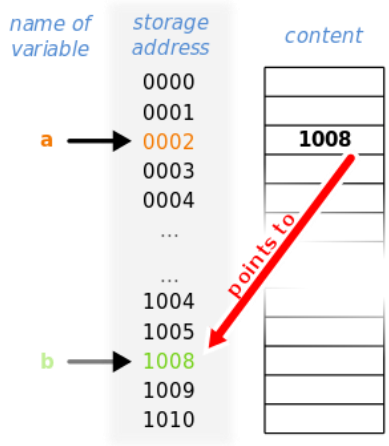
Pointer (computer programming)

In computer science, a **pointer** is an object in many programming languages that stores a memory address. This can be that of another value located in computer memory, or in some cases, that of memory-mapped computer hardware. A pointer *references* a location in memory, and obtaining the value stored at that location is known as *dereferencing* the pointer. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number and reading the text found on that page. The actual format and content of a pointer variable is dependent on the underlying computer architecture.

> I do consider assignment statements and pointer variables to be among computer science's "most valuable treasures."
>
> Donald Knuth, *Structured Programming, with go to Statements*[1]

Using pointers significantly improves performance for repetitive operations, like traversing iterable data structures (e.g. strings, lookup tables, control tables and tree structures). In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.



Pointer **a** pointing to the memory address associated with variable **b**. In this diagram, the computing architecture uses the same address space and data primitive for both pointers and non-pointers; this need should not be the case.

Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs). In object-oriented programming, pointers to functions are used for binding methods, often using virtual method tables.

A pointer is a simple, more concrete implementation of the more abstract *reference* data type. Several languages, especially low-level languages, support some type of pointer, although some have more restrictions on their use than others. While "pointer" has been used to refer to references in general, it more properly applies to data structures whose interface explicitly allows the pointer to be manipulated (arithmetically via *pointer arithmetic*) as a memory address, as opposed to a magic cookie or capability which does not allow such. Because pointers allow both protected and unprotected access to memory addresses, there are risks associated with using them, particularly in the latter case. Primitive pointers are often stored in a format similar to an integer; however, attempting to dereference or "look up" such a pointer whose value is not a valid memory address could cause a program to crash (or contain invalid data). To alleviate this potential problem, as a matter of type safety, pointers are considered a separate type parameterized by the type of data they point to, even if the underlying representation is an integer.

Other measures may also be taken (such as validation & bounds checking), to verify that the pointer variable contains a value that is both a valid memory address and within the numerical range that the processor is capable of addressing.

Formal description

In computer science, a pointer is a kind of reference.

A *data primitive* (or just *primitive*) is any datum that can be read from or written to computer memory using one memory access (for instance, both a *byte* and a *word* are primitives).

A *data aggregate* (or just *aggregate*) is a group of primitives that are logically contiguous in memory and that are viewed collectively as one datum (for instance, an aggregate could be 3 logically contiguous bytes, the values of which represent the 3 coordinates of a point in space). When an aggregate is entirely composed of the same type of primitive, the aggregate may be called an *array*; in a sense, a multi-byte *word* primitive is an array of bytes, and some programs use words in this way.

In the context of these definitions, a *byte* is the smallest primitive; each memory address specifies a different byte. The memory address of the initial byte of a datum is considered the memory address (or *base memory address*) of the entire datum.

A *memory pointer* (or just *pointer*) is a primitive, the value of which is intended to be used as a memory address; it is said that *a pointer points to a memory address*. It is also said that *a pointer points to a datum [in memory]* when the pointer's value is the datum's memory address.

More generally, a pointer is a kind of reference, and it is said that *a pointer references a datum stored somewhere in memory*; to obtain that datum is *to dereference the pointer*. The feature that separates pointers from other kinds of reference is that a pointer's value is meant to be interpreted as a memory address, which is a rather low-level concept.

References serve as a level of indirection: A pointer's value determines which memory address (that is, which datum) is to be used in a calculation. Because indirection is a fundamental aspect of algorithms, pointers are often expressed as a fundamental data type in programming languages; in statically (or strongly) typed programming languages, the type of a pointer determines the type of the datum to which the pointer points.

Uses

Pointers are directly supported without restrictions in languages such as PL/I, C, C++, Pascal, FreeBASIC, and implicitly in most assembly languages. They are primarily used for constructing references, which in turn are fundamental to constructing nearly all data structures, as well as in passing data between different parts of a program.

