Parallel array

In computing, a group of **parallel arrays** (also known as structure of arrays or SoA) is a form of implicit data structure that uses multiple arrays to represent a singular array of records. It keeps a separate, homogeneous data array for each field of the record, each having the same number of elements. Then, objects located at the same index in each array are implicitly the fields of a single record. Pointers from one object to another are replaced by array indices. This contrasts with the normal approach of storing all fields of each record together in memory (also known as array of structures or AoS). For example, one might declare an array of 100 names, each a string, and 100 ages, each an integer, associating each name with the age that has the same index.

Contents

Examples

An example in C using parallel arrays:

```c
int  ages[]   = {0,          17,        2,          52,         25};
char *names[] = {"None",     "Mike",    "Billy",    "Tom",      "Stan"};
int  parent[] = {0 /*None*/, 3 /*Tom*/, 1 /*Mike*/, 0 /*None*/, 3 /*Tom*/};

for (i = 1; i <= 4; i++) {
    printf("Name: %s, Age: %d, Parent: %s \n",
           names[i], ages[i], names[parent[i]]);
}
```

in Perl (using a hash of arrays to hold references to each array):

```perl
my %data = (
    first_name   => ['Joe',  'Bob',  'Frank',  'Hans'    ],
    last_name    => ['Smith','Seger','Sinatra','Schultze'],
    height_in_cm => [169,     158,    201,      199      ]);

for $i (0..$#{$data{first_name}}) {
    printf "Name: %s %s\n", $data{first_name}[$i], $data{last_name}[$i];
    printf "Height in CM: %i\n", $data{height_in_cm}[$i];
}
```

Or, in Python:

```python
first_names   = ["Joe",  "Bob",  "Frank",  "Hans"    ]
last_names    = ["Smith","Seger","Sinatra","Schultze"]
heights_in_cm = [169,     158,    201,      199      ]

for i in range(len(first_names)):
    print("Name: %s %s" % (first_names[i], last_names[i]))
    print("Height in cm: %s" % heights_in_cm[i])
```

```
# Using zip:
for first_name, last_name, height_in_cm in zip(first_names, last_names, heights_in_cm):
    print(f"Name: {first_name} {last_name}")
    print(f"Height in cm: {height_in_cm}")
```

Pros and cons

Parallel arrays have a number of practical advantages over the normal approach:

- They can be used in languages which support only arrays of primitive types and not of records (or perhaps don't support records at all).
- Parallel arrays are simple to understand, particularly for beginners who may not fully understand records.
- They can save a substantial amount of space in some cases by avoiding alignment issues. For example, some architectures work best if 4-byte integers are always stored beginning at memory locations that are multiple of 4. If the previous field was a single byte, 3 bytes might be wasted. Many modern compilers can automatically avoid such problems, though in the past some programmers would explictly declare fields in order of decreasing alignment restrictions.
- If the number of items is small, array indices can occupy significantly less space than full pointers, particularly on some architectures.
- Sequentially examining a single field of each record in the array is very fast on modern machines, since this amounts to a linear traversal of a single array, exhibiting ideal locality of reference and cache behaviour.
- They may allow efficient processing with SIMD instructions in certain instruction set architectures

Several of these advantage depend strongly on the particular programming language and implementation in use.

However, parallel arrays also have several strong disadvantages, which serves to explain why they are not generally preferred:

- They have significantly worse locality of reference when visiting the records non-sequentially and examining multiple fields of each record, because the various arrays may be stored arbitrarily far apart.
- They obscure the relationship between fields of a single record (e.g. no type information relates the index between them, an index may be used erroneously).
- They have little direct language support (the language and its syntax typically express no relationship between the arrays in the parallel array, and cannot catch errors).
- Since the bundle of fields is not a "thing", passing it around it tedious and error-prone. For example, rather than calling a function to do something to one record (or structure or object), the function must take the fields as separate arguments. When a new field is added or changed, many parameter lists must change, where passing objects as whole would avoid such changes entirely.
- They are expensive to grow or shrink, since each of several arrays must be reallocated. Multi-level arrays can ameliorate this problem, but impacts performance due to the additional indirection needed to find the desired elements.
- Perhaps worst of all, they greatly raise the possibility of errors. Any insertion, deletion, or move must always be applied consistently to all of the arrays, or the arrays will no longer be synchronized with each other, leading to bizarre outcomes.

The bad locality of reference can be alleviated in some cases: if a structure can be divided into groups of fields that are generally accessed together, an array can be constructed for each group, and its elements are records containing only these subsets of the larger structure's fields. (see data oriented design). This is a valuable way of speeding up access to very large structures with many members, while keeping the portions of the structure tied together. An alternative to tying them together using array indexes is to use references to tie the portions together, but this can be less efficient in time and space.

Another alternative is to use a single array, where each entry is a record structure. Many language provide a way to declare actual records, and arrays of them. In other languages it may be feasible to simulate this by declaring an array of n*m size, where m is the size of all the fields together, packing the fields into what is effectively a record, even though the particular language lacks direct support for records. Some compiler optimizations, particularly for vector processors, are able to perform this transformation automatically when arrays of structures are created in the program.

See also

- An example in the linked list article
- Column-oriented DBMS

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Page 209 of section 10.3: Implementing pointers and objects.
- Skeet, Jon (3 June 2014). "Anti-pattern: parallel collections" (http://codeblog.jonskeet.uk/2014/06/03/anti-pattern-parallel-collections/). Retrieved 28 October 2014.