1.What is CallStack?

**call stack** is a [stack](about:blank) data structure that stores information about the active [subroutines](about:blank) of a [computer program](about:blank). This kind of stack is also known as an **execution stack**,**control stack**, **run-time stack**, or **machine stack**, and is often shortened to just "the stack". Although maintenance of the call stack is important for the proper functioning of most [software](about:blank), the details are normally hidden and automatic in [high-level programming languages](about:blank).

A call stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finishes executing. An active subroutine is one that has been called but is yet to complete execution after which control should be handed back to the point of call. Such activations of subroutines may be nested to any level (recursive as a special case), hence the stack structure. If, for example, a subroutine DrawSquare calls a subroutine DrawLine from four different places, DrawLine must know where to return when its execution completes. To accomplish this, the [address](about:blank) following the call [instruction](about:blank), the *return address*, is pushed onto the call stack with each call.

Functions of the call stack[edit]

As noted above, the primary purpose of a call stack is:

**Storing the return address**

When a subroutine is called, the location (address) of the instruction at which it can later resume needs to be saved somewhere. Using a stack to save the return address has important advantages over alternatives. One is that each task has its own stack, and thus the subroutine can be [reentrant](about:blank), that is, can be active simultaneously for different tasks doing different things. Another benefit is that [recursion](about:blank) is automatically supported. When a function calls itself recursively, a return address needs to be stored for each activation of the function so that it can later be used to return from the function activation. This capability is automatic with a stack.

A call stack may serve additional functions, depending on the language, operating system, and machine environment. Among them can be:

**Local data storage**

A subroutine frequently needs memory space for storing the values of [local variables](about:blank), the variables that are known only within the active subroutine and do not retain values after it returns. It is often convenient to allocate space for this use by simply moving the top of the stack by enough to provide the space. This is very fast compared to [heap](about:blank) allocation. Note that each separate activation of a subroutine gets its own separate space in the stack for locals.

**Parameter passing**

Subroutines often require that values for [parameters](about:blank) be supplied to them by the code which calls them, and it is not uncommon that space for these parameters may be laid out in the call stack. Generally if there are only a few small parameters, [processor registers](about:blank) will be used to pass the values, but if there are more parameters than can be handled this way, memory space will be needed. The call stack works well as a place for these parameters, especially since each call to a subroutine, which will have differing values for parameters, will be given separate space on the call stack for those values.

**Evaluation stack**

Operands for arithmetic or logical operations are most often placed into registers and operated on there. However, in some situations the operands may be stacked up to an arbitrary depth, which means something more than registers must be used (this is the case of register spilling). The stack of such operands, rather like that in an [RPN calculator](about:blank), is called an evaluation stack, and may occupy space in the call stack.

**Pointer to current instance**

Some [object-oriented languages](about:blank) (e.g., [C++](about:blank)), store the [*this* pointer](about:blank) along with function arguments in the call stack when invoking methods. The *this* pointer points to the [object](about:blank) instanceassociated with the method to be invoked.

**Enclosing subroutine context**

Some programming languages (e.g., [Pascal](about:blank) and [Ada](about:blank)) support [nested subroutines](about:blank), allowing an inner routine to access the context of its outer enclosing routine, i.e., the parameters and local variables within the scope of the outer routine. Such static nesting can repeat - a function declared within a function declared within a function... The implementation must provide a means by which a called function at any given static nesting level can reference the enclosing frame at each enclosing nesting level. Commonly this reference is implemented by a pointer to the encompassing frame, called a "downstack link" or "static link", to distinguish it from the "dynamic link" that refers to the immediate caller (which need not be the static parent function). For example, languages often allow inner routines to call themselves recursively, resulting in multiple call frames for the inner routine's invocations, all of whose static links point to the same outer routine context. Instead of a static link, the references to the enclosing static frames may be collected into an array of pointers known as a *display* which is indexed to locate a desired frame. The [Burroughs B6500](about:blank) had such a display in hardware that supported up to 32 levels of static nesting.

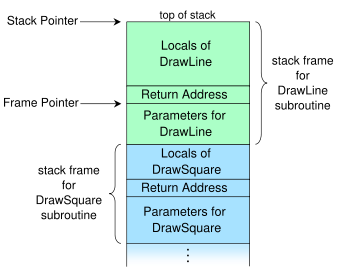
**Other return state**

Besides the return address, in some environments there may be other machine or software states that need to be restored when a subroutine returns. This might include things like privilege level, exception handling information, arithmetic modes, and so on. If needed, this may be stored in the call stack just as the return address is.

