Языки программирования и методы трансляции

Типы данных

Типы данных

Что такое тип данных?

Денотационный подход: набор значений

Структурный подход: данные с определенной внутренней структурой, описанной в терминах небольшого набора примитивных типов

Подход с точки зрения реализации: класс эквивалентности объектов

Подход с точки зрения абстракции данных: набор четко определенных операций, применимых к объектам этого типа

Типы данных

- Зачем нужны типы данных?
  - неявный контекст
  - проверка осмысленности некоторых операций
    - Проверка типов не способна предотвратить все бессмысленные операции.
    - Она выявляет достаточное количество таких ошибок, чтобы быть полезной.
- Полиморфизм: компилятор игнорирует некоторые детали; они определяются во время выполнения программы за счет контекста, задаваемого типом.

Проверка типов

Строгая типизация — язык не допускает применение операции к данным, для которых она не определена.

Статическая типизация — все необходимые проверки можно осуществить во время компиляции.

Проверка типов

Пример

Python — строго типизированный, но не статически типизированный

Ada — статически типизированный

Pascal — почти статически типизированный

Java — строго типизированный; что-то может быть проверено статически, что-то должно проверяться динамически

Терминология

Дискретные типы — конечные или счетные

- целочисленные
- булевы
- символьные
- перечислимые
- ограниченные

Скалярные типы — одномерные

- дискретные
- вещественные

Составные типы

- записи, объединения
- массивы
  - строки
- множества
- указатели
- списки
- файлы

Ортогональность

Ортогональность — важное свойство языка, особенно в отношении его системы типов.

- Набор функциональных возможностей является ортогональным при отсутствии ограничений на совместное использование этих возможностей.

Ортогональность

Пример (Pascal)

Ортогональность: Произвольные типы элементов массивов произвольных типов (в ранних версиях языка Fortran — только скалярные типы)

Неортогональность: Не допускается использовать вариантные записи в качестве произвольных полей других записей (вариантная часть записи должна располагаться после других полей).

Ортогональность делает язык более легким для понимания, использования и описания.

Проверка типов

Для системы типов определены

эквивалентность типов: когда типы двух значений совпадают?

совместимость типов: когда значение типа A может быть использовано в контексте, подразумевающем значение типа B?

логический вывод типа: каков тип выражения, если известны типы операндов?

Совместимость типов и эквивалентность типов

- Совместимость типов — более полезное понятие, так как оно определяет возможные действия.
- Эти термины часто (и не вполне корректно) используются как синонимы.

Эквивалентность типов

Форматирование не имеет значения:

Пример

- `struct { int a, b; }`
- ```
  struct {
      int a, b;
  }
  ```
- ```
  struct
  {
      int a;
      int b;
  }
  ```

Два основных подхода

Эквивалентность структур и эквивалентность имен

- Эквивалентность имен основана на объявлениях типов.
- Эквивалентность структур основана на смысле этих объявлений, определяемом некоторым образом.
- В настоящее время эквивалентность имен более популярна.

Эквивалентность типов

Существуют, по крайней мере, два распространенных варианта эквивалентности имен.

- Они по-разному смотрят на то, какие различия в описаниях типов считаются важными.
- Во всех трех схемах определения эквивалентности типов (эквивалентность структур, строгая и нестрогая эквивалентность имен) требуется приведение описания типа к стандартной форме, позволяющей абстрагироваться от очевидно неважных различий в описаниях.

Эквивалентность типов

- Эквивалентность структур выявляется простым сравнением описаний типов с заменой всех имен на описания соответствующих типов.
  - «развернуть» описание до встроенных типов
- Типы считаются эквивалентными, если их развернутые описания совпадают.

Приведение типов

- Когда выражение одного типа используется в контексте, в котором ожидается значение другого типа, обычно возникает ошибка типов.
- Но:

Пример

```
var a : integer;
b, c : real;
...
c := a + b;
```

Неявное приведение типов

- Подобный код допустим во многих языках — значение выражения приводится к нужному типу.
- Приведение может быть основано только на типах операндов или может также учитывать тип, ожидаемый в данном контексте.
- В языке Fortran все многочисленные правила приведения типов основаны на типах операндов.