In functional programming languages that rely heavily on lists, data references are managed abstractly by using primitive constructs like cons and the corresponding elements car and cdr, which can be thought of as specialised pointers to the first and second components of a cons-cell. This gives rise to some of the idiomatic "flavour" of functional programming. By structuring data in such cons-lists, these languages facilitate recursive means for building and processing data—for example, by recursively accessing the head and tail elements of lists of lists; e.g. "taking the car of

the cdr of the cdr". By contrast, memory management based on pointer dereferencing in some approximation of an array of memory addresses facilitates treating variables as slots into which data can be assigned imperatively.

When dealing with arrays, the critical lookup operation typically involves a stage called *address calculation* which involves constructing a pointer to the desired data element in the array. In other data structures, such as linked lists, pointers are used as references to explicitly tie one piece of the structure to another.

Pointers are used to pass parameters by reference. This is useful if the programmer wants a function's modifications to a parameter to be visible to the function's caller. This is also useful for returning multiple values from a function.

Pointers can also be used to allocate and deallocate dynamic variables and arrays in memory. Since a variable will often become redundant after it has served its purpose, it is a waste of memory to keep it, and therefore it is good practice to deallocate it (using the original pointer reference) when it is no longer needed. Failure to do so may result in a *memory leak* (where available free memory gradually, or in severe cases rapidly, diminishes because of an accumulation of numerous redundant memory blocks).

C pointers

The basic syntax to define a pointer is:[4]

```
int *ptr;
```

This declares `ptr` as the identifier of an object of the following type:

- pointer that points to an object of type `int`

This is usually stated more succinctly as "`ptr` is a pointer to `int`."

Because the C language does not specify an implicit initialization for objects of automatic storage duration,[5] care should often be taken to ensure that the address to which `ptr` points is valid; this is why it is sometimes suggested that a pointer be explicitly initialized to the null pointer value, which is traditionally specified in C with the standardized macro `NULL`:[6]

```
int *ptr = NULL;
```

Dereferencing a null pointer in C produces undefined behavior,[7] which could be catastrophic. However, most implementations simply halt execution of the program in question, usually with a segmentation fault.

However, initializing pointers unnecessarily could hinder program analysis, thereby hiding bugs.

In any case, once a pointer has been declared, the next logical step is for it to point at something:

```
int a = 5;
```

```
int *ptr = NULL;

ptr = &a;
```

This assigns the value of the address of `a` to `ptr`. For example, if `a` is stored at memory location of 0x8130 then the value of `ptr` will be 0x8130 after the assignment. To dereference the pointer, an asterisk is used again:

```
*ptr = 8;
```

This means take the contents of `ptr` (which is 0x8130), "locate" that address in memory and set its value to 8. If `a` is later accessed again, its new value will be 8.

This example may be clearer if memory is examined directly. Assume that `a` is located at address 0x8130 in memory and `ptr` at 0x8134; also assume this is a 32-bit machine such that an int is 32-bits wide. The following is what would be in memory after the following code snippet is executed:

```
int a = 5;
int *ptr = NULL;
```

Address	Contents
0x81300x(	0000005
0x81340x(	0000000

(The NULL pointer shown here is 0x00000000.) By assigning the address of `a` to `ptr`:

```
ptr = &a;
```

yields the following memory values:

Address	Contents
0x81300x(	0000005
0x81340x(	0008130

Then by dereferencing `ptr` by coding:

```
*ptr = 8;
```

the computer will take the contents of `ptr` (which is 0x8130), 'locate' that address, and assign 8 to that location yielding the following memory:

Address	Contents
0x81300x(	0000008
0x81340x(	0008130

Clearly, accessing `a` will yield the value of 8 because the previous instruction modified the contents of `a` by way of the pointer `ptr`.

Use in data structures

When setting up data structures like lists, queues and trees, it is necessary to have pointers to help manage how the structure is implemented and controlled. Typical examples of pointers are start pointers, end pointers, and stack pointers. These pointers can either be **absolute** (the actual physical address or a virtual address in virtual memory) or **relative** (an offset from an absolute start address ("base") that typically uses fewer bits than a full address, but will usually require one additional arithmetic operation to resolve).

Relative addresses are a form of manual memory segmentation, and share many of its advantages and disadvantages. A two-byte offset, containing a 16-bit, unsigned integer, can be used to provide relative addressing for up to 64 KiB (2^{16} bytes) of a data structure. This can easily be extended to 128, 256 or 512 KiB if the address pointed to is forced to be aligned on a half-word, word or double-word boundary (but, requiring an additional "shift left" bitwise operation—by 1, 2 or 3 bits—in order to adjust the offset by a factor of 2, 4 or 8, before its addition to the base address). Generally, though, such schemes are a lot of trouble, and for convenience to the programmer absolute addresses (and underlying that, a *flat address space*) is preferred.

