a memory chunk is allocated for `m` integers. If possible, `j` will refer to the same memory chunk as `i` (but there is no guarantee, of course).

The memory management functions implement C++ semantics: `alloc()` calls the default constructor for each element allocated; `realloc()` calls the destructor, default constructor, or copy constructor (depending on whether the memory area must shrink or grow); `free()` calls the destructor for each element freed.

Space. Space-allocated memory (see Space-memory management) is managed per each individual space. All space-allocated memory is returned to the operating system if a space is deleted. Freeing memory (via the `free()` operation) enables the memory to be reused by later `alloc()` operations.

Spaces manage several blocks of memory allocated from the operating system, the block sizes are automatically chosen based on the recent memory allocations of a space (and, if a space has been created by cloning, the memory allocation history of the original space). The exact policies can be configured, see the namespace `MemoryConfig`.

Memory chunks allocated from the operating system for spaces are cached among all spaces for a single thread. This cache for the current thread can be flushed by invoking the `flush()` member function of `Space`.

Variable implementations, propagators, branchers, view arrays, for example, are allocated from their `home` space. An important characteristic is that all these data structures have fixed size throughout their lifetimes or even shrink. As space-allocated memory is managed for later reusal if it is freed, frequent allocation/reallocation/deallocation leads to highly fragmented memory with little chance of reusal. Hence, space-allocated memory is *absolutely unsuited* for data structures that require frequent allocation/deallocation and/or resizing. For these data structures it is better to use the heap or freelists, if possible. See Section 29.2 for more information.

Note that space-allocated memory is available with allocators compatible with the C++ STL, see Using allocators with Gecode.

Region. All spaces for one thread share a chunk of memory for temporary data structures with very efficient allocation and deallocation (again, its exact size is defined in the namespace `MemoryConfig`). Assume that `home` refers to a space (say, during the execution of the `propagate()` function of a propagator), then

```
Region r(home);
```

creates a new region `r` for memory management (see Region memory management). A region does not impose any size limit on the memory blocks allocated from it. If necessary, a region transparently falls back to heap-allocated memory.

Several regions for the same space can exist simultaneously. For example,

```
Region r1(home);
int* i = r1.alloc<int>(n);
{
  Region r2(home);
  int* j = r2.alloc<int>(m);
}
```

However, it is essential that regions are used in a stack fashion: while region `r2` is in use, no allocation from any previous region (`r1` in our example) is allowed. That is, the following code will result in a crash (if we are lucky):

```
Region r1(home), r2(home);
int* j = r2.alloc<int>(m);
int* i = r1.alloc<int>(n);
```

The specialty of a region is that it does not require `free()` operations. If a region is destructed, all memory allocated from it is freed (unless several regions for the same space exist). If several regions for the same space are in use, the memory is entirely freed when all regions are destructed.

Even though a region does not require `free()` operations, it can profit from it: if the memory allocated last is freed first, the freed memory becomes immediately available for future allocation.

Tip 29.1 (Keep the scope of a region small)**.** In order to make best use of the memory provided by a region it is good practice to keep the scope of regions as small as possible. For example, suppose that in the following example

```
Region r(home);
{
  int* i = r.alloc<int>(n);
  ...
}
{
  int* i = r.alloc<int>(n);
  ...
}
```

the memory allocated for `i` is not used outside the two blocks. Then it is in fact better to rewrite the code to:

```
{
  Region r(home);
  int* i = r.alloc<int>(n);
  ...
}
{
  Region r(home);
  int* i = r.alloc<int>(n);
  ...
}
```

Then both blocks will have access to the full memory of the region. Furthermore, the creation of a region is very efficient and the code might even benefit from better cache behavior. ◀

Heap. The `heap` (a global variable, see Heap memory management) is nothing but a C++-wrapper around `malloc()` and `free()` as provided by the underlying operating system. In case memory is exhausted, an exception of type `MemoryExhausted` is thrown.

Space-allocated freelists. Freelists are allocated from space memory. Any object to be managed by a freelist must inherit from the class `FreeList` which already defines a pointer to a next freelist element. The sizes for freelist objects are quite constrained, check the values `fl_size_min` and `fl_size_max` as defined in the namespace `MemoryConfig`.

Allocation and deallocation is available through the member functions `fl_alloc()` and `fl_dispose()` of the class `Space`.

29.2 Managing propagator state

Many propagators require sophisticated data structures to perform propagation. The data structures are typically kept between successive executions of a propagator. There are two main issues for these data structures: *where* to allocate them and *when* to allocate them.

Where to allocate. Typically, the data structures used by a propagator are of dynamic size and hence cannot be stored in a simple member of the propagator. This means that the propagator is free to allocate the memory from either its own space or from the heap. Allocation from the heap could also mean to use other operations to allocate and free memory, such as `malloc()` and `free()` provided by the operating system or **`new`** and **`delete`** provided by C++.

In case the data structure does not change its size often, it is best to allocate from a space: allocation is more efficient and deallocation is automatic when the space is deleted.

In case the data structure requires frequent reallocation operations, it is better to allocate from the heap. Then, the memory will not be automatically freed when a space is deleted. The memory must be freed by the propagator's `dispose()` function. Furthermore,