Неявное приведение типов

C: многочисленные, но более простые правила

- `float` в выражениях преобразуется в `double`
- `short int` и `char` в выражениях преобразуются в `int`
- иногда происходит потеря точности при присваивании

Неявное приведение типов

Приведение типов ослабляет проверку типов.

- Сейчас считается, что приведение типов скорее вредно.
- В некоторых языках (Modula 2, Ada) приведение типов запрещено.
- Но в C++ приведение типов используется чрезвычайно активно.
- Правила приведения типов — одна из наиболее сложных для понимания частей языка.

Явные и неявные приведения типов

- явное приведение типов — type conversion
- неявное приведение типов — type coercion
- в контексте C для обозначения явного приведения используют также слово cast

Типы-записи

- Запись — возможно разнородная совокупность элементов, в которой отдельные элементы различаются по именам
- Вопросы разработки:
  1. Каков синтаксис ссылок на поля записи?
  2. Допускаются ли эллиптические ссылки?

Определение записей

- В языке COBOL вложенные записи вводятся нумерацией уровней; в других языках используются рекурсивные определения
- Ссылки на поля записей
  1. COBOL
     <имя поля> OF <имя записи 1> OF ... OF <имя записи n>
  2. Другие языки (нотация с точкой)
     <имя записи 1>.<имя записи 2>. ... .<имя записи n>.<имя поля>

Определение записей в языке COBOL

- В языке COBOL вложенные записи вводятся нумерацией уровней; в других языках используются рекурсивные определения

Пример

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID PIC X(10).
      05 LAST PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

