Heap Management

The heap is the **portion of the store** that is used for data that **lives indefinitely**, or **until the program explicitly deletes it**. While local variables typically become inaccessible when their procedures end, many languages enable us to create objects or other data whose existence is not tied to the procedure activation that creates them. For example, both **C++ and Java** give the programmer **new** to create objects that may be passed from procedure to procedure, so they continue to exist long after the procedure that created them is gone. **Such objects are stored on a heap**.

Allocating memory

There are **two ways** that memory gets allocated for data storage:

1. **Compile Time (or static) Allocation**
   - Memory for named variables is allocated by the compiler
   - Exact size and type of storage must be known at compile time
   - For standard array declarations, this is why the size has to be constant
2. **Dynamic Memory Allocation**
   - Memory allocated "on the fly" during run time
   - dynamically allocated space usually placed in a program segment known as the **heap** or the **free store**
   - **Exact amount of space or number of items** does not have to be known by the compiler in advance.
   - For dynamic memory allocation, **pointers are crucial**

The Memory Manager

The memory manager keeps track of all the free space in heap storage at all times. It **performs two basic functions**:

- **Allocation**. When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size. **If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available**, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.
- **Deallocation**. The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests. Memory

managers typically do not return memory to the operating system, even if the program's heap usage drops.

Dynamic Memory Allocation

We can dynamically allocate storage space **while the program is running**, but we cannot create new variable names "on the fly". For this reason, **dynamic allocation requires two steps**:

1. **Creating the dynamic space.**
2. **Storing its address in a pointer (so that the space can be accessed)**

To dynamically allocate memory in **C++**, we use the **new operator**.

Deallocation:

- Deallocation is the "**clean-up**" of space being used for variables or other data storage
- Compile time variables are automatically **deallocated** based on their known extent (this is the same as scope for "automatic" variables)
- It is the **programmer's job to deallocates dynamically created space**
- To deallocates dynamic memory, we use the **delete** operator

Implementing dynamic allocation

Where does memory for a program's heap come from?

- The program makes calls to the OS (system calls) to add memory to the heap of a running process (move the top of heap pointer)
  - On Linux, this is void *sbrk(int **incr**)
    - adds at least **incr** bytes to the end of the program's data segment
  - The key idea is to get **big chunks** of memory from the OS and then hand out small regions to the program on demand
    - OS calls can be a hundred times slower than local subroutines
- **How is the memory in the heap managed?**
  - Data structures need to be kept to:
    - Keep track of what memory is in use, and
    - Keep track of where the free memory regions are
  - Calls to **malloc()**, **free()**, etc., update these data structures

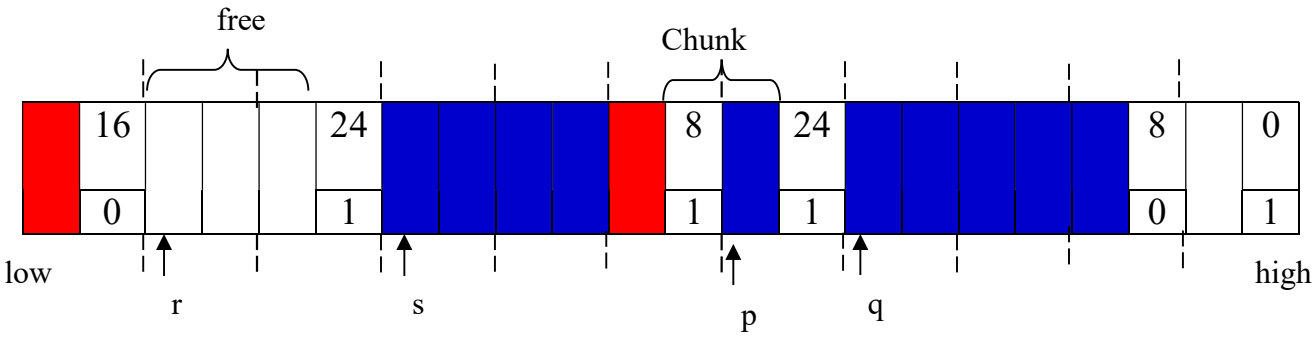
A simple heap structure

1. Divide the heap into blocks
2. **heap blocks:**

– header (32 bits)
– allocated block (multiple of 8, double-word aligned)
– optional padding for alignment or other reasons

