**Простые типы данных**

Под простыми типами данных мы имеем в виду языковые конструкции, встроенные в язык, то есть типы

данных изначально поддерживаемые в языке.

Классификация простых типов данных

- числовые

- целочисленные

- вещественные

- плавающие

- фиксированные

Не во всех языках есть такое усложнение класса чисел, например, в таких скриптовых языках как

JavaScript и ActionScript есть тип Number, обьекты класса Number могут принимать как

целочисленные значения, так и вещественные.

- логические

- символьные

-порядковые

-перечисления

-диапазоны

-ссылки и указатели

И еще иногда вводят функциональные тип(функции и процедуры)

**Целочисленные**

Перед разработчиком языка при создание целочисленного типа всегда возникают как минимум три

проблемы:

1. Фиксация представления – то есть фиксировать ли размеры типов данных, и набор

значений.

2. Беззнаковые целые числа – нужно ли выделять беззнаковый целый тип

3. Преобразования одного целочисленного типа в другой

Наиболее известные языки программирования с фиксацией представления типов данных – Java, C#.

Другое дело с языком С. Именно в языке С

решили ввести диапазон чисел, который должен быть покрывать диапазоны, вмещаемые в 1 - 8 байт:

-char

-short int

- int

- long int

- long long

Правда здесь отдельно ничего не ясно про каждый тип(кроме char – 1 байт), например short int на различных

платформах может занимать как 1 байт, так и 2 байта. Но зато ясно, что количество байт отводимых под char,

меньше либо равно количеству байт, отводимых под short int, количество байт отводимых под short int,

меньше либо равно количеству байт, отводимых под int, и т.п. Такая системы была унаследована в языка С++.

Рассмотрим беззнаковые целые числа.

Так в Модуле-2 существовали два типа INTEGER и CARDINAL. И приведение одного в другой

допускалось только явное(существовал специальный оператор для такого присваивания):

I:INTEGER; J:CARDINAL; I := INTEGER(J);

Но в современном мире языков программирования многие языки такими ограничениями не обладают.

Например, язык С++.

Так, к примеру в Обероне существует лишь один беззнаковый тип

byte(0..255), и Java, позаимствовавшая типы данных именно с Оберона была вынуждена включить

дополнительно включить в базовые арифметические операции беззнаковый сдвиг(>>>). То есть базисный

набор операции над целыми числами расширен по сравнению с базисным набором операций в С.

Существовали и другие решения данной проблемы. Так в С# решили допускать только расширяющиеся

преобразования(так как они безопасные). Обратные преобразования допускались только явно. Про

приведение типов в АДА смотреть ниже.

Подходы к реализации числовых типов данных.

1. Фиксирование базиса. То есть мы заранее говорим, что вот такие типы данных, как integer,byte …

являются целочисленными. Все остальные уже не целочисленные. В некоторым языках фиксируется

даже размер каждой переменной данного типа, то есть integer в Java занимает, например, строго 2

байта. Как правило код на таких языках интерпретируется. В Java, например, виртуальная Java –

машина. Еще один пример такого языка С#. Программы на таких языках являются мобильными, то

есть переносимыми с одной платформы на другую. В других языках, переносимость затруднена, там

не фиксированы ни представление чисел, ни семантика числовых операций.

2. Обобщенные числовые типы. Ада создавалось отчасти и для того, чтобы код на ней был

кроссплатформенным. Поэтому фиксирование базиса в такой ситуации было затруднено. Перед

создателями Ады ставились следующие задачи: Эффективность, Надежность, Читабельность. В

следствие этого создатели Ады придумали новую концепцию. Они ввели обобщенные числовые типы

данных. То есть типы являющиеся базой для всех других типов( все другие наследовались от них).

Объекты разных типов были несовместимы ни по какому множеству операций, но были совместимы

объекты подтипов. То есть разные подтипы совместимы между собой и со своим предком. Например:

Type Length is new integer;

Type Width is new integer;

Length и Width – новые целочисленные типы данных, при этом их нельзя ни присваивать друг

другу, ни сравнивать. Однако можно делать преобразования явным. В С или Паскале мы бы таким

способом ввели бы понятие эквивалентности.

Еще один пример:

Type Length in new integer range 0..MAXN.

Думаю в комментариях не нуждается - диапазон Length ограничен 0..MAXN.

В чем плюс обобщенных типов, так то что все ошибки обнаруживались на этапе компиляции.

Если требовались неявные преобразования вводились под – типы.

Sybtype t1 is t2 range 0..N.

И тогда преобразования из T1 в T2 допускались. При этом компилятор сам выбирает

оптимальное представление для таких чисел. То есть представление таких типов данных зависело не

от архитектуры, а от диапазона чисел. Теперь компилятор транслирую присваивание таких типов

данных либо проверяет возможность такого присваивания на этапе трансляции, или же вставляет код

по проверке допустимости такого присваивания(квазистатический контроль). В случае ошибки

выхода за границу в Ада возбуждается range error.

Вещественные типы данных.

При представлении действительных чисел в компьютере, как и целых чисел, используется чаще всего

двоичная система счисления (иногда двоично-шестнадцатеричная), следовательно, предварительно

десятичное число должно быть переведено в двоичную систему, а затем представлено в нормализованном

формате с Р = 2 (для двоично-шестнадцатеричной системы Р = 16).

Независимо от используемых систем счисления существует два основных типа представления чисел в

компьютере, называемые представлениями с фиксированной и с плавающей запятой.

В типах данных, использующих представление чисел с фиксированной запятой, все разряды ячейки,

кроме знакового разряда, служат для изображения разрядов чисел. Причем каждому разряду ячейки

соответствует всегда один и тот же разряд числа, что и фиксирует место запятой перед определенным

разрядом. Такая система упрощает выполнение арифметических действий, но сильно ограничивает диапазон

чисел, которые могут быть представлены в таком типе. Чаще всего это бывает диапазон -1 < x < 1. Этот

случай соответствует соглашению: что при чтении чисел, записанных в машине, запятая ставится

непосредственно перед старшим цифровым разрядом.

Наибольшее по абсолютной величине число, которое может быть представлено в машине таким

образом, равно 1 – р-n, наименьшее – отличное от нуля – р-n. Для представления чисел, не укладывающихся в

этот диапазон, программисту надо вводить масштабные множители, т.е. заменять истинные величины,

участвующие в решении задачи, их произведениями на специально подобранные коэффициенты. Это

существенно усложняет решение поставленной задачи. Поэтому для представления вещественных чисел в

современных компьютерах формат данных с фиксированной запятой не применяется.

Описанные первой главе целые типы данных представляются в компьютере в формате с

фиксированной запятой. Положение запятой в них зафиксировано после правого разряда.

Чтобы избавиться от недостатков представления действительных чисел с фиксированной запятой, в

современных ЭВМ принят способ их представления с плавающей запятой. Этот способ представления

опирается на нормализованную (экспоненциальную) запись чисел:

+- M \* Bp ( знак/ мантисса/порядок(B- основание системы счисления).

Представление в виде мантиссы и порядка не единственно. Для арифметических операция используется

нормализованное представление, то есть 1/B < -M < 0. То есть в системе с основанием B все степени B

представимы точно.

Еще раз скажу, что при вещественных операциях точность страдает, так мантиссу можно выбрать так, что

погрешность будет порядка нескольких сотен(не мало да!)

Использование как в компьютере, так и в калькуляторе представления чисел с плавающей запятой

усложняет схему арифметического устройства.