Определение записей в языке Ada

- Структуры записей определяются ортогональным образом

Пример

```
type Emp_Rec_Type is record
    First: String (1..20);
    Mid: String (1..10);
    Last: String (1..20);
    Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

Ссылки на записи

- Большинство языков используют нотацию с точкой
- Полные ссылки должны включать все имена записей
- В эллиптических ссылках имена записей могут быть опущены, если это не приводит к неоднозначности ссылки

Пример (COBOL)

Эллиптические ссылки на имя служащего:

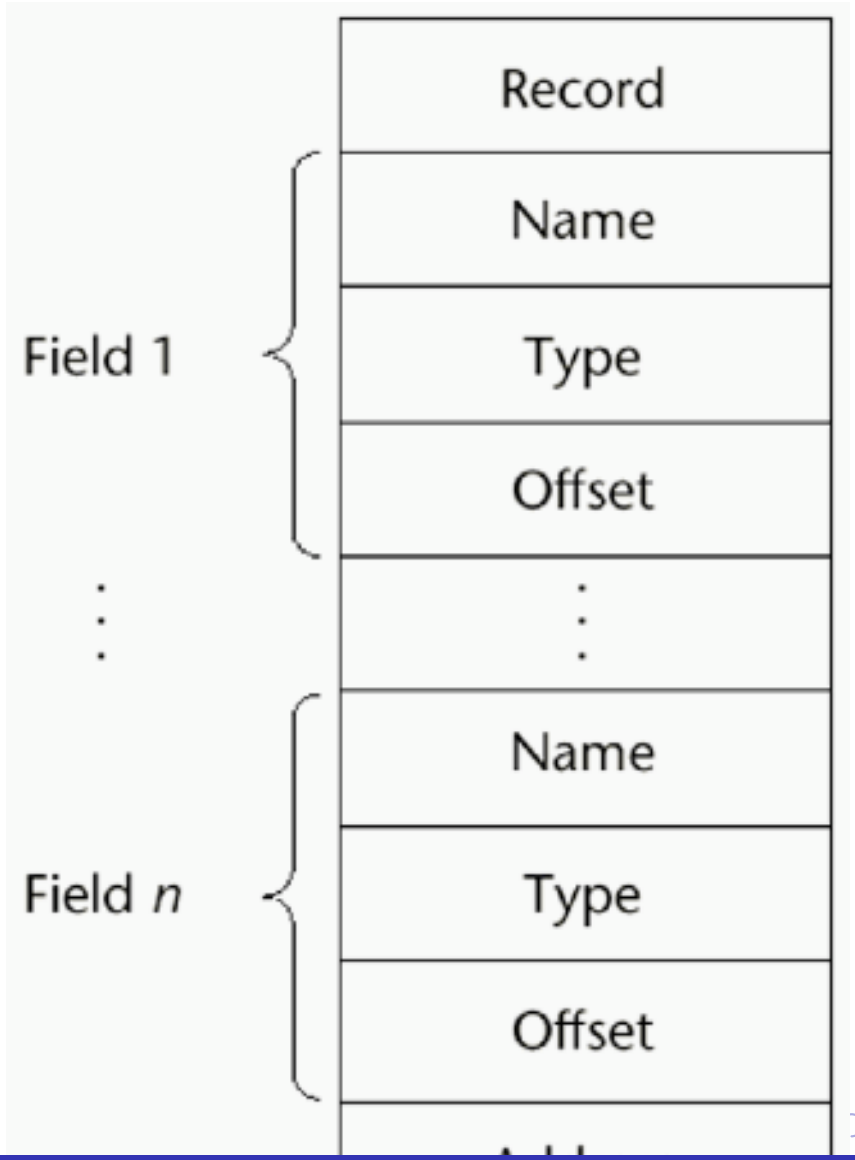
- FIRST
- FIRST OF EMP-NAME
- FIRST OF EMP-REC

Операции над записями

- Широко используется присваивание для операндов одного типа
- В языке Ada поддерживается сравнение записей
- Записи языка Ada могут быть инициализированы при помощи составных литералов
- MOVE_CORRESPONDING в языке COBOL
  - Копирует поле одной записи в соответствующее поле другой записи

Реализация типов-записей

С каждым полем связано смещение адреса относительно начала записи



Записи

- Обычно размещаются в памяти непрерывно.
- Возможны «дырки» в памяти из-за необходимости «выравнивания» данных.
- Хороший компилятор может переупорядочить поля, чтобы минимизировать дырки (компиляторы C как правило это делают).
- Записи, содержащие динамические массивы, сложны в реализации.

Типы-объединения

- Объединение — это тип, переменные которого могут хранить значения разных типов в разные моменты выполнения программы
- Вопросы проектирования
  - Должна ли проверка типов быть обязательной?
  - Должны ли объединения использоваться только в составе записей?

Записи (структуры) и варианты (объединения)

- Объединения (вариантные записи)
  - используют наложения полей в памяти
  - затрудняют проверку типов
- Если не поддерживаются теги (дескрипторы), невозможно определить, что хранится в объединении.

Размеченные и свободные объединения

- В языках Fortran, C и C++ отсутствует проверка типов для объединений — свободные объединения
- Проверка типов для объединений требует, чтобы каждое объединение включало метку типа
  - Поддерживается в языке Ada

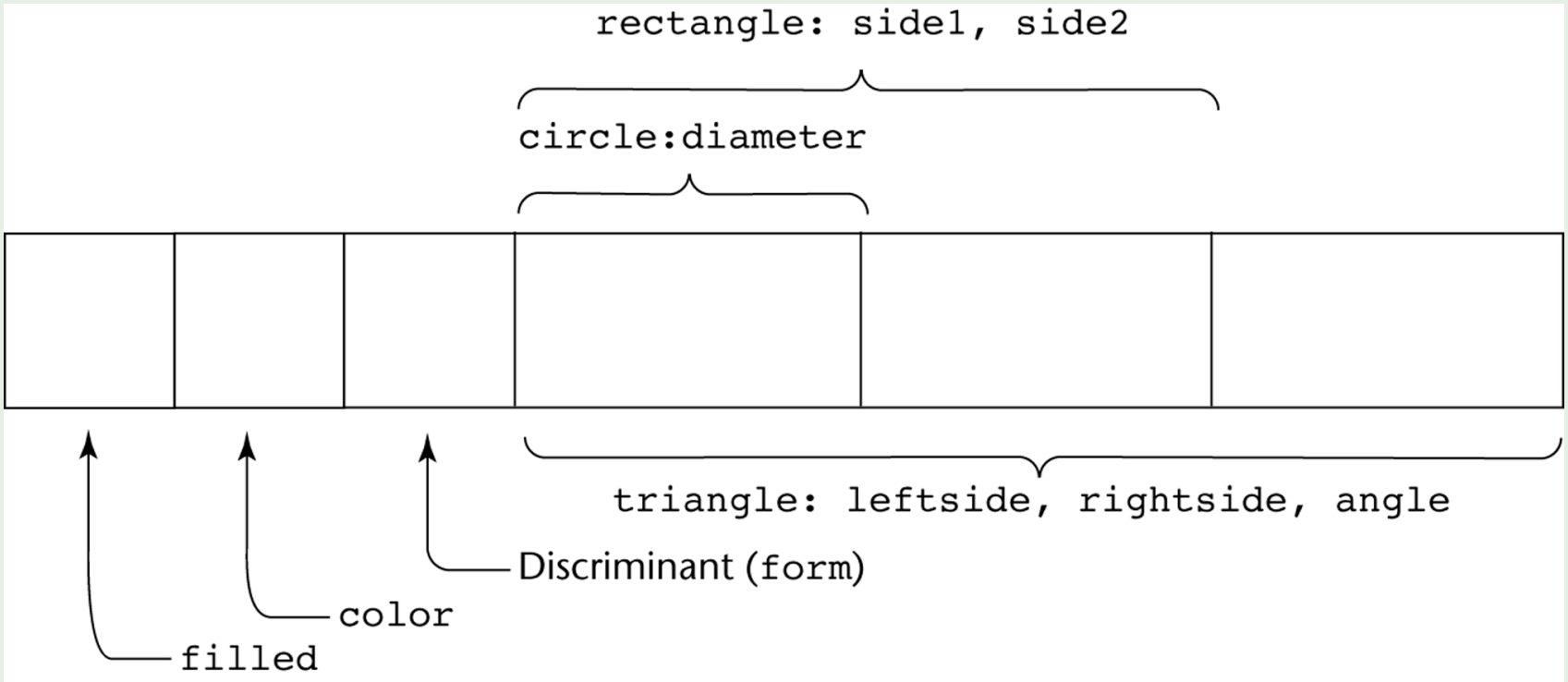
Типы-объединения языка Ada

Пример

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
```

