!!!), а для INT_LEAST8_MIN - это -127 или меньше (<= -2**7 + 1), для

INT_LEAST8_MAX - это 127 или больше (>= 2**7 - 1)

Целые типы в языке C (C++)

Таким образом - достигнуты как полнота, так и точное отображение в машинные типы данных.

Бескопромиссное стремление к эффективности.

Надежность? - для профи не проблема...(?)

Арифметические типы данных - целые

Современные индустриальные ЯП - подмножество подхода C++ фиксированные типы по размерам + синонимы для "родных" типов.

Крайний случай - Java

Пример 3 - Java - все полностью зафиксировано - только знаковые типы, дополнение до 2, размеры - 1,2,4,8 (в 8-битных байтах).

byte(8), short(16), int(32), long(64)

Добавим сюда еще char и bool и получим полный список простых типов данных для Java

Арифметические типы данных - целые

Пример 4: C# - тоже все фиксировано (фактически ориентировано на IA-32)

Имя знак. типа	Имя б/з типа	Имена типов-оберток
sbyte	byte	SByte Byte
short	ushort	Int16 UInt16
int	uint	Int32 UInt32
long	ulong	Int64 UInt64

Длина типа-обертки - длина основного типа, знаковые типы - с дополнением до 2 (=> диапазон sbyte: -128..127, byte: 0..255 и т.д.)
Типы-обертки - специальные базисные классы в ООЯП - для эффективности ("все есть объекты", а как же эффективность операций над простыми типами?)

Арифметические типы данных - целые

Пример 5: Go - фиксированные типы по размерам и представлению + специальные обозначения для "родных" особых типов - как в C, но проще и менее универсально (не рассчитан на кофеварки...)

Имя типа (б/з типа)	Примечание
int8 (uint8)	
int16 (uint16)	
int32 (uint32)	
int64 (uint64)	
int	Синоним для "родного" целого (>=32 бита)
uint	Синоним для "родного" б/з целого (>=32 бита)
byte	Синоним для uint8
rune	Для символов (не байт!!!) - синоним uint32
uintptr	Синоним для б/з типа, совместимого с указателем

Модель данных языка С и основные архитектуры компьютеров

Модель (целочисленных) данных - соотношения между размерами основных типов данных в реализации языка С. Определяются архитектурой (если 32 бита, то int не будет 16-битным) и платформой (в наст. время - это ОС).

Обозначения - I - int, L - long, P - pointer, число - размер в битах.

Не говорим про "экзотические" архитектуры (типа некоторых контроллеров TI), а про самые популярные архитектуры - все имеют 8-битный байт - мин. адресуемая ячейка памяти. Поэтому sizeof(char) == sizeof (unsigned char) == 1; CHAR_BIT ==8;

sizeof (short) == 16

Также подразумеваем, что sizeof (int) либо равен sizeof(long), либо меньше в 2 раза.

Модель данных языка С и основные архитектуры компьютеров

Обозначения - I - int, L - long, P - pointer, число - размер в битах

Обозначение	int в битах	long в битах	pointer в битах	Платформы
IP16	16	32	16	PDP/11 - UNIX
LP32	16	32	32	i8086(MS DOS-Win3.X)
ILP32	32	32	32	IA-32 - все ОС
ILP64	64	64	64	IA-64 (ОС - уже неважно)
LLP64	32	32	64	x86-64 ОС Windows
LP64	32	64	64	x86-64 ОС Linux (Open Group)

Арифметические типы данных - целые

Набор операций

"знаешь С - знаешь все"

Исключение - возведение в степень (x**y) - есть в Фортране, Питоне ...., но не в С (как и в куче других языков)

(2 вывода)

1. Арифметика:

+ - * / % (бинарные) + - (унарные)

Вопрос - деление - нацело (/), или специальная операция (div)?

Побитовые операции (логические + сдвиги):

& | ~ ^ (and, or, not , xor), <<, >>

Вопрос - сдвиги арифметические (знаковые) или логические (б/з)?

Если нет б/з типов => нужны беззнаковые сдвиги (зачем?)

Арифметические типы данных - целые

Набор операций

"знаешь С - знаешь все"

1. Арифметика:

+ - * / % (бинарные) + - (унарные)

Вопрос - деление - нацело (/), или специальная операция (div)?

2. Побитовые операции (логические + сдвиги):

& | ~ ^ (and, or, not , xor), <<, >>

Вопрос - сдвиги арифметические (знаковые) или логические (б/з)?

Если нет б/з типов => нужны беззнаковые сдвиги (зачем?)

Java: в дополнение к операции >> есть и операция >>> - логический сдвиг целого значения со знаком.

Почему нет <<< в дополнение к <<?

