Pointer (computer programming)

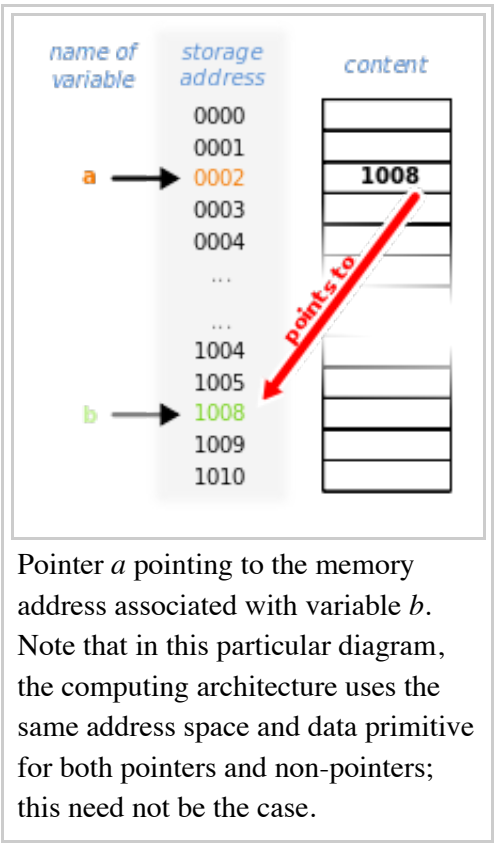
From Wikipedia, the free encyclopedia

In computer science, a **pointer** is a programming language object, whose value refers to (or "**points** to") another value stored elsewhere in the computer memory using its memory address. 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.

> 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]

Pointers to data significantly improve performance for repetitive operations such as traversing 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.

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 what are called virtual method tables.



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

A pointer is a simple, more concrete implementation of the more abstract *reference* data type. Several 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 where this is not possible. 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" a pointer whose value was never a valid memory address would cause a program to crash. 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 the contents of the pointer variable contain a value that is both a valid memory address and within the numerical range that the processor is capable of addressing).

Contents

History

Harold Lawson is credited with the 1964 invention of the pointer.[2] In 2000, Lawson was presented the Computer Pioneer Award by the IEEE "[f]or inventing the pointer variable and introducing this concept into PL/I, thus providing for the first time, the capability to flexibly treat linked lists in a general-purpose high level

language".[3] According to the Oxford English Dictionary, the **word** *pointer* first appeared in print as a *stack pointer* in a technical memorandum by the System Development Corporation.

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.

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 kilobytes of a data structure. This can easily be extended to 128K, 256K or 512K 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.

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 (back) 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.

Architectural roots

Pointers are a very thin abstraction on top of the addressing capabilities provided by most modern architectures. In the simplest scheme, an *address*, or a numeric index, is assigned to each unit of memory in the system, where the unit is typically either a byte or a word – depending on whether the architecture is byte-addressable or word-addressable – effectively transforming all of memory into a very large array. Then, if we have an address, the system provides an operation to retrieve the value stored in the memory unit at that address (usually utilizing the machine's general purpose registers).

In the usual case, a pointer is large enough to hold more addresses than there are units of memory in the system. This introduces the possibility that a program may attempt to access an address which corresponds to no unit of memory, either because not enough memory is installed (i.e. beyond the range of available memory) or the architecture does not support such addresses. The first case may, in certain platforms such as the Intel x86 architecture, be called a segmentation fault (segfault). The second case is possible in the current implementation of AMD64, where pointers are 64 bit long and addresses only extend to 48 bits. There, pointers must conform to certain rules (canonical addresses), so if a noncanonical pointer is dereferenced, the processor raises a general protection fault.

On the other hand, some systems have more units of memory than there are addresses. In this case, a more complex scheme such as memory segmentation or paging is employed to use different parts of the memory at different times. The last incarnations of the x86 architecture support up to 36 bits of physical memory addresses,

which were mapped to the 32-bit linear address space through the PAE paging mechanism. Thus, only 1/16 of the possible total memory may be accessed at a time. Another example in the same computer family was the 16-bit protected mode of the 80286 processor, which, though supporting only 16 MiB of physical memory, could access up to 1 GiB of virtual memory, but the combination of 16-bit address and segment registers made accessing more than 64 KiB in one data structure cumbersome. Some restrictions of ANSI pointer arithmetic may have been due to the segmented memory models of this processor family.