Типы-объединения языка Ada

Пример (Размеченное объединение из трех переменных)



Оценка объединений

- Потенциально небезопасная конструкция
  - без проверки типов
- Java и C# не поддерживают объединения
  - результат внимания, уделяемого вопросам безопасности в современных языках программирования

Связывание индексов и категории массивов

	Статические	Фиксированные стек-динамические	Стек-динамические	Фиксированные динамические	Динамические
Диапазон индексов	Статический	Статический	Динамический	Динамический	Динамический
Выделение памяти	Стати–ческое	Динамическое (при обработке объявления)	Динамическое (при обработке объявления)	Динамическое (по запросу пользователя)	Динамическое (по запросу пользователя)
Изменение диапазона и памяти	Нет	Нет	Нет	Нет	Да
Пример	Статические массивы в C, C++	Нестатические массивы в C, C++	Массивы в языке Ada	Указатели в C, C++	ArrayList в C#; Perl, JavaScript

Инициализация массивов

- В некоторых языках есть возможность задать начальные значения элементам массива в момент выделения памяти

Пример (C, C++; аналогично в Java, C#)

```
int list[] = {4, 5, 7, 83};
```

Пример (Символьные строки в C и C++)

```
char name[] = "freddie";
```

Пример (Массивы строк в C и C++)

```
char * names[] = {"Bob" ,"Jake" ,"Joe"};
```

Пример (Инициализация объектов класса String в Java)

```
String[] names = {"Bob" ,"Jake" ,"Joe"};
```

Операции над массивами

- APL обладает наиболее мощным набором операций над массивами для работы с векторами и матрицами, включающим и унарные операторы (например, оператор, переворачивающий вектор)
- В языке Ada есть оператор присваивания, а также конкатенации
- Fortran располагает поэлементными операциями, т.е. такими, которые применяются к парам элементов массивов