При сложении и вычитании чисел сначала производится подготовительная операция, называемая

выравниванием порядков. Она состоит в том, что мантисса числа с меньшим порядком сдвигается в своей

ячейке вправо на количество разрядов, равное разности порядков данных чисел. После этой операции

одноименные разряды чисел оказываются расположенными в соответствующих (одних и тех же) разрядах

обеих ячеек, и теперь сложение или вычитание мантисс выполняется достаточно просто, так же как над

числами с фиксированной запятой.

При умножении двух чисел с плавающей запятой их порядки складываются, а мантиссы –

перемножаются (предварительное выравнивание не производится).

При делении из порядка делимого вычитается порядок делителя, а мантисса делимого делится на

мантиссу делителя.

После операций над порядками и мантиссами получается порядок и мантисса результата, но последняя

может не удовлетворять ограничениям, накладываемым на мантиссы нормализованных чисел. Поскольку от

результата арифметических операций в машине требуется, чтобы он также; был нормализованным числом,

необходимо дополнительное преобразование результата, называемое нормализацией. В зависимости от

величины получившейся мантиссы результата, она сдвигается вправо или влево так, чтобы ее первая значащая

цифра попала в первый разряд после запятой. Одновременно порядок результата увеличивается или

уменьшается на число, равное величине сдвига.

Когда говорят о точности представления вещественных чисел, надо помнить следующее: десятичное

число, имеющее даже всего одну значащую цифру после запятой, вообще говоря, невозможно записать точно

в любом из вещественных типов. Объясняется это тем, что конечные десятичные дроби часто оказываются

бесконечными периодическими двоичными дробями. Так, 0,110 = 0,0(0011)2, а, значит, и в нормализованном

виде такое двоичное число будет иметь бесконечную мантиссу и не может быть представлено точно. При

записи подобной мантиссы в ячейку компьютера число не усекается, а округляется.

Плавающие числа нужны для представления математических расчетов В других сферах часто требуется

несколько иное, например если мы работаем с рублями, то нам важна точность до второго порядка после

запятой(копейка). То есть каждое число по идее является целым, но по виду оно вещественное. И если есть

только плавающий тип, то при представлении таких чисел неудобно использовать операции с плавающей

точкой. Поэтому в некоторых языках ввели delta – типы.

Aда: type DATA is delta 1/4096 range –M..M

При этом вся дальнейшая работа по представлению ложится опять на компилятор(он сам лучше

подберет как хранить это число).

В других языках появляются еще разновидность delta – типов: С# - decimal, currency.

Логический тип данных.

Логические, или булевские, данные предназначены для хранения логических значений "истина"(1) или

"ложь"(0). В большинстве языков базис операций над логическим типом сводится к операциям: or, not, and

• И (логическое умножение) (AND, &, \*),

• ИЛИ (логическое сложение) (OR, |, +),

• Отрицание (NOT, ~, !)

Булевый тип данных может быть реализован с использованием только одного бита, но обычно

используется минимальная адресуемая ячейка памяти (байт) или машинное слово, как эффективная единица

работы с регистрами и оперативной памятью.

Aда: type Boolean is (False, True); // представление типа Boolean

p : Boolean := True;

...

if p then

...

end if;

C#, C, Java, Algol: bool p;

Кстати С единственный язык с неявным преобразование int в bool

Delphi, Pascal: var p:boolean

Символьный тип данных.

Символьный тип (Сhar) — простой тип данных, предназначенный для хранения одного символа в

определённой кодировке. Может являться как однобайтовым (для стандартной таблицы символов), так и

многобайтовым (к примеру, для Юникода).

**п.2.4 Ограниченные типы данных**

**п.2.4.1 Перечисления (перечислимые типы данных)**

Паскаль:

Type EnumTyp = (va1, …, valN);

Операции:

:=, =, <, >, <>, >=, <= - основаны на функциях succ(x), pred(x)

ord(x) = 0…N-1 (Элементы упорядочены)

Таким образом перечислимый тип – некий способ удобного создания, хранения и использования констант.

преобразования:

EnumType -> Integer безопасно

Integer -> EnumType Небезопасно. Нужны проверки

**Статические проверки** – проверки при компиляции программы. Проверяемые данные не меняются в различных вызовах функции (запусках программы)

A:array [1..100] of integer;

……….

a[10] = 10; Статическая проверка

**Квазистатические проверки** - проверки при выполнении. Квазистатическим они называются потому, что проверяемые данные могут меняться в различных вызовах

функции (запусках программы). Поэтому эти проверки делаются именно при выполнении.

Компилятор обнаруживает опасное место в программе и вставляет в это место квазистатический код, который выполнится в процессе работы программы.

a:array [1..100] of integer;

……….

a[n] = 10; n не известна в момент компиляции. Если язык поддерживает квазистатические проверки, то здесь будет вставлен контроль выхода за границу массива

x:0…N-1

x:= expr; Вставится роверка вида «if(expr >= N) or (expr <0) then error()», т.к. значение expr не известно во время компиляции

В машинных языках нет квазистатических проверок. Поэтому нет и в Си

int a[10];

n = 11;

a[n] = 10; В Си проверки не будет

RTTI – динамическая информация о типе. Ради этого в Си++ вставлены квазистатические проверки.

В Си-89 добавили такую конструкцию:

enum ET(v0,…,vN);

которая эквивалентна последовательности строк

#define v0 0;

#define v1 1;

……..

Но так как нет квазистатических проверок, то все нижеследующие строки будут корректны:

enum ET y;

int I;

x = v1;

I = v2;

x = I;

I = x;

x = -5;

Негласная парадигма языка Си – «компактность кода», а квазистатические проверки, естественно, увеличивают генерируемый код.

В своё время перечислимые типы были очень популярны.

Но:

* Языки «Оберон»(1988) и «Оберон-2»(1993) уже не содержали перечислимые типы. В Обероне было «Расширение типов», которому противоречат перечислимые типы данных, которые невозможно расширить.
* uses (pascal, Ада) Вместе с перечислимым типом неявно импортируются все его константы на том же уровне видимости .Таким образом значения констант могут конфликтовать и перекрываться для разных перечислимых типов (в т.ч. из разных модулей)
* Java (1995) – нет перечислимых типов
* C# (1999) – перечислимые типы есть. Одно из назначений - хранение наборов значений параметров компонентов( например влево/вправо/по центру для выравнивания). Т.е. перечислимые типы интегрировали в визуальные средства проектирования.
* Java (2005) - расширение Java, в том числе добавление перечислимых типов, оформленных в виде полноценных классов

Вопросы, связанные с реализацией и использованием перечислимых типов данных:

* Проблема представления – реализовывать ли возможность задавать конкретные значения констант (например, для цветов)?
* Проблема эффективности – реализовывать ли возможность управления представлением?
* Проблема преобразований в другие целочисленные типы данных – разрешать или не разрешать?
* Неявный импорт – разрешать или нет?
* Удобство использования – ввод, вывод и т.д.