Арифметические типы данных - целые

Набор операций

1. Арифметика:

+ - * / % (бинарные) + - (унарные)

2. Побитовые операции (логические + сдвиги):

& | ~ ^ (and, or, not , xor), <<, >>

Вопрос: к каким типам применимы?

Индивидуально для ЯП:

Ада - б/з

C# - int, uint, long, ulong (для остальных - расширение до int(uint))

Python - да без проблем (единственный целый тип!)

JavaScript - ? (единственный тип Number, но он - аналог double в C)

Арифметические типы данных - целые

Набор операций

1. Арифметика:

+ - * / % (бинарные) + - (унарные)

2. Побитовые операции (логические + сдвиги):

& | ~ ^ (and, or, not , xor), <<, >>

Вопрос: к каким типам применимы?

Индивидуально для ЯП:

Ада - б/з

C# - int, uint, long, ulong (для остальных - расширение до int(uint))

Python - да без проблем (единственный целый тип!)

JavaScript - ? (единственный тип Number, но он - аналог double в C)

JS - операнды побитовых операций преобразуются в 32-битные б/з целые, при этом все "ненужное"(дробная часть, старшие разряды) отбрасывается.

Далее - "знаешь C - знаешь все"

И еще вопрос - а каких побитовых операций нет (даже в C)?

Арифметические типы данных - целые

Набор операций

1. Арифметика:

+ - * / % (бинарные) + - (унарные)

2. Побитовые операции (логические + сдвиги):

& | ~ ^ (and, or, not , xor), <<, >>

3. Операции сравнения:

<, >, <=, >=, == (=), != (<>, /=, #, ...)

Почему == ? Потому что = в C - это операция присваивания.

А есть еще и === (JS, Ruby), но ее семантика к целым не относится...

Арифметические типы данных - целые

Набор операций

1. Арифметика:

+ - * / % (бинарные) + - (унарные)

2. Побитовые операции (логические + сдвиги):

& | ~ ^ (and, or, not , xor), <<, >>

3. Операции сравнения:

<, >, <=, >=, == (=), != (<>, /=, #, ...)

4. Операции с побочным эффектом

Одна из самых противоречивых "фишек" языка С

Арифметические типы данных - целые

Операции с побочным эффектом

- операция присваивания (a = b != c << d = ++c = --d)
- составные присваивания (a += b)
- инкремент, декремент (i++, ++i, i--, --i)

Идея - из Алгола 68.

Зачем в C(на самом деле - в B)?

Одна из здравых причин - "подсказка" компилятору (в 1969 - актуально, сейчас - абсолютно нет - не надо подсказывать компилятору про микрооптимизацию)

Вторая - "компактность" и "выразительность" программ.

Сейчас - абсолютно неактуально... (что делает программист?)

Арифметические типы данных - целые

Пример 1 из C(C++)

```
int MyStrlen(const char * s) {
    const char * beg = s;
    while (*s++);
    return s - beg;
}
```

Пример 2 из C(C++)

```
char * MyStrcat(char * dst, const char * src) {
    char * beg = dst;
    while (*dst++ = *src++);
    return beg;
}
```

Арифметические типы данных - целые

Пример 3 из C(C++)

```
int MyStrcmp(const char * s1, const char * s2) {
  while (*s1++ == *s2++ && *s1);
  return *s1 - *s2;
}
```

А можно было легко не допустить ошибку (не цепляясь за "компактность" и "выразительность"):

```
int MyStrcmp(const char * s1, const char * s2) {
  while (*s1 == *s2 && *s1) { s1++; s2++;}
  return *s1 - *s2;
}
```

Современные компиляторы выдадут не менее эффективный и компактный код, чем вначале, зато он будет правильный!

Арифметические типы данных - целые

C#, Java - "рабски" наследовали побочные эффекты от C.

Python - игнорирует, равно как и более современные языки (Go, Swift)

x += y

Не выражение, а оператор!

++ (нет в Python, в Swift - исключен из последних версий)

i++; Go - оператор (++i вообще нет)

Внимание - курьез (или ошибка!):

++i в Питоне или Go?

В первом случае ОК (и что значит?), во втором - ошибка (сознательное ограничение синтаксиса)

Арифметические типы данных - целые

Преобразования.

Современные ЯП - только расширяющие и только в ЯП, где определены представления (диапазоны значений).

Общая идея расширяющих преобразований (integer promotions):

- от меньшего размера - к большему без смены знака
- от меньшего размера б/з - к большему со знаком

Остальные преобразования (сужающие) - запрещены (только явно!).

Исключение - C/C++ (исторически).

Арифметические типы данных - целые

Общий вывод - отличия невелики - в деталях. Но "дьявол - в деталях".