In order to provide a consistent interface, some architectures provide memory-mapped I/O, which allows some addresses to refer to units of memory while others refer to device registers of other devices in the computer. There are analogous concepts such as file offsets, array indices, and remote object references that serve some of the same purposes as addresses for other types of objects.

Uses

Pointers are directly supported without restrictions in languages such as PL/I, C, C++, Pascal, and 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, pointers and references are managed abstractly by the language using internal constructs like cons.

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. If the data elements in the array have lengths that are divisible by powers of two, this arithmetic is usually a bit more efficient. Padding is frequently used as a mechanism for ensuring this is the case, despite the increased memory requirement. 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
0x8130	0x00000005
0x8134	0x00000000

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

```
ptr = &a;
```

yields the following memory values:

Address	Contents
0x8130	0x00000005
0x8134	0x00008130

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
0x8130	0x00000008
0x8134	0x00008130

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

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, it is important to note that the `sizeof` operator will differ. 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 you assume that `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**
1000	2	0	0	0
1004	4	0	0	0
1008	3	0	0	0
100C	1	0	0	0
1010	5	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 (note that the "+1" really means to add one times the size of an `int` (4 bytes) not literally "plus one");
- `*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 $(i + 1)^{th}$ element of `array`;
- `*(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.

```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 */
};
```

Note that this pointer-recursive definition is essentially the same as the reference-recursive definition from the Haskell programming language:

```haskell
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 to m is passed instead. No copy of m itself 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 program the required 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. (Variables can also be stored in the CPU registers, but that's another matter) Dynamic memory allocation can only be made through pointers, and names (like with common variables) can't 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 = (struct Item *)malloc(sizeof(struct Item));
    if (item == NULL)
        return NULL;

    /* 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 = (char *)malloc(strlen(name) + 1);
    if (item->name == NULL) {
        free(item);
        return NULL;
    }
    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';
}
```

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` will 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.

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 a garbage collector.

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.

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.[10]

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.[11]

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.[12]

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
}
```

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.

Back pointer

In linked lists or tree structuress, 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.

Pointer declaration syntax overview

These pointer declarations cover most variants of pointer declarations. Of course is it possible to have triple pointers, but the main principles behind a triple pointer already exists in a double pointer.

```
char cff [5][5];    /* array of arrays of chars; a char can be any sign */
char *cfp [5];      /* array of pointers to chars */
char **cpp;         /* pointer to pointer to chars ("double pointer") */
char (*cpf) [5];    /* pointer to an array of chars */
char *cpF();        /* function which returns a pointer to chars */
char (*CFp)();      /* pointer to a function which returns chars */
char (*cfpF*())[5]; /* function which returns pointers to an array of chars */
char (*cpFf[5])();  /* an array of pointers to functions, which all return chars */
```

The () and [] have a higher priority than *. [13]

Wild branch

Where a pointer is used as the address of the entry point to a program or start of a 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. 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.

Simulation using an array index

It is possible to simulate pointer behavior using an index to an (normally one-dimensional) array.

Primarily for languages which do not support pointers explicitly but *do* support arrays, the array can be thought of and processed as if it were the entire memory range (within the scope of the particular array) and any index to it can be thought of as equivalent to a general purpose register in assembly language (that points to the individual bytes but whose actual value is relative to the start of the array, not its absolute address in memory). Assuming the array is, say, a contiguous 16 megabyte character data structure, individual bytes (or a string of contiguous bytes within the array) can be directly addressed and manipulated using the name of the array with a 31 bit unsigned integer as the simulated pointer (this is quite similar to the *C arrays* example shown above). Pointer arithmetic can be simulated by adding or subtracting from the index, with minimal additional overhead compared to genuine pointer arithmetic.

It is even theoretically possible, using the above technique, together with a suitable instruction set simulator to simulate *any* machine code or the intermediate (byte code) of *any* processor/language in another language that does not support pointers at all (for example Java / JavaScript). To achieve this, the binary code can initially be

loaded into contiguous bytes of the array for the simulator to "read", interpret and action entirely within the memory contained of the same array. If necessary, to completely avoid buffer overflow problems, bounds checking can usually be actioned for the compiler (or if not, hand coded in the simulator).

Support in various programming languages

Ada

Ada is a strongly typed language where all pointers are typed and only safe type conversions are permitted. All pointers are by default initialized to `null`, and any attempt to access data through a `null` pointer causes an exception to be raised. Pointers in Ada are called *access types*. Ada 83 did not permit arithmetic on access types (although many compiler vendors provided for it as a non-standard feature), but Ada 95 supports “safe” arithmetic on access types via the package `System.Storage_Elements`.

BASIC

Several old versions of BASIC for the Windows platform had support for STRPTR() to return the address of a string, and for VARPTR() to return the address of a variable. Visual Basic 5 also had support for OBJPTR() to return the address of an object interface, and for an ADDRESSOF operator to return the address of a function. The types of all of these are integers, but their values are equivalent to those held by pointer types.

Newer dialects of BASIC, such as FreeBASIC or BlitzMax, have exhaustive pointer implementations, however. In FreeBASIC, arithmetic on `ANY` pointers (equivalent to C's `void*`) are treated as though the `ANY` pointer was a byte width. `ANY` pointers cannot be dereferenced, as in C. Also, casting between `ANY` and any other type's pointers will not generate any warnings.