|  |
| --- |
| **Модула-2**  type ET = (v1, …, vN);  Преобразования :  ord(x); ET -> integer  val(T,i); ET -> integer. Либо выдаёт ошибку, либо выдаёт значение.  val(ET,v2) = 1;  Неявный импорт возможен  Удобность использования не расширена. |
| **С++ <-> C-89**  enum ET{…};  void f(ET x);  void f(int x); Будет перегрузка, потому что перечисление – новый тип данных  В Си++ typedef задаёт синоним типа, а не создаёт новый тип.  typedef int a;    void f(a x)  void f(int x) Перегрузки не будет, т.к. «a» не новый тип(=> будет ошибка)  В Си++ можно задавать значения константам перечисления  Enum FileAccept  {  FileRead = 1;  FileWrite = 2;  FileReadWrite = FileRead | FileWrite;  }  Удобство использования такое же, как и для int |
| **Ада**  Типы BOOLEAN и CHARACTER являются перечислимыми типами данных из пакета STANDARD  **Проблема** неявного импорта:  type SendColor is (Red,Yellow,Green)  type BasicColor is (Red,Green,Blue)  Ввели понятие «литерал» перечисления. Им является либо идентификатор, либо символ(‘символ’)    type Latin is (‘A’, ‘B’,’C’,… ,’Z’)  type ASCII Latin(‘A’, ‘B’,’C’,… ,’Z’, ‘a’, ’b’,…)  **Литерал перечисления –** функция без параметров, имя которой совпадает с литералом и возвращает нужную константу  procedure P(x:SendColor)  procedure P(x:BasicColor)  P(Wellow) – в функцию отправится «1»  P(Red) - ?? ошибка  **Решение** проблемы неявного импорта:  **Уточнение типа** - **T’expr** – выражение **expr** трактовать как выражение типа **Т**  P(BasicColor’Red) – правильно!  Представление:  for BasicColor use (Red => FF00000x, Green => FF00x, Blue =>FFx)  for BasicColor’Size use 24; - 24- количество битов  «’» специальная операция, применимая именами типов, позволяющая получить доступ к некоторым атрибутам |
| **C#**  Управление реализацией:  enum BasicColor  {  Red = 0xFF0000;  Green = 0xFF00  Blue = 0xFF  };  enum SendColor: Byte  {  Red;  Yellow;  Green;  };  Преобразования из **перечислимых типов в целочисленные** в современных языках **только явные**.  Атрибуты – средства сообщать компилятору некоторую информацию о реализации типа.  [FLAGS] – указывает, что перечисление может обрабатываться как битовое поле, которое является набором флагов.  [FLAGS]  enum FileAcces  {  FileRead = 1;  FileWrite = 2;  FileReadWrite = FileRead | FileWrite;  }  теперь, если вывести на экран FileAccept. FileReadWrite получим «FileRead, FileWrite». Без использования атрибута [FLAGS] получим 3. |
| **С#, Java**  **Неявный импорт**  FileAcces x;  X = FileAcces.Read; //Уточнение перечисления  **Удобство использования. Классы-обёртки**  Классификация ТД   1. **Java**     * Простые Типы Данных    * Референциальные типы данных    * Классы    * Массивы 2. **C#**    * Типы-значения    * Классы – размещаются только в динамической памяти   Для каждого типа-значения существует класс-обёртка  **С#** int - Int32,bool – Boolean, long – Int64  **Java** int – Integer,bool – Boolean, long – Long  В С# все обёртки находятся в .Net  Для всех перечислений имеется один класс-обёртка – «Enum»   * всем перечислениям доступны методы класса Enum ToString(), GetValues() и т.д.   int [] vals = (int []) Enum.GetValues(typeof(…)); |
| **Java v5.0 Tiger 2005**    Enum SendColor  {  Red,Yellow,Green //Статические члены  }  SendColor c = SendColor.Red;  В Java v5.0 константы перечисления - статические члены класса, находящиеся в классе и представляющее собой значения класса. Упорядочены. Значения задавать нельзя.    SendColors.valuesof();  .ordinal();  .value(“Red”) // -> 0  enum Apples  {  ….  Apple4(10); /// Каждое значение перечислимого типа должно быть со своей ценой  ….  int price;  public Apples(int p)  {  price = p;  }  public void set\_price(int p){…}  public int get\_price(){….}  }  …..    Apples x = Apples.Apple4; // можно писать без «new». Это исключение для перечислений  // Apples.Apple4 – значение **класса**, а не **поля класса**  **Объекты перечислений нельзя копировать.**  **Наследование от перечислений запрещено.** |

**п2.4.2 Диапазоны**

|  |
| --- |
| **Паскаль**  var  x: L..R; |
| **Модула 2**  var  x: [0..N];  y: CARDINAL[0..N];  i:INTEGER;  j:CARDINAL;    x := 0; Проверки не будет  x := i; Вставка квазистатической проверки, если нельзя вычислить i  x := y; Проверки не будет.  Напомню, что переменные типа CARDINAL могут принимать значения от 0 до 65535, а INTEGER от -32768 до 32767 (для 16-ти битных компиляторов) |
| **Ада**  Type Diap is range 0..N;  Type Pos is new INTEGER range 0..MAX\_INT; новый тип данных.  ….  I:integer;  Y:Pos;  Y := I; компилятор вставит сообщение об ошибке, т.к. это разные типы данных (из-за «new»)  Y:=Pos(I); Вставка проверки на знак    Subtype Natural is range 1..MAX\_INT;  J:Natural  ………  I := J; Проверки не будет  J := I; В этой строчке будет проверка |

Ни в одном из современных языков программирования нет типов-диапазонов, т.к. современных языках чётко определены индексы массив: 0 <= i <= N-1, а основной областью применения диапазонов было именно задание типов индекса массива.

**п.2.5 Указатели и ссылки**

Адрес :

* + Указатель
  + Имя
  + Метка

**Указатели**

|  |  |
| --- | --- |
| **Строгие**  Стандартный Паскаль, Модула-2, Ада, Оберон (со сборкой)  Pascal  Type PT = ^T;  {Modula-2  Type PT = pointer to T;}  var  i :T;  Инициализировать указатель можно только двумя способами – либо другим указателем, либо выделением новой памяти NEW(p : PT);  Поэтому все данные чётко разделяются на именованные, либо не именованные.  Указатель служит для работы с анонимными данными в динамической памяти.  Смысл – избежание части ошибок | **Не строгие**  C, C++, Turbo Pascal  Можно получать адрес любого объекта с помощью операции взятия адреса «&»  Существует абстрактный указатель «void \*»  T \* => void \* автоматически  void \* к T \* автоматически не приводится  T \* p;  void \*pp;  pp = p;  p = (T \*)pp; |

Проблемы строгих и нестрогих указателей:

* Удаление памяти (Dispose(p : PT)) не в своё время, что приводит к появлению **«висячих ссылок»** - указателей, которые должны на что-то указывать, но не указывают.
* Накопление **мусора** – памяти, на которую не указывает ни один указатель.

P,P1 : PT;

New( P );

New( P2 );

P := P2; {порождение мусора}

Dispose(P2); {P – «висячая ссылка», попытка обращения к памяти, которую она занимает, приведёт к ошибке}

Различают системы с динамической сборкой мусора и без таковой.

Строгий язык с динамической сборкой мусора довольно надёжен.

От висячих ссылок защиты нет

**Ада**

В чистой Аде есть только new(p). В модуле STANDARD есть UNCHECKED\_DEALLOCATION(p) – подчёркивается небезопасность этой операции

Примеры ошибок:

T \*p;

Void f()

{

T x;

P = &x; //!!!адрес локальной переменной!!!

}

void Foo()

{

f();

free(p); //!!!попытка освобождения невыделенной памяти!!! – выдастся ошибка

}

new(p1);

p := p1;

Dispose(p1); {р «висит»}

Для Java, C# - указатели трансформировались в ссылки

**Операторы перехода**

goto

break;

continue; (в Модуле-2 EXIT)

return;

В современных языках программирования goto является только локальным

Для организации не локальных переходов:

setjmp, longjmp – В Си++ используются для обработки ошибок.

throw, trace ­­– Обработка исключений

Также существуют специальные операторы для организации параллелеризма

Lock(obj) {блок} – Си#. Поток управления блокируется, если блок кем-то используется

accept, select – Ада

**Базисы:**

Язык Ассемблера <––> Си <––> Си++ < ––> Java, C#

Языки программирования в первую очередь различаются за счёт средств развития и их защиты.

Каков минимальный набор средств развития?