Пример

Применение оператора + к двум массивам позволяет получить массив сумм пар соответствующих элементов этих массивов

Прямоугольные и рваные массивы

- Прямоугольный массив — многомерный массив, в котором все строки содержат одинаковое число элементов и все столбцы содержат одинаковое число элементов
  - `myArray`[3, 7], C#
- Рваная матрица состоит из строк разной длины
  - Возможно, если многомерные массивы реализованы в виде массивов массивов
  - `myArray`[3][7]
  - C, C++, Java, C#

Сечения

- Сечение — подструктура массива; всего лишь механизм оформления ссылок на подструктуры
- Сечения полезны только в языках, располагающих операциями над массивами

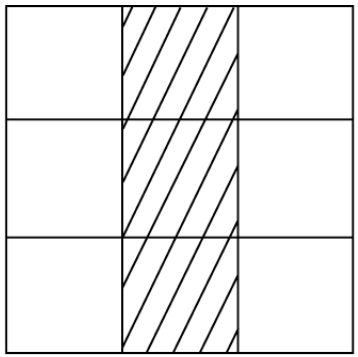
Примеры сечений

Пример (Fortran 95)

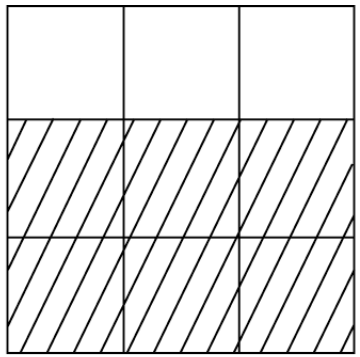
```
Integer,Dimension(10) :: Vector
Integer,Dimension(3,3) :: Mat
Integer,Dimension(3,3,4) :: Cube
```

`Vector(3 : 6)` — массив из четырех элементов

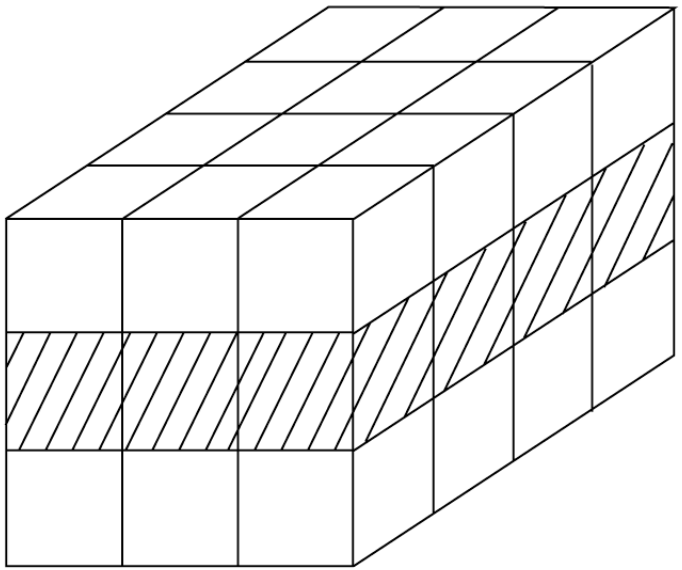
Примеры сечений в языке Fortran 95

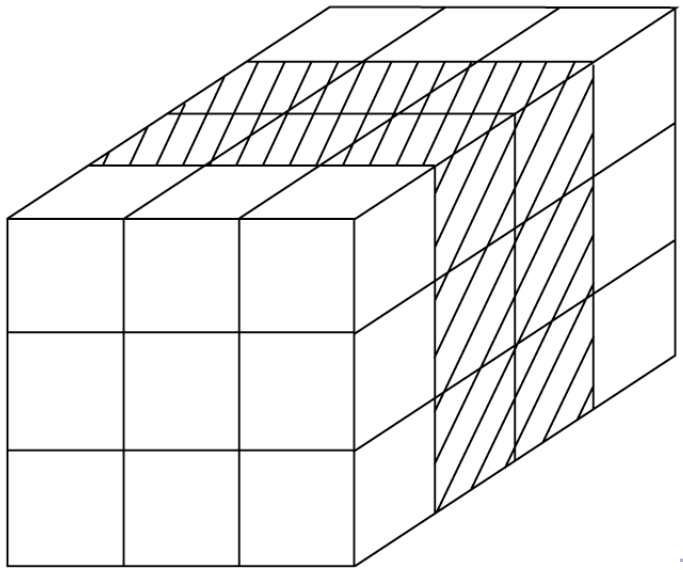


`MAT (1:3, 2)`



`MAT (2:3, 1:3)`





Реализация массивов

- Функция доступа отображает выражение с индексами на адрес в массиве
- Функция доступа для одномерных массивов:

  location(list[k]) = address(list[lower_bound]) +
  + ((k - lower_bound) * element_size)

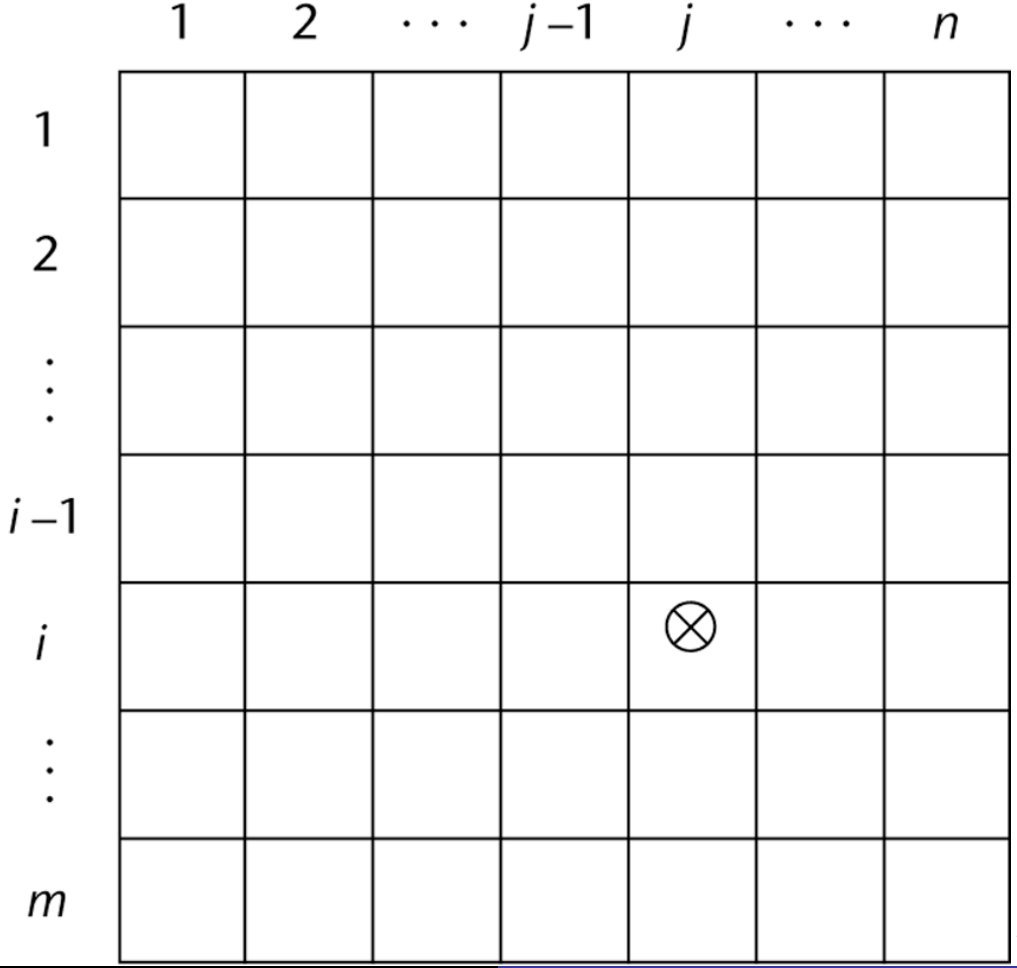
Доступ к элементам многомерных массивов

3	4	7
6	2	5
1	3	8

- Два общепринятых способа хранения массивов:
  - по строкам — используется в большинстве языков
  - 3, 4, 7, 6, 2, 5, 1, 3, 8
  - по столбцам — используется в языке Fortran
  - 3, 6, 1, 4, 2, 3, 7, 5, 8