The typical call stack is used for the return address, locals, and parameters (known as a *call frame*). In some environments there may be more or fewer functions assigned to the call stack. In the[Forth programming language](about:blank), for example, ordinarily only the return address, counted loop parameters and indexes, and possibly local variables are stored on the call stack (which in that environment is named the *return stack*), although any data can be temporarily placed there using special return stack handling code so long as the needs of calls and returns are respected; parameters are ordinarily stored on a separate *data stack* or *parameter stack*, typically called *the* stack in Forth terminology even though there is a call stack since it is usually accessed more explicitly. Some Forths also have a third stack for [floating point](about:blank) parameters.

Structure[edit]

A call stack is composed of **stack frames** (also called **activation records** or **activation frames**). These are [machine dependent](about:blank) and [ABI](about:blank)-dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine which has not yet terminated with a return. For example, if a subroutine named DrawLine is currently running, having been called by a subroutine DrawSquare, the top part of the call stack might be laid out like this:

[](about:blank)

A diagram like this can be drawn in either direction as long as the placement of the top, and so direction of stack growth, is understood. Furthermore, independently of this, architectures differ as to whether call stacks grow towards higher addresses or towards lower addresses. The logic of the diagram is independent of the addressing choice.

The stack frame at the top of the stack is for the currently executing routine. The stack frame usually includes at least the following items (in push order):

* the arguments (parameter values) passed to the routine (if any);
* the return address back to the routine's caller (e.g. in the DrawLine stack frame, an address into DrawSquare's code); and
* space for the local variables of the routine (if any).

**The stack and frame pointers**[edit]

The data stored in the stack frame may sometimes be accessed directly via the **stack pointer** register (SP, which indicates the current top of the stack). However, as the stack pointer is variable during the activation of the routine, memory locations within the stack frame are more typically accessed via a separate register which makes relative addressing simpler and also enables dynamic allocation mechanisms (see below). This register is often termed the **frame pointer** or **stack base pointer** (BP) and is set up at procedure entry to point to a *fixed* location in the frame structure (such as the return address).

**Stack frame sizes**[edit]

As different routines have different parameters and local data, stack frames have varying sizes. Although they may often be fixed across all activations of a particular routine, many modern languages also support *dynamic* allocations on the stack, which means that the local data area will vary from activation to activation with a size that may be unspecified when the program is[compiled](about:blank). In this case access via a frame pointer, rather than via the stack pointer, is usually necessary since the offsets from the stack top to values such as the return address would not be known at compile time. If the subroutine does not use dynamic stack allocation and does not call any further subroutines, the frame pointer is not needed, and the register may be used for other purposes.

**Storing the address to the caller's frame**[edit]

In most systems a stack frame has a field to contain the previous value of the frame pointer register, the value it had while the caller was executing. For example, the stack frame of DrawLinewould have a memory location holding the frame pointer value that DrawSquare uses (not shown in the diagram above). The value is saved upon entry to the subroutine and restored upon return. Having such a field in a known location in the stack frame enables code to access each frame successively underneath the currently executing routine's frame, and also allows the routine to easily restore the frame pointer to the caller's frame, just before it returns.

**Lexically nested routines**[edit]

*Further information:*[*Nested function*](about:blank)*and*[*Non-local variable*](about:blank)

Programming languages that support [nested subroutines](about:blank) also have a field in the call frame that points to the stack frame of the *latest* activation of the procedure that most closely encapsulates the callee, i.e. the immediate *scope* of the callee. This is called an **access link** or **static link** (as it keeps track of static nesting during dynamic and recursive calls) and provides the routine (as well as any other routines it may invoke) access to the local data of its encapsulating routines at every nesting level. Some architectures, compilers, or optimization cases store one link for each enclosing level (not just the immediately enclosing), so that deeply nested routines that access shallow data do not have to traverse several links; this strategy is often called a **display**.[[1]](about:blank#cite_note-1) Access link(s) can be optimized away in cases where an inner function does not access any (non constant) local data in the encapsulation—pure functions, i.e. routines communicating via argument(s) and return value(s) only would be an example of this. Some historical computers, such as the [Burroughs large systems](about:blank), had special "display registers" to support nested functions while compilers for most modern machines (such as the ubiquitous x86) simply reserve a few words on the stack for the pointers, as needed.

**Overlap**[edit]

For some purposes, the stack frame of a subroutine and that of its caller can be considered to overlap, the overlap consisting of the area where the parameters are passed from the caller to the callee. In some environments, the caller pushes each argument onto the stack, thus extending its stack frame, then invokes the callee. In other environments, the caller has a preallocated area at the top of its stack frame to hold the arguments it supplies to other subroutines it calls. This area is sometimes termed the **outgoing arguments area** or **callout area**. Under this approach, the size of the area is calculated by the compiler to be the largest needed by any called subroutine.