**Процедуры** (подпрограммы, далее **– п/п**)

**FORTRAN**

**Модульность**

**Межмодульные связи:**

**По данным** – общие блоки

**По управлению** – вызов п/п

Этого уже достаточно для создания больших сложных программ, но без защиты новых абстракций

**Глава 5. Подпрограмма**

**п5.1 Потоки управления – подпрограммы и сопрограммы**

Управление входит через заголовок в блоке и возвращается в точку вызове, после выполнения тела.

**CALLER** – вызывающий подпрограмму(надпрограмма)

**CALLEE** – вызываемая подпрограмма

….

Call

….

…..

Call

…..

return

…...…...

….

return

**SUBROUTINE** – подпрограмма

**COROUTINE** – сопрограмма

//PP

….

….

call P

….

….

call P

//P

….

….

return

Неравенство

//PP

….

….

resume P

….

….

resume P

//P

….

….

resume PP

….

….

resume PP

Равенство

Также нарушение априорного порядка выполнения команд может происходить при генерации исключений.

Впервые механизм сопрограмм был придуман для компилятора COBOL. Вспомните задание по Си++ в 4-м семестре, где нужно было написать транслятор модельного языка:

Лексический анализатор, Синтаксический анализатор, Генератор кода – всё это сопрограммы.

**Сопрограммы –** фактически квазипараллельные процессы.

**Модула-2**

Вызов сопрограммы аналогичен длинному переходу на некоторый абстрактный адрес, по которому находится команда сопрограммы, с которой нужно начать выполнение. Но, помимо этого, нужно ещё как-то запомнить адрес возврата и другую служебную информацию, передать входные параметры, наследуется часть контекста. Для этой цели Вирт в своём языке Модула-2 ввёл тип данных **ADDRESS** (то же самое, что и «void \*»).

Этот тип данных является потенциальной дырой в системе безопасности, т.к. любой указатель «Т \*» автоматически приводится к «void \*», и возможно обратное явное преобразование «Т \*» = (Т \*)«void \*». Для Вирта было неприятной неожиданность то, что программисты часто использовали тип данных ADDRESS.

Строгость типизации зависит от возможностей преобразования.

**Типы** – непересекающиеся области эквивалентности, определяемые операциями на объектами этих областей.

Итак, в Модуле-2 вызов сопрограммы имеет такой вид:

NEWPROCESS(P, C,N);

Где P – процедура без параметров типа PROCEDURE, который является встроенным, C – переменная типа ADDRESS. N - размер области для «запоминания» информации. Область начинается с адреса C

PROCEDURE NEWPROCESS(P : PROCEDURE; VAR C : ADDRESS; N : INTEGER);

Передача управления от одного процесса другому на уровне сопpогpамм осуществляется процедурой "Передать управление от процесса P1 процессу P2". В Модуле-2 эта процедура выглядела как

PROCEDURE TRANSFER(VAR P1,P2 : ADDRESS);

При этом в переменную P1 записывается запись реактивации этого процесса, а значение переменной P2 определяет запись активации процесса P2.

RESUME; **–** оператор языка.

Маленькое замечание:

Изначально Вирт вместо **ADDRESS** использовал тип **COROUTINE**, теперь понятнее? Тип **COROUTINE** был похож не структуру**.**

В современных языках сопрограммы трансформировались в понятие потока.

.Net Thread Квазипараллельный поток

C# 2.0:

foreach(T x in C)

Тип T должен реализовывать интерфейс IEnumerable. Этот интерфейс содержит метод GetEnumerator(), который возвращает объект некоторого типа, который должен реализовывать интерфейс IEnumerator со свойствами Current,методом Reset и методом bool MoveNext(). Любой класс, поддерживающий интерфейс IEnumerable должен содержать класс, поддерживающий IEnumerator.

yield-операторы в C# 2.0:

yield return <expression>;

yield break;

Итератор – процесс(сопрограмма), выдающий последовательно очередные элементы коллекции тогда, когда они понадобятся. yield-оператор используется в блоке итератора для предоставления значения объекта перечислителя или для сообщения о конце итерации. Т.е. это не простой «return» или «break», а оператор, совмещающий в себе дополнительно работу по переходу между сопрограммами (от процесса-итератора в основной процесс). Выражение expression вычисляется и возвращается в виде значения объекту перечислителя; выражение expression должно неявно преобразовываться в тип результата итератора.

ublic class List

{

//using System.Collections;

public static IEnumerable Power(int number, int exponent)

{

int counter = 0;

int result = 1;

while (counter++ < exponent)

{

result = result \* number;

yield return result;

}

}

static void Main()

{

// Display powers of 2 up to the exponent 8:

foreach (int i in Power(2, 8))

{

Console.Write("{0} ", i);

}

}

}

/\* Output:

2 4 8 16 32 64 128 256 \*/

Генеральная линия развития C# - добавление элементов функционального программирования

**п.5.2 Потоки данных в подпрограммах**

* Через глобальные данные
* Через параметры

**Побочный эффект действия процедур и функций** – изменение значений глобальных переменных и данных, а так же модификация данных, глобальных по отношению к самой процедуре\функции.

**Глобальная переменная** – переменная, которая видна везде

Возможен конфликт связей

int a

P1

P2

P1

P4

В объектно-ориентированной парадигме:

объект

Члены-данные

//Методы

…

void f(){ … ;

i = …;

… ;}

Глобальное, по отношению к данному объекту , пространство имён

Сначала «i» ищется в теле функции, потом в членах-даннных, и только после этого вне объекта

**Виды формальных параметров(семантика)**:

* Входные (in) – должны быть определены до входа
* Выходные (out) – должны быть определены к моменту выхода
* Вх/Вых(InOut) – и то и другое

**Способы передачи**

**Способ передачи –** способ связывания фактических и формальных параметров:

* По значению (семантика - in)
* По результату (семантика – out)
* По значению/результату (семантика – InOut)
* По адресу(по ссылке) (семантика - любая)
* По имени

**Ада83:**

Квалификаторы: in, out, inout

Procedure P(int X:T;inout Y:T;out Z:T);

X может быть выражением типа T. Компилятор может вставлять квазистатические проверки. Эффект процедуры – модификация Y и Z. Каков **способ передачи** определяет компилятор( что не есть хорошо, т.к. различные компиляторы в одной и той же ситуации могут выбрать разные способы передачи, что приведёт к различной работе программ).

Пользователь определяет **семантику** использования.

Запись активации

…..

……

Место для **формальных** параметров

……

Формальные параметры – те, которые объявлены в заголовке подпрограммы и используются в теле. Большинство ЯП переменные, которые объявлены в заголовке, считают частью тела.

Фактические параметры – те, которые передаются в подпрограмму при её вызове.

При вызове подпрограммы фактические параметры, указанные в команде вызова, становятся значениями соответствующих формальных параметров, чем и обеспечивается передача данных в подпрограмму.

|  |  |  |
| --- | --- | --- |
| Способ передачи | Семантика | Что делается |
| По значению | in | При вызове подпрограммы фактические параметры копируются в Запись Активацаии |
| По результату | Out | При выходе из подпрограммы из записи активации формальные параметры копируются в фактические |
| По значению и результату | inout | При вызове подпрограммы фактические параметры копируются в Запись Активацаии  При выходе из подпрограммы из записи активации формальные параметры обратно копируются в фактические |
| По Адресу | Любая | При передаче по Адресу в Запись активации копируется адрес фактического параметры. Именование и разыменование происходят автоматически |

* В **Фортране** обычно параметры передаются по адресу, но когда передаётся простой объект данных, чтобы не происходило лишних операций разыменования, можно передавать «по значению и результату»( /<параметр>/)
* В **Аде-83** способ передачи зависел от компилятора, т.е. компилятор сам выбирал способ передачи в зависимости от ситуации. Пример программы, в которой это важно:

Procedure P(inout X : T; inout Y : T)

X := <newvalueX>;

<возбуждение исключения>

Y := <newvalueY>;

End P;

….