```
dim as integer f = 257
dim as any ptr g = @f
dim as integer ptr i = g
assert(*i = 257)
assert( (g + 4) = (@f + 1) )
```

C and C++

In C and C++ pointers are variables that store addresses and can be *null*. Each pointer has a type it points to, but one can freely cast between pointer types (but not between a function pointer and non-function pointer type). A special pointer type called the “void pointer” allows pointing to any (non-function) variable type, but is limited by the fact that it cannot be dereferenced directly (it shall be cast). The address itself can often be directly manipulated by casting a pointer to and from an integral type of sufficient size, though the results are implementation-defined and may indeed cause undefined behavior; while earlier C standards did not have an integral type that was guaranteed to be large enough, C99 specifies the `uintptr_t` *typedef name* defined in `<stdint.h>`, but an implementation need not provide it.

C++ fully supports C pointers and C typecasting. It also supports a new group of typecasting operators to help catch some unintended dangerous casts at compile-time. Since C++11, the C++ standard library also provides smart pointers (`unique_ptr`, `shared_ptr` and `weak_ptr`) which can be used in some situations as a safe alternative to primitive C pointers. C++ also supports another form of reference, quite different from a pointer, called simply a *reference* or *reference type*.

Pointer arithmetic, that is, the ability to modify a pointer's target address with arithmetic operations (as well as magnitude comparisons), is restricted by the language standard to remain within the bounds of a single array object (or just after it), and will otherwise invoke undefined behavior. Adding or subtracting from a pointer moves it by a multiple of the size of the datatype it points to. For example, adding 1 to a pointer to 4-byte integer values will increment the pointer by 4. This has the effect of incrementing the pointer to point at the next element in a contiguous array of integers—which is often the intended result. Pointer arithmetic cannot be performed on `void` pointers because the void type has no size, and thus the pointed address can not be added to, although gcc and other compilers will perform byte arithmetic on `void*` as a non-standard extension, treating it as if it were `char *`.

Pointer arithmetic provides the programmer with a single way of dealing with different types: adding and subtracting the number of elements required instead of the actual offset in bytes. (though the `char` pointer, `char` being defined as always having a size of one byte, allows the element offset of pointer arithmetic to in practice be equal to a byte offset) In particular, the C definition explicitly declares that the syntax `a[n]`, which is the `n`-th element of the array `a`, is equivalent to `*(a + n)`, which is the content of the element pointed by `a + n`. This implies that `n[a]` is equivalent to `a[n]`, and one can write, e.g., `a[3]` or `3[a]` equally well to access the fourth element of an array `a`.

While powerful, pointer arithmetic can be a source of computer bugs. It tends to confuse novice programmers, forcing them into different contexts: an expression can be an ordinary arithmetic one or a pointer arithmetic one, and sometimes it is easy to mistake one for the other. In response to this, many modern high-level computer languages (for example Java) do not permit direct access to memory using addresses. Also, the safe C dialect Cyclone addresses many of the issues with pointers. See C programming language for more discussion.

The **`void` pointer**, or **`void*`**, is supported in ANSI C and C++ as a generic pointer type. A pointer to `void` can store an address to any non-function data type, and, in C, is implicitly converted to any other pointer type on assignment, but it must be explicitly cast if dereferenced inline. K&R C used `char*` for the "type-agnostic pointer" purpose (before ANSI C).