A one byte offset, such as the hexadecimal ASCII value of a character (e.g. X'29') can be used to point to an alternative integer value (or index) in an array (e.g., X'01'). In this way, characters can be very efficiently translated from 'raw data' to a usable sequential index and then to an absolute address without a lookup table.

C arrays

In C, array indexing is formally defined in terms of pointer arithmetic; that is, the language specification requires that `array[i]` be equivalent to `*(array + i)`.[8] Thus in C, arrays can be thought of as pointers to consecutive areas of memory (with no gaps),[8] and the syntax for accessing arrays is identical for that which can be used to dereference pointers. For example, an array `array` can be declared and used in the following manner:

```
int array[5];      /* Declares 5 contiguous integers */
int *ptr = array;  /* Arrays can be used as pointers */
ptr[0] = 1;        /* Pointers can be indexed with array syntax */
*(array + 1) = 2;  /* Arrays can be dereferenced with pointer syntax */
*(1 + array) = 2;  /* Pointer addition is commutative */
2[array] = 4;      /* Subscript operator is commutative */
```

This allocates a block of five integers and names the block `array`, which acts as a pointer to the block. Another common use of pointers is to point to dynamically allocated memory from malloc which returns a consecutive block of memory of no less than the requested size that can be used as an array.

While most operators on arrays and pointers are equivalent, the result of the `sizeof` operator differs. In this example, `sizeof(array)` will evaluate to `5*sizeof(int)` (the size of the array), while `sizeof(ptr)` will evaluate to `sizeof(int*)`, the size of the pointer itself.

Default values of an array can be declared like:

```
int array[5] = {2, 4, 3, 1, 5};
```

If `array` is located in memory starting at address 0x1000 on a 32-bit little-endian machine then memory will contain the following (values are in hexadecimal, like the addresses):

	0	1	2	3
10002	0	0	0	
10044	0	0	0	
10083	0	0	0	
100C1	0	0	0	
10105	0	0	0	

Represented here are five integers: 2, 4, 3, 1, and 5. These five integers occupy 32 bits (4 bytes) each with the least-significant byte stored first (this is a little-endian CPU architecture) and are stored consecutively starting at address 0x1000.

The syntax for C with pointers is:

- `array` means 0x1000;
- `array + 1` means 0x1004: the "+ 1" means to add the size of 1 `int`, which is 4 bytes;
- `*array` means to dereference the contents of `array`. Considering the contents as a memory address (0x1000), look up the value at that location (0x0002);
- `array[i]` means element number `i`, 0-based, of `array` which is translated into `*(array + i)`.

The last example is how to access the contents of `array`. Breaking it down:

- `array + i` is the memory location of the (i)th element of `array`, starting at i=0;
- `*(array + i)` takes that memory address and dereferences it to access the value.

C linked list

Below is an example definition of a linked list in C.

```
/* the empty linked list is represented by NULL
 * or some other sentinel value */
#define EMPTY_LIST  NULL

struct link {
    void        *data;  /* data of this link */
    struct link *next;  /* next link; EMPTY_LIST if there is none */
};
```

This pointer-recursive definition is essentially the same as the reference-recursive definition from the Haskell programming language:

```
data Link a = Nil
            | Cons a (Link a)
```

`Nil` is the empty list, and `Cons a (Link a)` is a cons cell of type `a` with another link also of type `a`.

The definition with references, however, is type-checked and does not use potentially confusing signal values. For this reason, data structures in C are usually dealt with via wrapper functions, which are carefully checked for correctness.

Pass-by-address using pointers

Pointers can be used to pass variables by their address, allowing their value to be changed. For example, consider the following C code:

```c
/* a copy of the int n can be changed within the function without affecting the calling code */
void passByValue(int n) {
    n = 12;
}

/* a pointer m is passed instead. No copy of the value pointed to by m is created */
void passByAddress(int *m) {
    *m = 14;
}

int main(void) {
    int x = 3;

    /* pass a copy of x's value as the argument */
    passByValue(x);
    // the value was changed inside the function, but x is still 3 from here on

    /* pass x's address as the argument */
    passByAddress(&x);
    // x was actually changed by the function and is now equal to 14 here

    return 0;
}
```

Dynamic memory allocation

In some programs, the required amount of memory depends on what *the user* may enter. In such cases the programmer needs to allocate memory dynamically. This is done by allocating memory at the *heap* rather than on the *stack*, where variables usually are stored (although variables can also be stored in the CPU registers). Dynamic memory allocation can only be made through pointers, and names – like with common variables – cannot be given.