P(a,a);

Предположим, что оно не обрабатывается. Тогда запись активации пропадает. Значит, если была передача по ссылке, то значение «а» изменится, а если по значению и результату, то не изменится, т.к. копировать в «а» будет нечего.

**Энтропия** – явление, при котором программа может выдавать различные результаты при одних и тех же исходных данных. Если в программе есть энтропия, то это очень плохо. Очевидно, что при программировании на Ада риск энтропии значительно повышается, т.к. не известно какой способ передачи выберет в этот раз компилятор.

* В **Ада-95** – по значению, по ссылке
* В **Си** – не определяется семантика использования. Способ передачи только по значению
* В **Си++** – по значению, по адресу (ссылке). Контроль семантики: in – ссылка константная, out, inout – не константная

void f( T &); //Компилятор считает, что f будет менять значение => константный объект //передавать нельзя

Это указание для компилятора, чтобы он следил за соблюдением семантики

class X

{  
 void f(); //неконстантная функция;

}

……..

const X a;// X \* const this; внутри методов

a.f();// ошибка!!

class X

{  
 void f() const; //константная функция;

}

……..

const X a;// X \* const this; внутри методов

a.f();// Правильно!!

* **С#, Java**

void f( T x) {….;}

……

T a; // a – ссылка, если Т – объект

f(a); // передаётся ссылка

Оба языка поддерживают идею примитивных типов (которые в C# являются подмножеством типов-значений — value types), и оба для трансляции примитивных типов в объектные обеспечивают их автоматическое «заворачивание» в объекты (boxing) и «разворачивание» (unboxing) (в Java — начиная с версии 1.5).

**object** – предок всех классов. => Объект любого класса неявно приводится к типу object.

object o;

int i;

o = i;

i = o; // ошибка!

Автоупаковка, Автораспаковка:

o = i; // o = new Integer(i) – Java

// o = new Int32(i) – C#

Так что если функция объявляется как void f(object o), то в неё можно передавать любой объект( для примитивных типов будет производится автоупраковка\распаковка)

* **Java:**

В Java параметры метода передаются только по значению, но поскольку для экземпляров классов передаётся ссылка, ничто не мешает изменить в методе экземпляр, переданный через параметр. Структур в Java нет.

Передача объектов примитивных типов в методы «как по ссылке» выполняется через классы-обёртки:

void f(Integer X){…X = ….; }

…..

int i = 1;

Integer px = new Integer(i);

f(px);

i = px;

Integer – **класс-обёртка** для примитивного типа «int». Суть способа – преобразовать объект примитивного типа в объект класса и работать внутри функции с объектом класса.

* **C#**

для C#создана более развитая терминология**:**

**Тип-значение(value type)** – тип, объекты которого передаются по значению. Если где-то нужен объект такого типа, то отводится место под сам объект. типами-значениями являются простые(примитивные) типы данных и структуры

**Референциальный тип** – тип, объекты которого передаются по ссылке. Если где-то нужен объект такого типа, то отводится место под адрес. Референциальными типами являются классы(любой массив – наследник класса Array, строка это объект класса String, и т.д.)

В принципе, в C# можно передавать объекты простых типов в функции с помощью классов-обрёток, но C# также поддерживает явное описание передачи параметров по ссылке – ключевые слова ref и out. «ref» реализует семантику inout, а «out» реализует семантику out. При использовании out компилятор вставляет квазистатический контроль на наличие в методе присваивания значения, зато не требует инициализированность фактического параметра перед вызовом метода.

void f(ref int x){x = -3;}

….

int a = 0;

f( ref a); // а будет передано по ссылке, если бы объект «а» был структурой, то он так же // передался бы по ссылке

**Разрешать или не разрешать функции с переменным количеством параметров?**

**Си**

В Си можно было создавать функции с переменным количеством параметров при помощи библиотеки «stdargs»: va\_list, va\_start, va\_end, va\_next и т.д

**Си#**

void f(param object [] args){…. args [0] = …; }

void g(object [] args){…. args [0] = …; }

….

f(a , b);

f();

f(1);

f(new X[]{new X(). new X()});// Ошибка!!

f(new X(), new X()); // Правильно!

g(new X[]{new X(). new X()});// Правильно!!

**java**

void func(Object a[])

{

for(int i = 0;i < a.length; i++)

System.out.println(a[i]);

}

Для **java 1.5** – func(1,2,3, new Object(), "word");

Для **java 1.4** – func(new Object[] {1, 2, "some string"});

**Передача параметров по имени**

**Алгол-60**

Передаём сам объект «как он есть» . Фактически передаётся его идентификатор как строчка. Далее в теле процедуры идентификатор формального параметра заменяется на идентификатор фактического

Обоснование невозможности написание процедруры **swap** на Algol-60:

procedure Swap(a,b); //a, b передаются по имени

Begin

T tmp;

tmp := a; a:= b; b := tmp;

End;

….

swap(i, A[i]);

T tmp;

tmp := i;

i := A[i];

A[i] := tmp;// A [ A[i] ] := i; неправильно!!!

swap(A[i], i);

T tmp;

tmp := A[i];

A[i] := i;

i:= tmp;// i := A[i] – всё правильно

Решение проблемы: С каждым параметром передавалась процедура **thunk**, которая выполнялась при каждом обращении к фактическому параметру.

**Параметры по умолчанию**

В С++ можно задавать параметры по умолчанию: void f(int a = 4){ …;}

В C# эту возможность убрали. Вместо этого предлагается использовать перегрузку:

f(){ f(1, 2); }

f(int i ){ f(i , 2) ;}

f(int i, int j ) { … ; }

**Классы**

Принципиальное отличие класса от модуля заключается в том, что класс – это тип данных, а модуль нет. Но во многих вещах они похожи.

**Тип Данных = Структура Данных + Множество Операций над этими данными**

В C#, Java всё является классами или находится в классах в качестве статических челнов. Даже математические функции находятся в классе – System.Math. И для вызова функции cos(x) требуется написать Math.Cos(x);

Наибольшее сходство между классом и модулем достигается если класс содержит только статические методы и поля. При этом такой класс, как правило, реализуется в виде модуля.

**п1. Понятие членов класса**

синтаксис в **C++.Java,C#:**

class Name

{

….

Определение членов класса

…..

}

В Си++ допускается вынесение определений, т.е. В Си++ можно члены класса лишь объявлять. В Java, C# все определения должны быть внутри класса

синтаксис в **Delphi:**

type T =

class (наследование)

*обявление* членов класса

end;

Члены класса:

* Члены-данные
* Члены-типв
* Члены-функции(методы)

**Чем члены-функции отличаются от обычных функций?**

Такой функции при вызове всегда передаётся указатель «**this**»(в Delphi «**self**») на объект в памяти, от имени которого вызывается функция.

|  |  |  |
| --- | --- | --- |
| **Java,C#,**  T x;  x = new T(«параметры конструктора»);  В первой строчке определяется ссылка на объект (выделяется память для хранения ссылки), место в динамической памяти под объект не отводится. Во второй непосредственно отводится место в динамической памяти («куче») для объекта и адрес присваивается ссылке на объект. | **C++.**  T x;  T x(«параметры конструктора»);  T x = T(«параметры конструктора»);  В этих определениях выделяется место не под ссылку на объект, а под сам объект (не в динамической памяти). Чтобы выделить место в динамической памяти, нужно использовать операцию «new» | **Dephpi**  x : T  x – ссылка, её ещё надо проинициализировать. |

Ещё раз о структура в Си#:

struct Name // не имеет ссылочной семантики

{

….