Use[edit]

**Call site processing**[edit]

Usually the call stack manipulation needed at the site of a call to a subroutine is minimal (which is good since there can be many call sites for each subroutine to be called). The values for the actual arguments are evaluated at the call site, since they are specific to the particular call, and either pushed onto the stack or placed into registers, as determined by the [calling convention](about:blank)being used. The actual call instruction, such as "Branch and Link," is then typically executed to transfer control to the code of the target subroutine.

**Subroutine entry processing**[edit]

In the called subroutine, the first code executed is usually termed the [subroutine prologue](about:blank), since it does the necessary housekeeping before the code for the statements of the routine is begun.

The prologue will commonly save the return address left in a register by the call instruction by pushing the value onto the call stack. Similarly, the current stack pointer and/or frame pointer values may be pushed. Alternatively, some instruction set architectures automatically provide comparable functionality as part of the action of the call instruction itself, and in such an environment the prologue need not do this.

If frame pointers are being used, the prologue will typically set the new value of the frame pointer register from the stack pointer. Space on the stack for local variables can then be allocated by incrementally changing the stack pointer.

The [Forth programming language](about:blank) allows explicit winding of the call stack (called there the "return stack").

**Return processing**[edit]

When a subroutine is ready to return, it executes an epilogue that undoes the steps of the prologue. This will typically restore saved register values (such as the frame pointer value) from the stack frame, pop the entire stack frame off the stack by changing the stack pointer value, and finally branch to the instruction at the return address. Under many calling conventions the items popped off the stack by the epilogue include the original argument values, in which case there usually are no further stack manipulations that need to be done by the caller. With some calling conventions, however, it is the caller's responsibility to remove the arguments from the stack after the return.

**Unwinding**[edit]

Returning from the called function will pop the top frame off of the stack, perhaps leaving a return value.

Some languages (such as [Pascal](about:blank)) allow a global goto statement to transfer control out of a nested function and into a previously invoked outer function. This operation requires the stack to be unwound, removing as many stack frames as necessary to restore the proper context to transfer control to the target statement within the enclosing outer function. Such transfers of control are generally used only for error handling.

Other languages (such as [Object Pascal](about:blank)) provide [exception handling](about:blank), which also requires unwinding of the stack. The stack frame of a function contains one or more entries specifying exception handlers. When an exception is thrown, the stack is unwound until an exception handler is found that is prepared to handle (catch) the exception. [Common Lisp](about:blank) allows control of what happens when the stack is unwound by using the unwind-protect special operator.

When applying a [continuation](about:blank), the stack is (logically) unwound and then rewound with the stack of the continuation. This is not the only way to implement continuations; for example, using multiple, explicit stacks, application of a continuation can simply activate its stack and wind a value to be passed. The [Scheme programming language](about:blank) allows arbitrary thunks to be executed in specified points on "unwinding" or "rewinding" of the control stack when a continuation is invoked.

Call stack inspection[edit]

The call stack can sometimes be inspected as the program is running. Depending on how the program is written and compiled, the information on the stack can be used to determine intermediate values and function call traces. This has been used to generate fine-grained automated tests,[2] and in cases like Ruby and Smalltalk, to implement first-class continuations.

Performance analysis[edit]

Taking regular-time samples of the call stack can be very useful in profiling the performance of programs. The reason is if a subroutine's pointer appears on the call stack sampling data many times, It is likely a code bottleneck and should be inspected for performance problems.[[*dubious*](about:blank)*–*[*discuss*](about:blank#Dubious)] See [Performance analysis](about:blank) and [Deep sampling](about:blank).

Security[edit]

In a language with free pointers and/or non-checked array writes (such as C), the mixing of control flow data affecting the execution of code (return addresses, saved frame pointers) and simple program data (parameters, return values) in a call stack is a security risk, possibly [exploitable](about:blank) through [buffer overflows](about:blank).

2.Functional Pointer table?

I have a table of function pointers, and a function pointer variable p which  
steps through the functions.  
  
But I'm not allowed to increment the variable using ++p.  
  
What's the problem here?  
  
Also, I may not be interested in returning from any of these functions (each  
will call the next according to some global variable). Any recommended way  
of doing this (throwing away return address) other than a crude asm("pop  
R")?  
  
Thanks,  
  
Bart.  
#include <stdio.h>  
#include <stdlib.h>  
  
void f1(void);  
void f2(void);  
void f3(void);  
void f4(void);  
  
int main(void)  
{  
void (\*table[])(void)={&f1,&f2,&f3,&f4}; /\* table of function pointers \*/  
void (\*p)(void); /\* pointer to one of the functions (I hope) \*/  
  
p=table[0];  
  
while(1)  
{ (\*p)(); /\* Call function @p \*/  
++p; /\* COMPILE ERROR HERE \*/  
};  
  
}  
  
void f1(void){puts("F1 CALLED");return;};  
void f2(void){puts("F2 CALLED");return;};  
void f3(void){puts("F3 CALLED");return;};  
void f4(void){puts("F4 CALLED"); exit(0);};

3.What is Functional Pointers?