Pointers are used to store and manage the addresses of dynamically allocated blocks of memory. Such blocks are used to store data objects or arrays of objects. Most structured and object-oriented languages provide an area of memory, called the *heap* or *free store*, from which objects are dynamically allocated.

The example C code below illustrates how structure objects are dynamically allocated and referenced. The standard C library provides the function `malloc()` for allocating memory blocks from the heap. It takes the size of an object to allocate as a parameter and returns a pointer to a

newly allocated block of memory suitable for storing the object, or it returns a null pointer if the allocation failed.

```
/* Parts inventory item */
struct Item {
    int         id;     /* Part number */
    char *      name;   /* Part name   */
    float       cost;   /* Cost        */
};

/* Allocate and initialize a new Item object */
struct Item * make_item(const char *name) {
    struct Item * item;

    /* Allocate a block of memory for a new Item object */
    item = malloc(sizeof(struct Item));
    if (item == NULL)
        returnNULL ;

    /* Initialize the members of the new Item */
    memset(item, 0, sizeof(struct Item));
    item->id =   -1;
    item->name = NULL;
    item->cost = 0.0;

    /* Save a copy of the name in the new Item */
    item->name = malloc(strlen(name) + 1);
    if (item->name == NULL) {
        free(item);
        returnNULL ;
    }
    strcpy(item->name, name);

    /* Return the newly created Item object */
    return item;
}
```

The code below illustrates how memory objects are dynamically deallocated, i.e., returned to the heap or free store. The standard C library provides the function `free()` for deallocating a previously allocated memory block and returning it back to the heap.

```
/* Deallocate an Item object */
void destroy_item(struct Item *item) {
    /* Check for a null object pointer */
    if (item == NULL)
        return;

    /* Deallocate the name string saved within the Item */
    if (item->name != NULL) {
        free(item->name);
        item->name = NULL;
    }

    /* Deallocate the Item object itself */
    free(item);
}
```

Memory-mapped hardware

On some computing architectures, pointers can be used to directly manipulate memory or memory-mapped devices.

Assigning addresses to pointers is an invaluable tool when programming microcontrollers. Below is a simple example declaring a pointer of type int and initialising it to a hexadecimal address in this example the constant 0x7FFF:

```
int *hardware_address = (int *)0x7FFF;
```

In the mid 80s, using the BIOS to access the video capabilities of PCs was slow. Applications that were display-intensive typically used to access CGA video memory directly by casting the hexadecimal constant 0xB8000 to a pointer to an array of 80 unsigned 16-bit int values. Each value consisted of an ASCII code in the low byte, and a colour in the high byte. Thus, to put the letter 'A' at row 5, column 2 in bright white on blue, one would write code like the following:

```
#define VID ((unsigned short (*)[80])0xB8000)

void foo(void) {
    VID[4][1] = 0x1F00 | 'A';
}
```

Use in control tables

Control tables that are used to control program flow usually make extensive use of pointers. The pointers, usually embedded in a table entry, may, for instance, be used to hold the entry points to subroutines to be executed, based on certain conditions defined in the same table entry. The pointers can however be simply indexes to other separate, but associated, tables comprising an array of the actual addresses or the addresses themselves (depending upon the programming language constructs available). They can also be used to point to earlier table entries (as in loop processing) or forward to skip some table entries (as in a switch or "early" exit from a loop). For this latter purpose, the "pointer" may simply be the table entry number itself and can be transformed into an actual address by simple arithmetic.

Typed pointers and casting

In many languages, pointers have the additional restriction that the object they point to has a specific type. For example, a pointer may be declared to point to an integer; the language will then attempt to prevent the programmer from pointing it to objects which are not integers, such as floating-point numbers, eliminating some errors.

For example, in C

```
int *money;
char *bags;
```

`money` would be an integer pointer and `bags` would be a char pointer. The following would yield a compiler warning of "assignment from incompatible pointer type" under GCC

```
bags = money;
```

because `money` and `bags` were declared with different types. To suppress the compiler warning, it must be made explicit that you do indeed wish to make the assignment by typecasting it

```
bags = (char *)money;
```

which says to cast the integer pointer of `money` to a char pointer and assign to `bags`.