<определения членов>

….

}

…….

Name x;// память отводится здесь же

x = new Name(<параметры>); // Явное выделение динамической памяти

Name [] arr = new Name[50];// В памяти отведётся место под 50 объектов типа «Name», //а не под 50 указателей

От структур нельзя наследовать. Сами структуры неявно наследуются от класс System.Struct

**Обращение**

«имя\_объекта».«имя\_члена»

Для членов-данных по определению выполнение операции обращения к элементу класса «.»(class member acces operator) является просто смещением относительно адреса «this»(«self»)

class X

{  
 ….

объявления \ определения для Си++, определения для C#, Java

…..

}

Внутри всех функций-членов члены класса видимы непосредственно.

Однако формальные параметры метода класса относятся к блоку метода и могут закрывать члены класса. Тогда доступ к членам классам можно получить через указатель this при помощи операции обращение к элементу класса «.»: this.«имя члена».

В **Delphi** формальные параметры функций-членов находятся в той же области видимости, что и все остальные члены класса и, следовательно, не могут с ними совпадать.

**Члены-типы**

STL – набор соглашений. Одно из соглашений – контейнеры должны сами себя описывать. В Delphi членов-типов нет.

**Статические члены**

SmallTank **class variable**

**instance variable**

**class variable** – члены-данные класса, которые принадлежат всем экземплярам класса.

**instance variable –** принадлежат экземплярам класса, у экземпляра своя **instance variable.**

С точки зрении Си++ статические члены классов отличаются от глобальных только областью видимости.

class T

{

….

static int x;

static void f();

……

}

…

T t;

t.x;//операция доступа

//или, что тоже самое

T::x//операция разрешения видимости

t.f();//операция доступа

//или, что тоже самое

T::f() //операция разрешения видимости

В C#,Java,Delphi обращение к статическим членам происходит только через тип класса.

В статических функциях нет ths/self => в них нельзя использовать нестатические члены класса, т.к. по умолчанию все обращения к нестатическим членам идут через указатель ths/self

В C#, Java статические члены используются намного чаще, чем в Си++, Delphi, т.к. в C#, Java нет глобальных функций и переменных:

public class Name

{

….

public static void Main(string [] args)

….

}

**Паттерн Singleton (Одиночка)**

Если установка соединения между клиентом и сервером слишком трудна, требует больших затрат ресурсов и времени, то неэффективно каждый раз устанавливать его заново. Нужно иметь единственный экземпляр соединения и запретить произвольное создание экземпляров (для этого можно, например, сделать конструктор приватным).

class Singleton

{

static private Singleton \*Instance;//**объявление**,не путать **с определением**

private:  
 Singleton();

Singleton(const Singleton &);

public:  
 static getInstance()

{

if(Instance == NULL)

Instance = new Singleton(); //здесь нужен конструктор

return instance; //здесь нужен конструктор копирования

}

}

Определение объекта подразумевает размещение его в памяти.

Т.к. внутри класса имеется только объявление static private Singleton \*Instance, то где-то вне класса нужно произвести определение этого статического члена:

Singleton \* Singleton::Instance = NULL; //отличается от определения (размещения) глобальной переменной только тем, что кроме этого места Singleton::Instance нигде нельзя будет далее использовать, т.к. это скрытый член класса.

Это единственный случай, когда можно инициализировать скрытые члены класса

**Вложенные типы данных (классы)**

class Outer

{  
 ….

class Inner //Inner – обычный класс, но с ограниченной областью   
 { //видимости

…..

};

….

};

Все классы являются статическими членами. Ко всем именам правила прав доступа применяются одинаково, т.е. специфика имени не участвует в разрешении прав доступа.

**C#**

В **C#** имеется понятие «статического класса»

static class Mod

{

public static void f () { ….;}

public static int i;

}

**Статический класс** – служит контейнером для статических членов. От статических классов нельзя наследовать. Нельзя создавать объекты статических классов. Статические классы подчёркивают их чисто модульную природу.

Без «static» - обычный класс в понятии Си++. На вложенность классов не влияет.

**Статические члены** – набор ресурсов, независимых от какого-либо экземпляра.

**Java**

Статический импорт – импорт всех статических членов класса. Часто применяется к математическим функциям.

**Статические классы в Java:**

public class Outer

{  
 ….

public static class Inner //Тоже самое, что и в C#, C++ без «static»   
 {

…..

};

….

};

Это сделано для того, что доступ к Inner был через Outer

Декларируются внутри основного класса и обозначаются ключевым словом static. Не имеют доступа к членам внешнего класса за исключением статических. Может содержать статические поля, методы и классы, в отличие от других типов внутренних классов в языке Java.

**Внутренние классы в Java:** без «static»

public class Outer // «Outer» является владельцем «Inner»

{  
 ….

static class Inner //Тоже самое, что и в C#, C++ без «static»   
 {

…..

};

….

};

Декларируются внутри основного класса. В отличие от статических внутренних классов, имеют доступ к членам внешнего класса, например «Outer.this». Не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант).

**Инкапсуляция**

Единица защиты – тип класса или экземпляр класса

Атом защиты – член класса.

В современных языках единицей защиты является тип класса. Правила защиты во всех языках определяются для всех экземпляров одинаково. Нельзя один экземпляр защитить больше чем другой.

Управление инкапсуляцией:

* Управление доступом – C++, C#, D
* Управление видимостью – Java

Управление видимостью – «private»-членов как бы просто нет для других классов, они «невидимы».

Управление доступом – все не скрытые (не переопределённые) члены видны, т.е. компилятор постоянно «знает» об их существовании, но при обращении проверяются права на доступ. При попытке обращения к недоступному члену выдаётся ошибка.

class X

{

public:

virtual void f();

void h();

};

class Y: public X

{

private:

virtual void f();

void h();

};

class Z: public Y

{

public:

virtual void f(); //В Java заместится функция, видимая в данный момент – X::f

void g(){ … h();…. } //В **Си++** для этой строчки будет выдана ошибка – с точки зрения управления доступом попытка вызова функции Y::h() незаконна, т.к. она приватна(к ней нет доступа вне класса Y).

В **Java** спокойно вызовется функция X::h() и никакой ошибки не будет, т.к. функция Y::h() просто вычеркнута из рассмотрения (невидима)

}

**Три уровня инкапсуляции:**

1. public
2. private
3. protected

«свой» - член данного класса

«чужой» - все внешние классы

«свои» - члены наследованных классов

**public** разрешает доступ всем

**private** разрешает доступ только «своему»

**protected** разрешает доступ «своим» и «своему»

class X

{

….  
 protected: void f();

……

};

class Z:public X

{

….

…..

};

class Y:public X

{

void g()

{

f(); //так можно

Y another;

another.f(); // так тоже можно

Z getanother;

getanother.f(); // В C# и Java так нельзя (у класса Y и класса Z независимые контракты // с классом X). В C++ так можно

(X)getanother.f(); // В C# и Java так будет работать (в C++ и подавно)

};

**Перегрузка операций**

a += b;// ~a.operator+=(b)

a + b;

Есть два пути вычисления этого выражения:

a.operator+(b); или

operator+(a,b);

class X

{

public:

X operator+(X & a);

X(T);

};

//либо

X operator +(X & a, X & b);

T t;

X a,b;

a = a + b;

a = t + b; // ищет T.operator+(X), operator+(T,X) и т.д.