Обнаружение элемента в многомерном массиве

Общий вид location(a[i, j]) = address(a[row_lb, col_lb]) +
(((i - row_lb) * n) + (j - col_lb)) * element_size



Дескрипторы при компиляции

Массив
Тип элемента
Тип индекса
Нижняя граница индексов
Верхняя граница индексов
Адрес

Одномерный массив

Многомерный массив
Тип элемента
Тип индекса
Число размерностей
Диапазон индексов 1
$\vdots$
Диапазон индексов n
Адрес

Многомерный массив

Ассоциативные массивы

- Ассоциативный массив — неупорядоченная коллекция элементов-значений, проиндексированных соответствующим числом других элементов, называемых ключами
  - Ключи определяются пользователем и должны храниться
  - Каждый элемент: значение + ключ
- Вопросы проектирования:
  - В какой форме осуществляется ссылка на элементы?

Ассоциативные массивы в языке Perl

hashes

- Имена начинаются с %; литералы заключаются в скобки
  `%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" => 65, ...);`
- При обращении к элементу индекс-ключ указывается в фигурных скобках
  `$hi_temps{"Wed"} = 83;`
- Элементы можно удалить из массива при помощи оператора `delete`
  `delete $hi_temps{"Tue"};`
- Удалить все элементы из массива
  `%hi_temps = ();`

Массивы и ассоциативные массивы

Массив : когда требуется обработать каждый элемент

Ассоциативный массив : когда нужно найти элемент в массиве

Записи и массивы

- Прозрачная и безопасная структура
- Записи используются для разнородных наборов данных
- Доступ к элементам массивов гораздо медленнее доступа к полям записей, так как индексы вычисляются динамически (поля записей — статичны)
- Можно было бы использовать динамические индексы для доступа к полям записей, но это исключило бы проверку типов и было бы гораздо медленнее

Указатели и ссылочные типы

- Область значений переменной типа указатель состоит из адресов в памяти и специального значения — nil
- Дают возможность косвенной адресации
- Дают возможность динамического управления памятью
- Указатель можно использовать для доступа к месту, где осуществляется динамическое выделение памяти (обычно это место называют кучей)

Вопросы проектирования указателей

- Каковы область видимости и время жизни переменной-указателя?
- Каково время жизни динамической переменной?
- Имеются ли ограничения относительно типов, на которые могут указывать указатели?
- Используются ли указатели для динамического управления памятью, косвенной адресации или и того, и другого?
- Должен ли язык поддерживать типы-указатели, ссылочные типы или и то, и другое?