A 2005 draft of the C standard requires that casting a pointer derived from one type to one of another type should maintain the alignment correctness for both types (6.3.2.3 Pointers, par. 7):[9]

```
char *external_buffer = "abcdef";
int *internal_data;

internal_data = (int *)external_buffer;  // UNDEFINED BEHAVIOUR if "the resulting pointer
                                         // is not correctly aligned"
```

In languages that allow pointer arithmetic, arithmetic on pointers takes into account the size of the type. For example, adding an integer number to a pointer produces another pointer that points to an address that is higher by that number times the size of the type. This allows us to easily compute the address of elements of an array of a given type, as was shown in the C arrays example above. When a pointer of one type is cast to another type of a different size, the programmer should expect that pointer arithmetic will be calculated differently. In C, for example, if the `money` array starts at 0x2000 and `sizeof(int)` is `4` bytes whereas `sizeof(char)` is `1` byte, then `money + 1` will point to 0x2004, but `bags + 1` would point to 0x2001. Other risks of casting include loss of data when "wide" data is written to "narrow" locations (e.g. `bags[0] = 65537;`), unexpected results when bit-shifting values, and comparison problems, especially with signed vs unsigned values.

Although it is impossible in general to determine at compile-time which casts are safe, some languages store run-time type information which can be used to confirm that these dangerous casts are valid at runtime. Other languages merely accept a conservative approximation of safe casts, or none at all.

Value of pointers

In C and C++, even if two pointers compare as equal that doesn't mean they are equivalent. In these languages *and* LLVM, the rule is interpreted to mean that "just because two pointers point to the same address, does not mean they are equal in the sense that they can be used interchangeably", the difference between the pointers referred to as their *provenance*.[10] Casting to an integer type such as `uintptr_t` is implementation-defined and the comparison it provides does not provide any more insight as to whether the two pointers are interchangeable. In addition, further conversion to bytes and arithmetic will throw off optimizers trying to keep track the use of pointers, a problem still being elucidated in academic research.[11]

Making pointers safer

As a pointer allows a program to attempt to access an object that may not be defined, pointers can be the origin of a variety of programming errors. However, the usefulness of pointers is so great that it can be difficult to perform programming tasks without them. Consequently, many languages

have created constructs designed to provide some of the useful features of pointers without some of their pitfalls, also sometimes referred to as *pointer hazards*. In this context, pointers that directly address memory (as used in this article) are referred to as **raw pointers**, by contrast with smart pointers or other variants.

One major problem with pointers is that as long as they can be directly manipulated as a number, they can be made to point to unused addresses or to data which is being used for other purposes. Many languages, including most functional programming languages and recent imperative languages like Java, replace pointers with a more opaque type of reference, typically referred to as simply a *reference*, which can only be used to refer to objects and not manipulated as numbers, preventing this type of error. Array indexing is handled as a special case.

A pointer which does not have any address assigned to it is called a wild pointer. Any attempt to use such uninitialized pointers can cause unexpected behavior, either because the initial value is not a valid address, or because using it may damage other parts of the program. The result is often a segmentation fault, storage violation or wild branch (if used as a function pointer or branch address).

In systems with explicit memory allocation, it is possible to create a dangling pointer by deallocating the memory region it points into. This type of pointer is dangerous and subtle because a deallocated memory region may contain the same data as it did before it was deallocated but may be then reallocated and overwritten by unrelated code, unknown to the earlier code. Languages with garbage collection prevent this type of error because deallocation is performed automatically when there are no more references in scope.

Some languages, like C++, support smart pointers, which use a simple form of reference counting to help track allocation of dynamic memory in addition to acting as a reference. In the absence of reference cycles, where an object refers to itself indirectly through a sequence of smart pointers, these eliminate the possibility of dangling pointers and memory leaks. Delphi strings support reference counting natively.

The Rust programming language introduces a *borrow checker*, *pointer lifetimes*, and an optimisation based around optional types for null pointers to eliminate pointer bugs, without resorting to garbage collection.

Special kinds of pointers

Kinds defined by value

Null pointer

A **null pointer** has a value reserved for indicating that the pointer does not refer to a valid object. Null pointers are routinely used to represent conditions such as the end of a list of unknown length or the failure to perform some action; this use of null pointers can be compared to nullable types and to the *Nothing* value in an option type.

Dangling pointer

A **dangling pointer** is a pointer that does not point to a valid object and consequently may make a program crash or behave oddly. In the Pascal or C programming languages, pointers that are not specifically initialized may point to unpredictable addresses in memory.