Правильным является **operator+(X(T),T)**

Если требуется, чтобы доступ к приватным членам был не только у «своего», можно для этой этого объявить нужную дружественную конструкцию в теле класса:

friend «объявление друга»;// Можно писать сразу определение. Другом может быть функция или целый класс

friend «прототип глобальной функции»

friend «прототип функции-члена другого класса»

friend class «имя класса-друга»;// Все методы этого класса становятся дружественными

В Delphi, C#, Java друзей нет

В них реализованы этот механизм реализован немного по-другому:

Delphi **UNIT**

Java **package**

C# **assembly**

В **Java** по умолчанию пакетный доступ. Это значит, что использовать класс может каждый класс из этого пакета. Если класс объявить как «public class …», то он будет доступен и вне пакета. Использовать класс – наследовать, создавать объекты.

**C#:**

**Сборка** – надъязыковое понятие в .NerFramework. Сборка представляет собой совокупность файлов + манифест сборки, Любая сборка, статическая или динамическая, содержит коллекцию данных с описанием того, как ее элементы связаны друг с другом. Эти метаданные содержатся в манифесте сборки. Манифест сборки содержит все метаданные, необходимые для задания требований сборки к версиям и удостоверения безопасности, а также все метаданные, необходимые для определения области действия сборки и разрешения ссылок на ресурсы и классы.

Внутри сборки идёт разделение на пространства имён, которые содержат описания классов.

Для использования какого-либо пространства имён нужно сначала подключить сборку, содержащую его. Пространство имён может быть «размазана» по нескольким сборкам.

В C# для членов классов имеются следующие квалификаторы доступа:

* public
* private // по умолчанию
* protected
* internal – член доступен только в классах из сборки
* internal protected – член доступен только в классах-наследниках, находящихся в сборке

Для самих классов:

* public – класс можно использовать в любых классах
* internal – класс можно использовать только в классах из его сборки (по умолчанию)

public class X //Этот класс может унаследовать любой класс

{

….

internal int a; // Это переменная доступна только внутри сборки

….

}

internal class Y //Этот класс может унаследовать только класс из сборки

{

….

}

Delphi

type T = class

…. // здесь объявляются члены, видимые везде их данного модуля и не видимые

// других

public

….

protected

….

private

…..

end;

UNIT – единица дистрибуции

**Специальные функции**

Функции-члены, обладающие семантикой обычных функций-членов, о которых компилятор имеет дополнительную информацию.

**Конструктор** – порождение, инициализация

**Деструктор** – уничтожение (В Java и C# не деструкторов, вместо это можно сделать собственный метод Destroy() )

**Управление жизненным циклом объекта**

* создание, инициализация
* использование
* уничтожение

У конструктора нет возвращаемого значения.

Т.к. все объекты в C# и Java и Delphi размещаются в динамической памяти, то в этих языках обязательна операция явного размещения объектов:

X = new X(); // В Си++ X \* a = new X;

Синтаксис конструкторов и деструкторов:

C++. C#, Java, D

class X

{

X(«параметры»);// В С# и Java обязательно определение тела

}

Delphi

type X = class

constructor Create; // Имя конструктора произвольное

destructor Destroy; // имя деструктора произвольное

end;

…..

a:X;

….

a := X.Create;

В C++, C#, Java конструкторы не наследуются, но могут автоматически генерироваться компилятором по определённым правилам.

**Классификация конструкторов:**

1. Конструктор умолчания X();
2. Конструктор копирования X(X &); X(const X & );
3. Конструктор преобразования X(T); X(T &); X(const T &);

В классах из простанства имён System платформы .NetFramework не определены конструкторы копирования. Вместо этого, если это предусмотрено проектировщиками, имеется метод clone();

**Абстрактные классы и интерфейсы**

Сами по себе иерархии классов бесполезны, если в них нету динамически связанных методов.

* Если существуют признаки, общие для всех объектов в класса иерархии, то эти признаки целесообразно поместить в базовый класс
* У каждого объекта класса имеется состояние (текущие значения параметров) и поведение (методы класса)
* Некоторая функциональность (особенности поведения), общая для всех классов в иерархии, может быть реализована только при дальнейшей детализации в производных классах.

=> приходим к понятию абстрактного базового класса

В языках Delphi, Оберон-2, TP 5.5, C++, Java, C# имеется языковая поддержка абстрактных классов.

Пример:

**Figure** – абстрактная фигура

Общие данные – (x,y) – координаты фигуры

Общие методы – Draw(), Move() – не могут быть реализованы для абстрактной фигуры

Что будет если просто не определить витуальную функцию в базовом классе:

Полиморфный класс – класс, в котором есть виртуальные функции (и. следовательно, таблица виртуальных функций)

class X

{

public:

virtual void f(); //Объявлена, но не определена

void g();//Объявлена, но не определена

}

class Y: public X

{

public

virtual void f() {…;}//Замещена

}

….

X \* py = new Y(); // уже здесь компилятор выдаст ошибку из-за того, т.к. непонятно что записывать в таблицу виртуальных функций

py -> f();

X a; // здесь компилятор также выдаст ошибку из-за того, из-за того, что не сможет заполнить таблицу виртуальных функций. Если бы функция X:: f() не была виртуальной, то ошибки здесь не было бы.

a.g();// ошибка, т.к X::g() не определена

Решение проблемы – языковая поддержка

В **C#, Java**:

**abstract** перед классом и перед функцией, для которой не существует реализации на данном уровне детализации. Такие функции и классы, их содержащие, называют аббстрактными

abstract class Figure //Абстрактный класс

{

abstract public void Draw(); // Функция без реализации

abstract public void Move();// Функция без реализации

}

**В C++:**

Чисто виртуальная функция (абстрактная) – **virtual «прототип» = 0;**

**В Аде:**

procedure P( X : T ) is abstract;

где T – тегированный тип.

**Объекты абстрактных классов нельзя создавать.**

При вызове виртуальной функции в конструкторе виртуальность вызова снимается:

сlass X{…}

class Y: public X{…}

class Z: public Y {…}

При создании объекта класса Z сначала вызовется конструктор класса X, потом класса Y и в самом конце класса Z. Нельзя вызвать в конструкторе класса Y замещённую функцию из Z, потому что объект Z ещё не до конца создан и для него ещё даже нет таблицы виртуальных функций.

Существует метод, который не должен быть чисто виртуальным – деструктор. Он всегда должен быть виртуальным и реализованным.

Base \* px = new Derived;

«использование»

delete(px); // уничтожиться должен объект класса Derived

**Различие между абстрактными классами и абстрактными типами данных:**

В абстрактных классах абстрагируются от реализаций некоторых методов. В абстрактных типах данных абстрагируются от всей структуры.

Например множество – каждое множество должно поддерживать операции

* include(const T &)
* exclude( const T &)

При этом, естественно, реализация этих методов зависит от типа элементов, способа хранения и т.д.

class Iset

{

virtual void include(const T &) = 0;

virtual exclude(const T &) = 0;

«статические члены»

}

Такой класс называется **класс-интерфейс.** В таких классах не имеет смысл объявлять нестатические поля, т.к. не реализовано ни одного методы для работы с ними.

class Slist{…}

class Slist\_Set: public Iset, private Slist {…; } // Iset – интерфейс

Iset \* Move()

{

….

return new Slist\_set(«параметры»);

}

В C# и Java, в отличие от C++, существует языкового понятия интерфейса:

interface «имя»

{

«объявления членов»

}

Внутри интерфейса могут быть статические поля. Поля без «static» будут восприняты как статические. Также членами интерфейса могут быть методы и свойства, которые считаются чисто виртуальными и публичными. Т.е. интерфейс – чистый контракт, не реализующий структуру.

Если класс наследует интерфейс и не определяет все методы интерфейса, то он становится абстрактным.

**Множественное наследование**

Только в C++ поддерживается множественное наследование. В C# и Java множественное наследование поддерживается только для интерфейсов.