```
int x = 4;
void* p1 = &x;
int* p2 = p1;       // void* implicitly converted to int*: valid C, but not C++
int a = *p2;
int b = *(int*)p1;  // when dereferencing inline, there is no implicit conversion
```

C++ does not allow the implicit conversion of `void*` to other pointer types, even in assignments. This was a design decision to avoid careless and even unintended casts, though most compilers only output warnings, not errors, when encountering other ill casts.

```
int x = 4;
```

```
void* p1 = &x;
int* p2 = p1;                    // this fails in C++: there is no implicit conversion from void*
int* p3 = (int*)p1;              // C-style cast
int* p4 = static_cast<int*>(p1); // C++ cast
```

In C++, there is no `void&` (reference to void) to complement `void*` (pointer to void), because references behave like aliases to the variables they point to, and there can never be a variable whose type is `void`.

C#

In the C# programming language, pointers are supported only under certain conditions: any block of code including pointers must be marked with the `unsafe` keyword. Such blocks usually require higher security permissions than pointerless code to be allowed to run. The syntax is essentially the same as in C++, and the address pointed can be either managed or unmanaged memory. However, pointers to managed memory (any pointer to a managed object) must be declared using the `fixed` keyword, which prevents the garbage collector from moving the pointed object as part of memory management while the pointer is in scope, thus keeping the pointer address valid.

An exception to this is from using the `IntPtr` structure, which is a safe managed equivalent to `int*`, and does not require unsafe code. This type is often returned when using methods from the `System.Runtime.InteropServices`, for example:

```
// Get 16 bytes of memory from the process's unmanaged memory
IntPtr pointer = System.Runtime.InteropServices.Marshal.AllocHGlobal(16);

// Do something with the allocated memory

// Free the allocated memory
System.Runtime.InteropServices.Marshal.FreeHGlobal(pointer);
```

The .NET framework includes many classes and methods in the `System` and `System.Runtime.InteropServices` namespaces (such as the `Marshal` class) which convert .NET types (for example, `System.String`) to and from many unmanaged types and pointers (for example, `LPWSTR` or `void*`) to allow communication with unmanaged code.

COBOL

The COBOL programming language supports pointers to variables. Primitive or group (record) data objects declared within the `LINKAGE SECTION` of a program are inherently pointer-based, where the only memory allocated within the program is space for the address of the data item (typically a single memory word). In program source code, these data items are used just like any other `WORKING-STORAGE` variable, but their contents are implicitly accessed indirectly through their `LINKAGE` pointers.

Memory space for each pointed-to data object is typically allocated dynamically using external `CALL` statements or via embedded extended language constructs such as `EXEC CICS` or `EXEC SQL` statements.

Extended versions of COBOL also provide pointer variables declared with `USAGE IS POINTER` clauses. The values of such pointer variables are established and modified using `SET` and `SET ADDRESS` statements.

Some extended versions of COBOL also provide `PROCEDURE-POINTER` variables, which are capable of storing the addresses of executable code.

PL/I

The PL/I language provides full support for pointers to all data types (including pointers to structures), recursion, multitasking, string handling, and extensive built-in functions. PL/I was quite a leap forward compared to the programming languages of its time.

D

The D programming language is a derivative of C and C++ which fully supports C pointers and C typecasting.

Eiffel

The Eiffel object-oriented language employs value and reference semantics without the need for pointer arithmetics. Nevertheless, pointer classes are provided. They offer pointer arithmetics, typecasting, explicit memory management, interfacing with non-Eiffel software, and other features.

Fortran

Fortran-90 introduced a strongly typed pointer capability. Fortran pointers contain more than just a simple memory address. They also encapsulate the lower and upper bounds of array dimensions, strides (for example, to support arbitrary array sections), and other metadata. An *association operator*, => is used to associate a `POINTER` to a variable which has a `TARGET` attribute. The Fortran-90 `ALLOCATE` statement may also be used to associate a pointer to a block of memory. For example, the following code might be used to define and create a linked list structure:

```
type real_list_t
  real :: sample_data(100)
  type (real_list_t), pointer :: next => null ()
end type

type (real_list_t), target :: my_real_list
type (real_list_t), pointer :: real_list_temp

real_list_temp => my_real_list
do
  read (1,iostat=ioerr) real_list_temp%sample_data
  if (ioerr /= 0) exit
  allocate (real_list_temp%next)
  real_list_temp => real_list_temp%next
end do
```

Fortran-2003 adds support for procedure pointers. Also, as part of the *C Interoperability* feature, Fortran-2003 supports intrinsic functions for converting C-style pointers into Fortran pointers and back.

Go

Go has pointers. Its declaration syntax is equivalent to that of C, but written the other way around, ending with the type. Unlike C, Go has garbage collection, and disallows pointer arithmetic. Reference types, like in C++, do not exist. Some built-in types, like maps and channels, are boxed (i.e. internally they are pointers to mutable structures), and are initialized using the `make` function. As a different (than reference types) approach to unified syntax between pointers and non-pointers, the arrow (`->`) operator has been dropped—it is possible to use the dot operator directly on a pointer to a data type to access a field or method of the dereferenced value, as if the dot operator were used on the underlying data type. This, however, only works with 1 level of indirection.

Java

Unlike C, C++, or Pascal, there is no explicit representation of pointers in Java. Instead, more complex data structures like objects and arrays are implemented using references. The language does not provide any explicit pointer manipulation operators. It is still possible for code to attempt to dereference a null reference (null pointer), however, which results in a run-time exception being thrown. The space occupied by unreferenced memory objects is recovered automatically by garbage collection at run-time.[14]

Modula-2

Pointers are implemented very much as in Pascal, as are `VAR` parameters in procedure calls. Modula-2 is even more strongly typed than Pascal, with fewer ways to escape the type system. Some of the variants of Modula-2 (such as Modula-3) include garbage collection. For instance, when it comes to computing time, one must carry the remainder past 12.

Oberon

Much as with Modula-2, pointers are available. There are still fewer ways to evade the type system and so Oberon and its variants are still safer with respect to pointers than Modula-2 or its variants. As with Modula-3, garbage collection is a part of the language specification.

Pascal

Unlike many languages that feature pointers, standard ISO Pascal only allows pointers to reference dynamically created variables that are anonymous and does not allow them to reference standard static or local variables.[15] It does not have pointer arithmetic. Pointers also must have an associated type and a pointer to one type is not compatible with a pointer to another type (e.g. a pointer to a char is not compatible with a pointer to an integer). This helps eliminate the type security issues inherent with other pointer implementations, particularly those

used for PL/I or C. It also removes some risks caused by dangling pointers, but the ability to dynamically let go of referenced space by using the `dispose` standard procedure (which has the same effect as the `free` library function found in C) means that the risk of dangling pointers has not been entirely eliminated.[16]

However, in some commercial and open source Pascal (or derivatives) compiler implementations —like Free Pascal,[17] Turbo Pascal or the Object Pascal in Embarcadero Delphi— a pointer is allowed to reference standard static or local variables and can be cast from one pointer type to another. Moreover pointer arithmetic is unrestricted: adding or subtracting from a pointer moves it by that number of bytes in either direction, but using the `Inc` or `Dec` standard procedures with it moves the pointer by the size of the data type it is *declared* to point to. An untyped pointer is also provided under the name `Pointer`, which is compatible with other pointer types.

Perl