3. **4-byte header:**

– 29 bits for size of block (why 29?)
– 1 bit to indicate free/allocated
– 2 bits available for other purposes; we won't use them
– size is size of heap block, not allocated block
– end of heap marked by **header w/ size = 0**



- Each cell above is 4 bytes (e.g. 4 byte chunks on a 32-bit system)
- Headers: size and used flag (**1** marked in use and **0** indicates it is free)
- Blue areas: allocated blocks
- Red areas: wasted space due to alignment
- How can we find the address of the next block?
- What kinds of things could be stored at the areas pointed to by the pointers?

Above could be generated by this sequence:

r = malloc (8); (**malloc** find usable block on free list)
s = malloc(16);
p = malloc(4); p ()
q = malloc(20);
free(r);

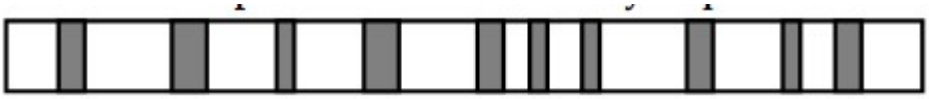
Managing and Coalescing Free Space

When an object is deallocated manually, the memory manager must make its **chunk free**, so it can be allocated again. In some circumstances, it may also be possible to **combine (coalesce) that chunk with adjacent chunks** of the heap, to form a larger chunk. There is an **advantage** to doing so, since we can always use a large chunk to do the work of small chunks of equal total size, but many small chunks cannot hold one large object, as the combined chunk could.
If we keep a bin for chunks of one fixed size, as Lea does for small sizes, then we may prefer not to coalesce adjacent blocks of that size into a chunk of double the size. It is simpler to keep all the chunks of one size in as many pages as we need, and never coalesce them. Then, a simple allocation/deallocation scheme is to keep a **bitmap**, with one bit for each chunk in the bin. A **1** indicates the chunk is **occupied**; **0** indicates it is **free**. When a chunk is deallocated, we change its **1** to a **0**. When we need to allocate a chunk, we find any chunk with a 0 bit, change that bit to a **1**, and use the corresponding chunk. If there are no free chunks, we get a new page, divide it into chunks of the appropriate size, and extend the bit vector.

Reducing Fragmentation

At the beginning of program execution, the heap is one contiguous unit of free space. As the program allocates and deallocates memory, this space is **broken up into free and used chunks of memory**, and the free chunks need not reside in a contiguous area of the heap. We refer to the **free chunks of memory as holes**. With each allocation request, the memory manager must place the requested chunk of memory into a **large-enough hole**. Unless a hole of exactly the right size is found, we need to split some hole, creating a yet **smaller hole**.
With each deallocation request, the freed chunks of memory are added back to the pool of free space. We coalesce contiguous holes into larger holes, as the holes can only get smaller otherwise. **If we are not careful, the memory may end up getting fragmented, consisting of large numbers of small, noncontiguous holes**. It is then possible that no hole is large enough to satisfy a future request, even though there may be sufficient aggregate free space.

Problem: free creates holes (fragmentations) results, lots of free space but cannot satisfy request



Best-Fit and Next-Fit Object Placement

We reduce fragmentation by controlling how the memory manager places new objects in the heap. **It has been found empirically that a good strategy for minimizing fragmentation for real-life programs is to allocate the requested memory in the smallest available hole** that is large enough. This **best-fit algorithm** tends to spare the large holes to satisfy subsequent, larger requests. An alternative, **called first-fit**, where an object is placed in the first (**lowest-address**) hole in which it fits, takes less time to place objects, but has been found inferior to best-fit in overall performance.
To implement **best-fit placement** more efficiently, we can **separate free space into bins**, according to their **sizes**. One practical idea is to have many more bins for the smaller sizes, because there are usually many more small objects.