The following example code shows a dangling pointer:

```
int func(void) {
    char *p1 = malloc(sizeof(char)); /* (undefined) value of some place on the heap */
    char *p2;       /* dangling (uninitialized) pointer */
    *p1 = 'a';      /* This is OK, assuming malloc() has not returned NULL. */
    *p2 = 'b';      /* This invokes undefined behavior */
}
```

Here, `p2` may point to anywhere in memory, so performing the assignment `*p2 = 'b';` can corrupt an unknown area of memory or trigger a segmentation fault.

Wild branch

Where a pointer is used as the address of the entry point to a program or start of a function which doesn't return anything and is also either uninitialized or corrupted, if a call or jump is nevertheless made to this address, a "wild branch" is said to have occurred. In other words, a wild branch is a function pointer that is wild (dangling).

The consequences are usually unpredictable and the error may present itself in several different ways depending upon whether or not the pointer is a "valid" address and whether or not there is (coincidentally) a valid instruction (opcode) at that address. The detection of a wild branch can present one of the most difficult and frustrating debugging exercises since much of the evidence may already have been destroyed beforehand or by execution of one or more inappropriate instructions at the branch location. If available, an instruction set simulator can usually not only detect a wild branch before it takes effect, but also provide a complete or partial trace of its history.

Kinds defined by structure

Autorelative pointer

An **autorelative pointer** is a pointer whose value is interpreted as an offset from the address of the pointer itself; thus, if a data structure has an autorelative pointer member that points to some portion of the data structure itself, then the data structure may be relocated in memory without having to update the value of the auto relative pointer.[12]

The cited patent also uses the term **self-relative pointer** to mean the same thing. However, the meaning of that term has been used in other ways:

- to mean an offset from the address of a structure rather than from the address of the pointer itself;
- to mean a pointer containing its own address, which can be useful for reconstructing in any arbitrary region of memory a collection of data structures that point to each other.[13]

Based pointer

A **based pointer** is a pointer whose value is an offset from the value of another pointer. This can be used to store and load blocks of data, assigning the address of the beginning of the block to the base pointer.[14]

Kinds defined by use or datatype

Multiple indirection

In some languages, a pointer can reference another pointer, requiring multiple dereference operations to get to the original value. While each level of indirection may add a performance cost, it is sometimes necessary in order to provide correct behavior for complex data structures. For example, in C it is typical to define a linked list in terms of an element that contains a pointer to the next element of the list:

```
struct element {
    struct element *next;
    int            value;
};

struct element *head = NULL;
```

This implementation uses a pointer to the first element in the list as a surrogate for the entire list. If a new value is added to the beginning of the list, `head` has to be changed to point to the new element. Since C arguments are always passed by value, using double indirection allows the insertion to be implemented correctly, and has the desirable side-effect of eliminating special case code to deal with insertions at the front of the list:

```
// Given a sorted list at *head, insert the element item at the first
// location where all earlier elements have lesser or equal value.
void insert(struct element **head, struct element *item) {
    struct element **p; // p points to a pointer to an element
    for (p = head; *p != NULL; p = &(*p)->next) {
        if (item->value <= (*p)->value)
            break;
    }
    item->next = *p;
    *p = item;
}

// Caller does this:
insert(&head, item);
```

In this case, if the value of `item` is less than that of `head`, the caller's `head` is properly updated to the address of the new item.

A basic example is in the argv argument to the main function in C (and C++), which is given in the prototype as `char **argv`—this is because the variable `argv` itself is a pointer to an array of strings (an array of arrays), so `*argv` is a pointer to the 0th string (by convention the name of the program), and `**argv` is the 0th character of the 0th string.

Function pointer

In some languages, a pointer can reference executable code, i.e., it can point to a function, method, or procedure. A function pointer will store the address of a function to be invoked. While this facility can be used to call functions dynamically, it is often a favorite technique of virus and other malicious software writers.

```
int sum(int n1, int n2) {   // Function with two integer parameters returning an integer value
    return n1 + n2;
}

int main(void) {
    int a, b, x, y;
    int (*fp)(int, int);    // Function pointer which can point to a function like sum
    fp = &sum;              // fp now points to function sum
    x = (*fp)(a, b);        // Calls function sum with arguments a and b
    y = sum(a, b);          // Calls function sum with arguments a and b
}
```

Back pointer

In doubly linked lists or tree structures, a back pointer held on an element 'points back' to the item referring to the current element. These are useful for navigation and manipulation, at the expense of greater memory use.