Dynamic and static memory allocation

In computer science, **dynamic memory allocation** is the allocation of memory storage for use in a computer program during the runtime of that program. It can be seen also as a way of distributing ownership of limited memory resources among many pieces of data and code.

Dynamically allocated memory exists until it is released either explicitly by the programmer, exiting a block, or by the garbage collector. This is in contrast to static memory allocation, which has a fixed duration. It is said that an object so allocated has a *dynamic lifetime*.

Usually, memory is allocated from a large pool of unused memory area called **the heap** (also called the **free store**). Since the precise location of the allocation is not known in advance, the memory is accessed indirectly, usually via a reference. The precise algorithm used to organize the memory area and allocate and deallocate chunks is hidden behind an abstract interface and may use any of the methods described below.

Static memory allocation refers to the process of allocating memory at compile-time before the associated program is executed, unlike dynamic memory allocation where memory is allocated as required at run-time.

An application of this technique involves a program module (e.g. function or subroutine) declaring static data locally, such that these data are inaccessible in other modules unless references to it are passed as parameters or returned. A single copy of static data is retained and accessible through many calls to the function in which it is declared. Static memory allocation therefore has the advantage of modularizing data within a program design in the situation where these data must be retained through the runtime of the program.

The use of static variables within a class in object oriented programming enables a single copy of such data to be shared between all the objects of that class.

Object constants known at compile-time, like string literals, are usually allocated statically. In object-oriented programming, the virtual method tables of classes are usually allocated statically.

Stacks in computing architectures are regions of memory where data is added or removed in a Last-In-First-Out manner.

In most modern computer systems, each thread has a reserved region of memory referred to as its stack. When a function executes, it may add some of its state data to the top of the stack; when the function exits it is responsible for removing that data from the stack. If a region of memory lies on the thread's stack, that memory is said to have been allocated on the stack.

Because the data is added and removed in a last-in-first-out manner, stack allocation is very simple and typically faster than heap allocation. Another advantage is that memory on the stack is automatically reclaimed when the function exits, which can be convenient for the programmer.

A disadvantage of stackbased memory allocation is that a thread's stack size can be as small as a few dozen kilobytes. Allocating more memory on the stack than is available can result in a crash due to stack overflow. Another disadvantage is that the memory stored on the stack is automatically deallocated when the function that created it returns, and thus the function must copy the data if they should be available to other parts of the program after it returns.

Some processors families, such as the x86, have special instructions for manipulating the stack of the currently executing thread. Other processor families, including PowerPC and MIPS, do not have explicit stack support, but instead rely on convention and delegate stack management to the operating system's Application Binary Interface (ABI).

In computing, **`malloc`** is a subroutine provided in the C programming language's and C++ programming language's standard library for performing dynamic memory allocation.

Rationale

The C programming language manages memory either *statically* or *automatically*. Static-duration variables are allocated in main (fixed) memory and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and, until C99 which allows variable-sized arrays, automatic-duration variables, the size of the allocation must be a compile-time constant. If the required size will not be known until run-time — for example, if data of arbitrary size is being read from the user or from a disk file — using fixed-size data objects is inadequate. Some platforms provide the `alloca` function,[1] which allows run-time allocation of variable-sized automatic variables on the stack. C99 supports variable-length arrays of block scope having sizes determined at runtime.

The lifetime of allocated memory is also a concern. Neither static- nor automatic-duration memory is adequate for all situations. Stack-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly but more flexibly managed, typically by allocating it from a *heap*, an area of memory structured for this purpose. In C, one uses the library function `malloc` to allocate a block of memory on the heap. The program accesses this block of memory via a pointer which `malloc` returns. When the memory is no longer needed, the pointer is passed to `free` which deallocates the memory so that it can be used for other purposes.

Dynamic memory allocation in C

The `malloc` function is the basic function used to allocate memory on the heap in C. Its function prototype is

```
void *malloc(size_t size);
```

which allocates *`size`* bytes of memory. If the allocation succeeds, a pointer to the block of memory is returned.

`malloc` returns a void pointer (`void *`), which indicates that it is a pointer to a region of unknown data type. Note that because `malloc` returns a void pointer, it needn't be explicitly cast to a more specific pointer type: ANSI C defines an implicit coercion between the void pointer type and other pointer types. An explicit cast of `malloc`'s return value is sometimes performed because `malloc` originally returned a `char *`, but this cast is unnecessary in modern C code.[2][3] However, omitting the cast creates an incompatibility with C++, which requires it.

Memory allocated via `malloc` is persistent: it will continue to exist until the program terminates or the memory is explicitly deallocated by the programmer (that is, the block is said to be "freed"). This is achieved by use of the `free` function. Its prototype is

```
void free(void *pointer);
```

which releases the block of memory pointed to by `pointer`. `pointer` must have been previously returned by `malloc` or `calloc` (or a function which uses one of these, eg `strdup`), and must only be passed to `free` once.