The Perl programming language supports pointers, although rarely used, in the form of the pack and unpack functions. These are intended only for simple interactions with compiled OS libraries. In all other cases, Perl uses references, which are typed and do not allow any form of pointer arithmetic. They are used to construct complex data structures.[18]

See also

- Address constant
- Bounded pointer
- Buffer overflow
- Function pointer
- Hazard pointer
- Opaque pointer
- Pointer swizzling
- Reference (computer science)
- Static program analysis
- Storage violation
- Tagged pointer
- Variable (computer science)

References

1. Donald Knuth (1974). "Structured Programming with go to Statements" (PDF). *Computing Surveys* **6** (5): 261–301. doi:10.1145/356635.356640.
2. Milestones in Computer Science and Information Technology (http://books.google.com/books?id=JTYPKxug49IC&pg=PA204&lpg=PA204&dq=Harold+Lawson+pointer&source=web&ots=C5QdVz2xM8&sig=xhh0SWuR-L72H6H9xgEmxD5qzBc&hl=en&ei=lCuLSbTwKY_-0AWErp2iBw&sa=X&oi=book_result&resnum=10&ct=result)
3. IEEE Computer Society awards list (http://awards.computer.org/ana/award/viewPastRecipients.action?id=13)
4. ISO/IEC 9899, clause 6.7.5.1, paragraph 1.
5. ISO/IEC 9899, clause 6.7.8, paragraph 10.
6. ISO/IEC 9899, clause 7.17, paragraph 3: *NULL... which expands to an implementation-defined null pointer constant...*
7. ISO/IEC 9899, clause 6.5.3.2, paragraph 4, footnote 87: *If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined... Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer...*
8. Plauger, P J; Brodie, Jim (1992). *ANSI and ISO Standard C Programmer's Reference*. Redmond, WA: Microsoft Press.

pp. 108, 51. ISBN 1-55615-359-7. "An array type does not contain additional holes because all other types pack tightly when composed into arrays *[at page 51]*"
9. WG14 N1124 (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf), C – Approved standards: ISO/IEC 9899 – Programming languages – C (http://www.open-std.org/jtc1/sc22/wg14/www/standards.html), 2005-05-06.
10. us patent 6625718 (http://worldwide.espacenet.com/textdoc?DB=EPODOC&IDX=us6625718), Steiner, Robert C. (Broomfield, CO), "Pointers that are relative to their own present locations", issued 2003-09-23, assigned to Avaya Technology Corp. (Basking Ridge, NJ)
11. us patent 6115721 (http://worldwide.espacenet.com/textdoc?DB=EPODOC&IDX=us6115721), Nagy, Michael (Tampa, FL), "System and method for database save and restore using self-pointers", issued 2000-09-05, assigned to IBM (Armonk, NY)
12. Based Pointers (http://msdn.microsoft.com/en-us/library/57a97k4e.aspx)
13. Ulf Bilting, Jan Skansholm, "Vägen till C" (the Road to C), third edition ,page 169, ISBN 91-44-01468-6
14. Nick Parlante, [1] (http://cslibrary.stanford.edu/102/PointersAndMemory.pdf#%22Pointers%20and%20Memory%22), Stanford Computer Science Education Library (http://cslibrary.stanford.edu#), pp. 9–10 (2000).
15. ISO 7185 Pascal Standard (unofficial copy), section 6.4.4 Pointer-types (http://standardpascal.org/iso7185.html#6.4.4%20Pointer-types) and subsequent.
16. J. Welsh, W. J. Sneeringer, and C. A. R. Hoare, "Ambiguities and Insecurities in Pascal," *Software Practice and Experience 7*, pp. 685–696 (1977)
17. Free Pascal Language Reference guide, section 3.4 Pointers (http://www.freepascal.org/docs-html/ref/refse15.html#x43-490003.4)
18. // Making References (Perl References and nested data structures) (http://perldoc.perl.org/perlref.html#Making-References)

External links

- Pointers and Memory (http://cslibrary.stanford.edu/) Introduction to pointers – Stanford Computer Science Education Library
- 0pointer.de (http://0pointer.de/) A terse list of minimum length source codes that dereference a null pointer in several different programming languages
- "The C book" – containing pointer examples in ANSI C (http://publications.gbdirect.co.uk/c_book/chapter5/pointers.html)
- Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 22, Working Group WG 14 (2007-09-08). *International Standard ISO/IEC 9899* (PDF). *Committee Draft*. .

Wikimedia Commons has media related to ***Pointers (computing)***.

Wikiversity has learning materials about ***Pointers***

The Wikibook *C Programming* has a page on the topic of: ***Pointers***

Retrieved from "https://en.wikipedia.org/w/index.php?title=Pointer_(computer_programming)&oldid=724185554"

Categories: Data types | Primitive types | American inventions

---