For sizes that do not have a private bin, we find the one bin that is allowed to include chunks of the desired size. Within that bin, we can use either a **first-fit or a best-fit strategy**; i.e., we **either look for and select the first** chunk that is sufficiently large or, we spend more time and find the smallest chunk that is sufficiently large. Note that when the fit is not exact, the remainder of the chunk will generally need to be placed in a bin with smaller sizes.

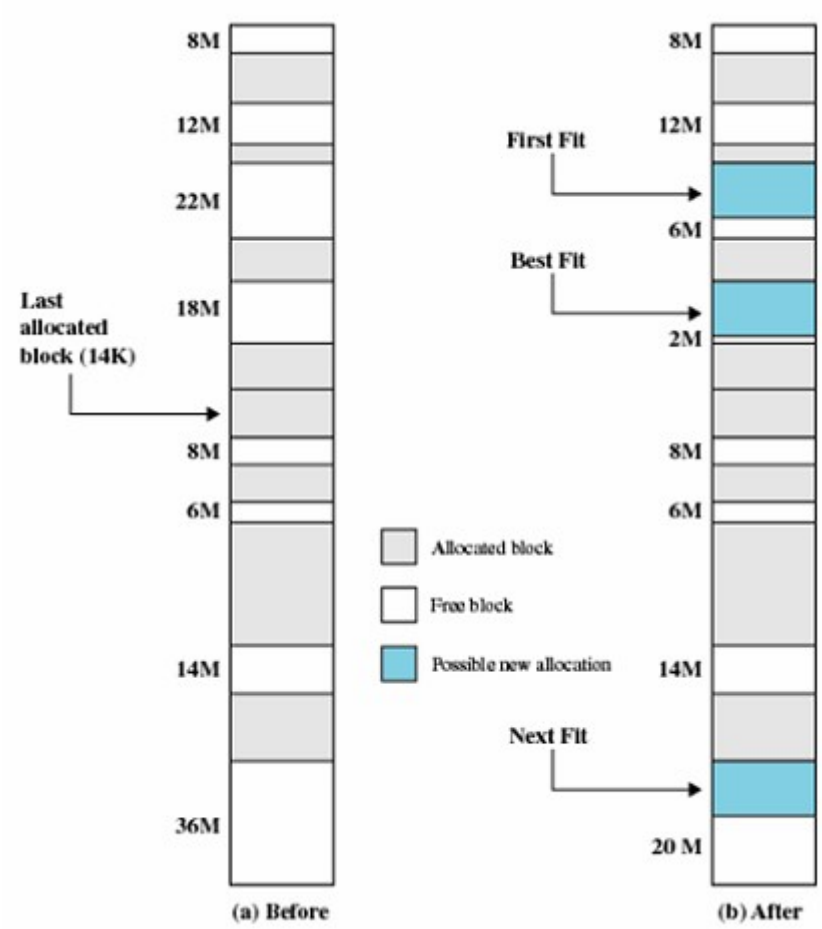
While **best-fit** placement tends to **improve space utilization**, it may not be the best in terms of spatial locality. Chunks allocated at about the same time by a program tend to have similar reference patterns and to have similar lifetimes. Placing them close together thus improves the program's spatial locality. One **useful adaptation of the best-fit algorithm** is to modify the placement in the case when a chunk of the exact requested size cannot be found. In this case, we use a **next-fit strategy**, **trying to allocate the object in the chunk that has last been split**, whenever enough space for the new object remains in that chunk. **Next-fit also tends to improve the speed of the allocation operation.**

We can summaries the placement algorithms → selecting among free blocks of main memory as follows:

- **Best-Fit: Closest in size to the request**
- **First-Fit**: Scans the main memory from the beginning and first available block that is large enough
- **Next-Fit**: Scans the memory from the location of last placement and chooses next available block that is large enough

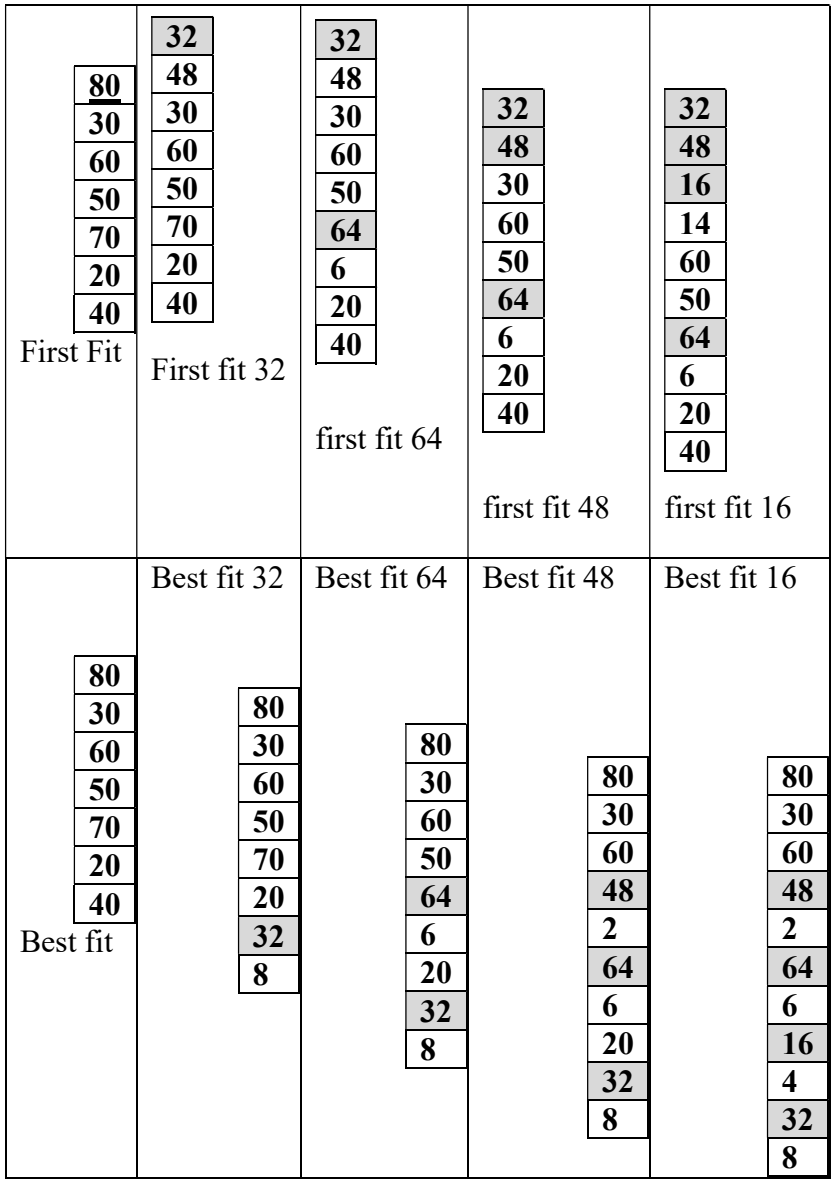
Compaction is time consuming → OS must be clever in plugging holes while assigning processes to memory

Allocation of **16 MB** block using three placement algorithms



Example

Suppose the heap consists of **seven chunks**. The sizes of the chunks, in order, are **80**, **30, 60, 50, 70, 20, 40 bytes**. If your request space for **objects of the following sizes: 32, 64, 48, 16, in that order**, what does the free space list look like after satisfying the requests, if the method of selecting chunks is (a) First fit, b) Best fit).



If the heap look like in the figure below

	Allocated block
	Free block
	Possible new allocation