Usage example

The standard method of creating an array of ten integers on the stack is:

```
int array[10];
```

To allocate a similar array dynamically, the following code could be used:

```
/* Allocate space for an array with ten elements of type int. */
int *ptr = malloc(10 * sizeof (int));
if (ptr == NULL)
    {
    /* Memory could not be allocated, the program should handle the error here
as appropriate. */
    }
else
    {
    /* If ptr is not NULL, allocation succeeded. */
    }
```

`malloc` returns a null pointer (`NULL`) to indicate that no memory is available, or that some other error occurred which prevented memory being allocated.

The improper use of `malloc` and related functions can frequently be a source of bugs.

Allocation failure

`malloc` is not guaranteed to succeed - if there is no memory available, or if the program has exceeded the amount of memory it is allowed to reference, `malloc` will return a `NULL` pointer. Depending on the nature of the underlying environment, this may or may not be a likely occurrence. Many programs do not check for `malloc` failure. Such a program would attempt to use the `NULL` pointer returned by `malloc` as if it pointed to allocated memory, and the program would crash. This has traditionally been considered an incorrect design, although it remains common, as memory allocation failures only occur rarely in most situations, and the program frequently can do nothing better than to exit anyway. Checking for allocation failure is more important when implementing libraries - since the library might be used in low-memory environments, it is usually considered good practice to return memory allocation failures to the program using the library and allow it to choose whether to attempt to handle the error.

Memory leaks

When a call to `malloc`, `calloc` or `realloc` succeeds, the return value of the call should eventually be passed to the `free` function. This releases the allocated memory, allowing it to be reused to satisfy other memory allocation requests. If this is not done, the allocated memory will not be released until the process exits — in other words, a memory leak will occur. Typically, memory leaks are caused by losing track of pointers, for example not using a temporary pointer for the return value of `realloc`, which may lead to the original pointer being overwritten with NULL, for example:

```
void *ptr;
size_t size = BUFSIZ;

ptr = malloc(size);

/* some further execution happens here... */

/* now the buffer size needs to be doubled */
if (size > SIZE_MAX / 2) {
  /* handle overflow error */
  /* ... */
  return (1);
}
size *= 2;
ptr = realloc(ptr, size);
if (ptr == NULL) {
 /* the realloc failed (it returned NULL), but the original address in ptr has been lost
      so the memory cannot be freed and a leak has occurred */
  /* ... */
  return (1);
}
/* ... */
```

Use after free

After a pointer has been passed to `free`, it becomes a dangling pointer: it references a region of memory with undefined content, which may not be available for use. However, nothing prevents the pointer from being used. For example:

```
int *ptr = malloc(sizeof (int));
free(ptr);
*ptr = 0; /* Undefined behavior! */
```

Code like this has undefined behavior: its effect may vary. Commonly, the system may have reused freed memory for other purposes. Therefore, writing through a pointer to a deallocated region of memory may result in overwriting another piece of data somewhere else in the program. Depending on what data is overwritten, this may result in data corruption or cause the program to crash at a later time. A particularly bad example of this problem is if the same pointer is passed to `free` twice, known as a *double free*. To avoid this, some programmers set pointers to `NULL` after passing them to free: free(NULL) is safe (it does nothing).[4] However, this will not protect other aliases to the same pointer from being doubly freed.

Freeing unallocated memory

Another problem is when free is passed an address that wasn't allocated by malloc. This can be caused when a pointer to a literal string or the name of a declared array is passed to free, for example:

```
char *msg = "Default message";
int tbl[100];
```

passing either of the above pointers to free will result in undefined behaviour. Passing the NULL pointer to free is safe, and does nothing.

Implementations

The implementation of memory management depends greatly upon operating system and architecture. Some operating systems supply an allocator for malloc, while others supply functions to control certain regions of data.

The same dynamic memory allocator is often used to implement both malloc and operator new in C++. Hence, we will call this the **allocator** rather than malloc. (However, note that it is never proper for a C++ program to treat malloc and new interchangeably. For example, free cannot be used to release memory that was allocated with new.[5])

Allocation size limits

The largest possible memory block malloc can allocate depends on the host system, particularly the size of physical memory and the operating system implementation. Theoretically, the largest number should be the maximum value that can be held in a *size_t* type, which is an implementation-dependent unsigned integer representing the size of an area of memory. The maximum value is (size_t) -1, or the constant SIZE_MAX in the C99 standard. The C standards guarantee that a certain minimum (0x7FFF in C90, 0xFFFF in C99) for at least one object can be allocated.