C#: class D: [«класс»]{, «интерфейс»}

Java: class D extends Base implements «интерфейс» {, «интерфейс»}

Проблемы, связанные с множественным наследованием:

* **Конфликт имён**

**Java:**

interface Icard

{  
 void Draw(){ …; }// Раздать карты

}

interface IGUIControl

{

{

void Draw(){ …; } //нарисовать колоду

}

class Sample implements ICard, IGUIControl {…; }// Ошибка!!

**C++**

class D public I1, public I2

{

virtual void I1::f(){…;} // Операция разрешения видимости

virtual void I2::f(){…;} // Операция разрешения видимости

}

…

D \* px = new D;

px-> f(); // ошибка

px->I1::f(); // ошибка

((I1 \*)px) -> f(); //Работает! **Явное приведение**

**C#:** Неявная и явная реализация интерфейсов

interface ISample

{  
 void f();

}

class CoClass: ISample

{  
 public void f(){…} // неявная реализация, «public» перед void f() обязателен

}

class CoClass2: ISample

{

void ISample.f(){…} // явная реализация. Запрещено указывать public, private, protected.

}

CoClass x = new CoClass();

x.f();// работает!

CoClass2 x = new CoClass2()’

x.f();/// ошибка! Цитата компилятора: «CoClass2' does not contain a definition for 'f'

((ISample)x).f(); // работает! **Явное приведение**

В платформе .Net имеется класс

class FileStream:IDisposable

{

….

void IDisposable.Dispose(){…;}

void Close(){ ((IDisposable)this).Dispose();}

….

}

FileStream реализует интерфейс IDisposable явным образом, т.е. через объект FileStream нельзя вызвать метод Dispose. Вместо этого имеется не виртуальный метод Close, который явно вызывается Dispose внутри себя.

class IControl

{

void Paint();

}

interface IEdit:IControl

{

…..

}

interface IDropList:Icontrol

{  
 …..

}

class ComboBox : IDropList, IEdit // Элемент управления ComboBox реализует оба //интерфейса. И для каждого из них должен быть свой //Paint;

{

void IDropList.Paint(){…; } //Явная реализация

void IEdit.Paint(){….;}//Явная реализация

public void Paint() //Этот метод будет вызываться по умолчанию

{

.…

((IEdit)this).Paint(); // Явный вызов

…..

}

}

В C# реализованные методы интерфейсов считаются по умолчанию «sealed» - запечатанными. В наследниках они перекрываются.

class ExampleClass : IControl

{  
 public void Paint(){ … ;}

}

class FancyComBox: ExampleClass

{

public void Paint(){….; } // Компилятор выдаст предупреждение для этой строчки, в котором сообщит, что «FancyComBox.Paint() скрывает унаследованный метод ExampleClass.Paint(). Используйте ключевое слово «new», если это сделано целенаправленно.» Т.е. если поставить «new» перед определением, то предупреждение исчезнет.

public void override Paint (){… ; } // Для этой строчки компилятор выдаст ошибку, т.к. нельзя замещать(переопределять) методы, не указанные в базовых классах как virtual, override или abstact

}

В целом множественное наследование можно заменить включением (агрегацией).

Как было показано выше, конфликт имён решается через явное приведение и уточнение класса.

Но существует ещё одна проблема – эффективность динамического полиморфизма (виртуальных функций)

Эта проблема возникает только при наследовании по данным.

Рассмотрим одиночное наследование:

class A

{

public:

A(){};

Подъобъект Avptr**объект класса С**a1a2a3c1

Таблица виртуальных функций A + C

**адрес C::a(),** которая замещает функцию A::a()**адрес A::second()адрес C::goo()**

virtual void a(){ a1 = 1;};

virtual void second(){..;}

int a1, a2, a3;

};

class C : public A

{

public:

C() : A(){};

virtual void goo(){};

void a(){}; // переопределение

int c1;

};

….

C c;

Если в классе C переопределить метод, то в соответствующую ячейку в таблице виртуальных функций будет записан указатель на новый метод. Если же в классе C добавляются новые функции – они дописываются в конец таблицы. При вызове методов никаких лишних действий не происходит.

Подъобъект Avptr**объект класса С**a1a2a3Подъобъект Bvptrb1b2b3c1

Таблица виртуальных функций B

**адрес C::bar(),** которая замещает функцию B::bar()**адрес B::bar()**

Таблица виртуальных функций A + C

**адрес C::a(),** которая замещает функцию A::a()**адрес A::second()адрес C::goo()**

А теперь рассмотрим множественное наследование:

class A

{

public:

A(){};

virtual void a(){ a1 = 1;};

virtual void second(){..;}

int a1, a2, a3;

};

class B

{

public:

B(){};

virtual void bar(){};

virtual void bbar(){};

int b1, b2, b3;

};

class C : public A

{

public:

C() : A(){};

virtual void goo(){};// Собственная новая виртуальная функция

void a(){}; // переопределение

void bar();// переопределение

int c1;

};

….

C c;

Тут надо обратить внимание на следующее:

* Таблица виртуальных методов самого нижнего класса в иерархии доступна через первый указатель vptr.
* Каждый подобъект, который содержит виртуальные методы, имеет свою таблицу виртуальных функций.

Если в классе C переопределить метод, то в соответствующую ячейку в таблице родительского объекта будет записан указатель на новый метод. Если же в классе C добавляются новые функции – они дописываются в конец первой таблицы.

Такой алгоритм становится понятен, если рассмотреть возможные преобразования типов:

* С -> A. Через указатель на класс A можно вызывать только методы, которые прописаны в этом классе.
* C -> B. Ситуация аналогична, только мы можем вызывать виртуальные методы, определенные в классе B.

Новые виртуальные методы (которых нет в родительских классах) можно использовать только через указатель на класс C. В этом случае всегда используется первая таблица виртуальных функций.

**Сложность реализации заключается в следующем:**

Во время преобразования типов меняется адрес указателя:

C c;

B \*p = &c;

Указатель p будет содержать **адрес** **объекта c + смещение подобъекта B**. Т.е. все вызовы методов через такой указатель будут использовать вторую таблицу виртуальных методов объекта C. Но ведь в такой ситуации при вызове переопределённой в **C** функции через указатель на **B** в эту функцию передастся неправильный указатель this! Он будет указывать не на C, как это нужно, а на B.

Приходится **расширять таблицу виртуальных функций** добавлением в неё смещения от указателя на объект класса до таблицы виртуальных функций для каждой функции. Если виртуальная функция из B переопределена в C, то для неё такое смещение будет равно (-**смещение подобъекта B).** Если же не была переопределена, то оно будет равно нулю. Для всех виртуальных функций из класса A это смещение будет нулевым, т.к. указатель на подобъект A совпадает с указателем на весь объект C(объект А находится в начале объекта C). Теперь в функцию можно передать правильный указатель:

**this = current\_this + offset**

где **current\_this –** на подобъект, через который вызывается функция. **offset** – значение, которое берётся из **расширенной таблицы виртуальных функций**.

Без наследования по данным таких проблем не возникает, т.к. указатель на таблицу виртуальных функций всегда один.

**Ромбовидное и не ромбовидное наследование**

Не ромбовидное: : В объекте Z будет два экземпляра объекта A с разными реализациями таблицы виртуальных функций

AПодобъект XXAПодобъект YYZ

сlass A{ .. ;}

class X:public A{ …; }

class Y:public A{… ; }

class Z: public X, public Y {…;}

A A

X Y

Z

Ромбовидное: В объекте Z будет только один экземпляр объекта A

AXYZ

сlass A{ .. ;}

class X: public virtual A{ …; }

class Y: public virtual A{… ; }

class Z: public X, public Y {…;}

A

X Y

Z