Function Pointers provide some extremely interesting, efficient and elegant programming techniques. You can use them to replace *switch/if*-statements, to realize your own *late-binding* or to implement*callbacks*. Unfortunately - probably due to their complicated syntax - they are treated quite stepmotherly in most computer books and documentations. If at all, they are addressed quite briefly and superficially. They are less error prone than normal pointers cause you will never allocate or deallocate memory with them. All you've got to do is to understand what they are and to learn their syntax. But keep in mind: Always ask yourself if you really need a function pointer. It's nice to realize one's own *late-binding* but to use the existing structures of C++ may make your code more readable and clear. One aspect in the case of *late-binding* is runtime: If you call a virtual function, your program has got to determine which one has got to be called. It does this using a V-Table containing all the possible functions. This costs some time each call and maybe you can save some time using function pointers instead of virtual functions. Maybe not ... BTW: Modern compilers are very good! With myBorland Compiler the time I was able to save calling a virtual function which multiplies two floats was about 2 percent.

[Main Index](about:blank) [top](about:blank)

### 1.1  What is a Function Pointer?

Function Pointers are pointers, i.e. variables, which point to the address of a function. You must keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus a function in the program code is, like e.g. a character field, nothing else than an address. It is only important how you, or better your compiler/processor, interpret the memory a pointer points to.

**Uses of Function Pointers**

Pointer to function is useful when you want to choose a function dynamically at run time – usually based on certain conditions. Function Pointers are used to implement generic functions like qsort(), callbacks, state machines and anywhere else run time binding of a function is required. Also Function pointers are used to store interrupt handlers in tables.

To begin, check out the qsort fucntion

|  |
| --- |
| void qsort(void \*base, size\_t nmemb, size\_t size,  int (\*compar)(const void \*, const void \*)); |

Here, you need to pass a pointer to the function that compares the objects that you are sorting. Since qsort() is a generic implementation and does not care what you are sorting, it is up to you to implement a funtion which compares the objects that you are sorting and pass it on to qsort(). qsort() just invokes your compare function by dereferencing the function pointer.

4.Virtual Table?

A **virtual method table**, **virtual function table**, **virtual call table**, [dispatch table](about:blank), or **vtable**, is a mechanism used in a [programming language](about:blank) to support [dynamic dispatch](about:blank) (or [run-time](about:blank) method[binding](about:blank)).

Suppose a program contains several [classes](about:blank) in an [inheritance](about:blank) hierarchy: a [superclass](about:blank), Cat, and two [subclasses](about:blank), HouseCat and Lion. Class Cat defines a [virtual function](about:blank) named speak, so its subclasses may provide an appropriate implementation (e.g. either meow or roar).

When the program calls the speak method on a Cat pointer (which can point to a Cat class, or any subclass of Cat), the calling code must be able to determine which implementation to call, depending on the actual type of object that is pointed to. Because the type of object pointed to by the Cat pointer is not determined at [compile-time](about:blank), the decision as to which branch to take cannot be decided at compile-time.

There are a variety of different ways to implement such dynamic dispatch, but the vtable (virtual table) solution is especially common among [C++](about:blank) and related languages (such as [D](about:blank) and [C#](about:blank)). Languages which separate the programmatic interface of objects from the implementation, like [Visual Basic](about:blank) and [Delphi](about:blank), also tend to use the vtable approach, because it allows objects to use a different implementation simply by using a different set of method pointers

An object's dispatch table will contain the [addresses](about:blank) of the object's dynamically bound methods. Method calls are performed by fetching the method's address from the object's dispatch table. The dispatch table is the same for all objects belonging to the same class, and is therefore typically shared between them. Objects belonging to type-compatible classes (for example siblings in an inheritance hierarchy) will have dispatch tables with the same layout: the address of a given method will appear at the same offset for all type-compatible classes. Thus, fetching the method's address from a given dispatch table offset will get the method corresponding to the object's actual class.[[1]](about:blank#cite_note-1)

The [C++](about:blank) standards do not mandate exactly how dynamic dispatch must be implemented, but compilers generally use minor variations on the same basic model.

Typically, the compiler creates a separate vtable for each class. When an object is created, a pointer to this vtable, called the **virtual table pointer**, **vpointer** or **VPTR**, is added as a hidden member of this object (becoming its first member unless it's made the last[[2]](about:blank#cite_note-2)). The compiler also generates "hidden" code in the [constructor](about:blank) of each class to initialize the vpointers of its objects to the address of the corresponding vtable. Note that the location of the vpointer in the object instance is not standard among all compilers, and relying on the position may result in unportable code. For example, [g++](about:blank) previously placed the vpointer at the end of the object.[[3]](about:blank#cite_note-3)