Операции над указателями

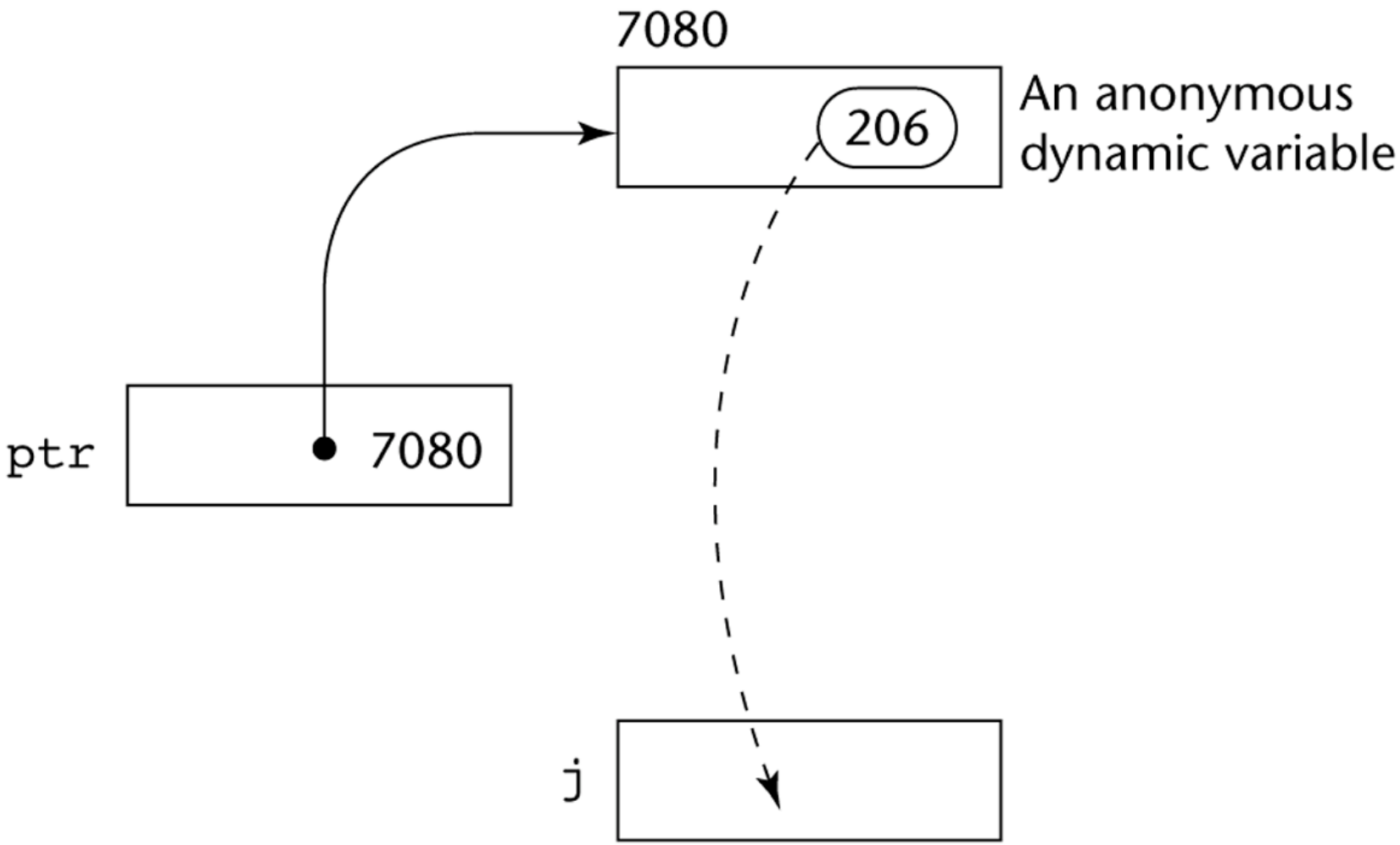
- Две основные операции: присваивание и разыменование
- Присваивание устанавливает значение переменной-указателя равным некоторому осмысленному адресу
- Разыменование возвращает значение, находящееся по адресу, являющемуся значением указателя
  - Разыменование может быть явным или неявным
  - В C++ используется явная операция разыменования, задаваемая при помощи *

Пример (Присвоить `j` значение, хранящееся по адресу, на которое указывает `ptr`)

```
j = *ptr
```

Разыменование указателя

Пример (j = *ptr)



Проблемы, связанные с указателями

- «Висячие» указатели (опасно)
  - Указатель указывает на динамическую переменную, которая была удалена из памяти
- «Потерянные» динамические переменные
  - Размещенная в памяти динамическая переменная, более недоступная программе (мусор)

Пример

- Указатель `p1` указывает на только что созданную динамическую переменную
- Затем указатель `p1` «переводят» на другую только что созданную динамическую переменную

Указатели в языке Ada

- Снижена вероятность появления висячих указателей, так как динамические объекты могут быть автоматически удалены из памяти в конце области видимости типа указателя
- Проблема потерянных динамических переменных присутствует в языке Ada

Указатели в языках C и C++

- Чрезвычайно гибкие, но требуют осторожности при использовании
- Указатели могут указывать на любую переменную независимо от того, когда она была размещена в памяти
- Используются для динамического управления памятью и адресации
- Поддерживается арифметика указателей
- Операторы явного разыменования и взятия адреса
- Не требуется, чтобы тип был фиксирован (`void*`)
- `void*` может указывать на любой тип, и на него распространяется проверка типов (не может быть разыменован)