Stack-Based Allocation

Each instance of a **subroutine** at runtime has its own **frame** (also called an **activation record**) on the stack, containing arguments and return values, local variables, temporaries, and bookkepping information.

The **activation record** contains **arguments** and **return values**, **local variables**, **temporaries**, and **bookkepping** information.

Temporaries are typically intermediate values produced in complex calculations.

Bookkeeping information typically includes the **subroutines** return address, a **reference** to the **stack frame** of the caller (also called the **dynamic link**), **saved values** of registers needed by both the **caller** and the **callee**, and various other values.

Arguments to be passed to subsequent routines lie at the top of the **frame**, where the **callee** can easily find them

Maintenence of the stack is the reponsibility of the subroutine **calling sequence**—the code executed by the caller immediately before and after the call—and of the **prologue** (code executed at the beginning) and **epilogue** (code executed at the end) of the subroutine itself.

Sometimes the **calling sequence** is ued to refer to the combined operations of the **caller**, the **prologue**, and the **epilogue**.

If a language permits recursion, static allocation of local variables is no longer an option, since the number of instances of a variable may need to exist at the same time is conceptually unbounded. The natural nesting of subroutine calls make it easy to allocate space for locals on a stack.

The location of a stack frame cannot be predicted at compile time the offsets of objects *within* a frame usually *can* be statically determined. Moreover, the compiler can arrange for a particular register, known as the **frame pointer** to always point to a known location within the frame of the current subroutine. Code that needs to access a local variable within the current frame, or an argument near the top of the calling frame can do so by adding a predetermined offset to the value in the frame pointer. Almost every processor provides a **displacement addressing** mechanism that allows this addition to be specified implicitly as part of an ordinary `load` or `store` instruction. The stack grown "downward" toward lower addresses in most language implementations. Some machines provide special `push` and `pop` instructions that assume this direction of growth. Local variables, temporaries, and bookkeeping information typically have negative offsets from the frame pointer. Arguments and returns typically have positive offsets; they reside in the caller's frame.