

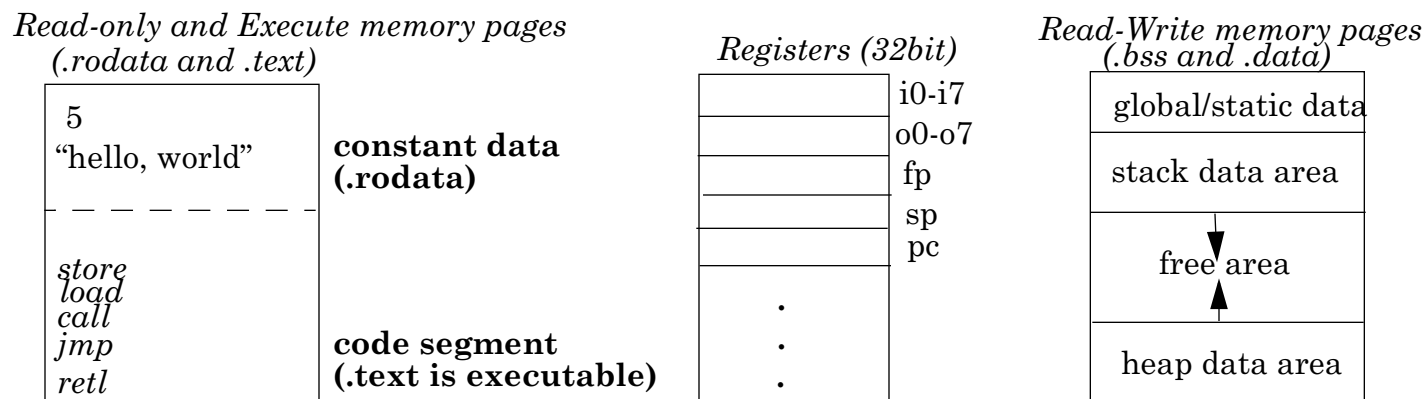
Language Storage Management

There are four data storage areas that a typical imperative programming language like C or C++ will utilize.

1. **Global** or **static** data area - persists for the entire execution of the program.
2. **Stack** or local data area - persists for the execution of a procedure or block.
3. **Heap** or dynamic data area - persists from the time allocated to the time freed, or end of program.
4. **Registers** or temporary data areas - allocated by compiler generated run-time as needed by the program.

All of the non-register data areas are pre-allocated by the operating system when a program starts to execute and are placed in *read-write* memory pages. The compiler arranges in advance the size and layout of data in the global/static area as well as on the stack. The global/static variable-to-location bindings do not change while the program runs. The stack variable-to-location bindings change as procedures are invoked and they return. The heap is available as soon a program starts running, but data is allocated under control of the program. General registers are used for fast temporary storage, special purpose registers are used for execution: program counter (pc), stack pointer (sp), frame pointer (fp).

The code for your program is placed in *read-execute* memory pages. Any constant data (such as integer constants 1,2,... and string literals, like “hello, world”) are placed in *read-only* memory pages. Note that because string literals are read-only, if you try to change them, you get a **segmentation violation**. Conceptually, a run-time execution environment looks like this. Notice that the heap and stack grow into each other:



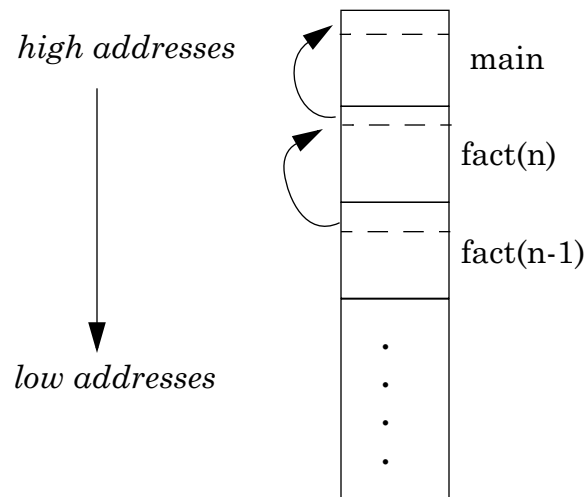
Stack Activation Frames

In most imperative language, activation frames containing procedure arguments and variables declared local to a block are allocated on the stack. The reason is that sequential programs consists of a sequence of nested procedure activations, and a LIFO data structure is the most appropriate (and efficient) representation.

Because of lexical scopes, bindings of parameter and local variable names are dynamically constructed during run-time. When a procedure is invoked, an activation record is created and parameters and local variables are then bound to locations in the activation record or to registers if the datum (e.g., integer, pointer) can fit in a register.

Activation frames are linked, such that the “top-most” frame (e.g., the one for ‘main’) is the head of a list of activation frames. Special linkage using a *frame pointer register* (fp) is used to “chain” the frames together so that the stack can be unwound when a procedure returns, or when an exception is thrown. This is called *stack unwinding*.

A stack of activation frames growing downward for recursive invocations of the factorial function, starting at main, Each frame stores the frame pointer (fp) of the previous frame.



It should be obvious that a stack of activation frames is required to implement recursion. In fact, the idea came from Algol60, which was the first imperative language to employ a stack and activation frames as a technique for implementing recursion.

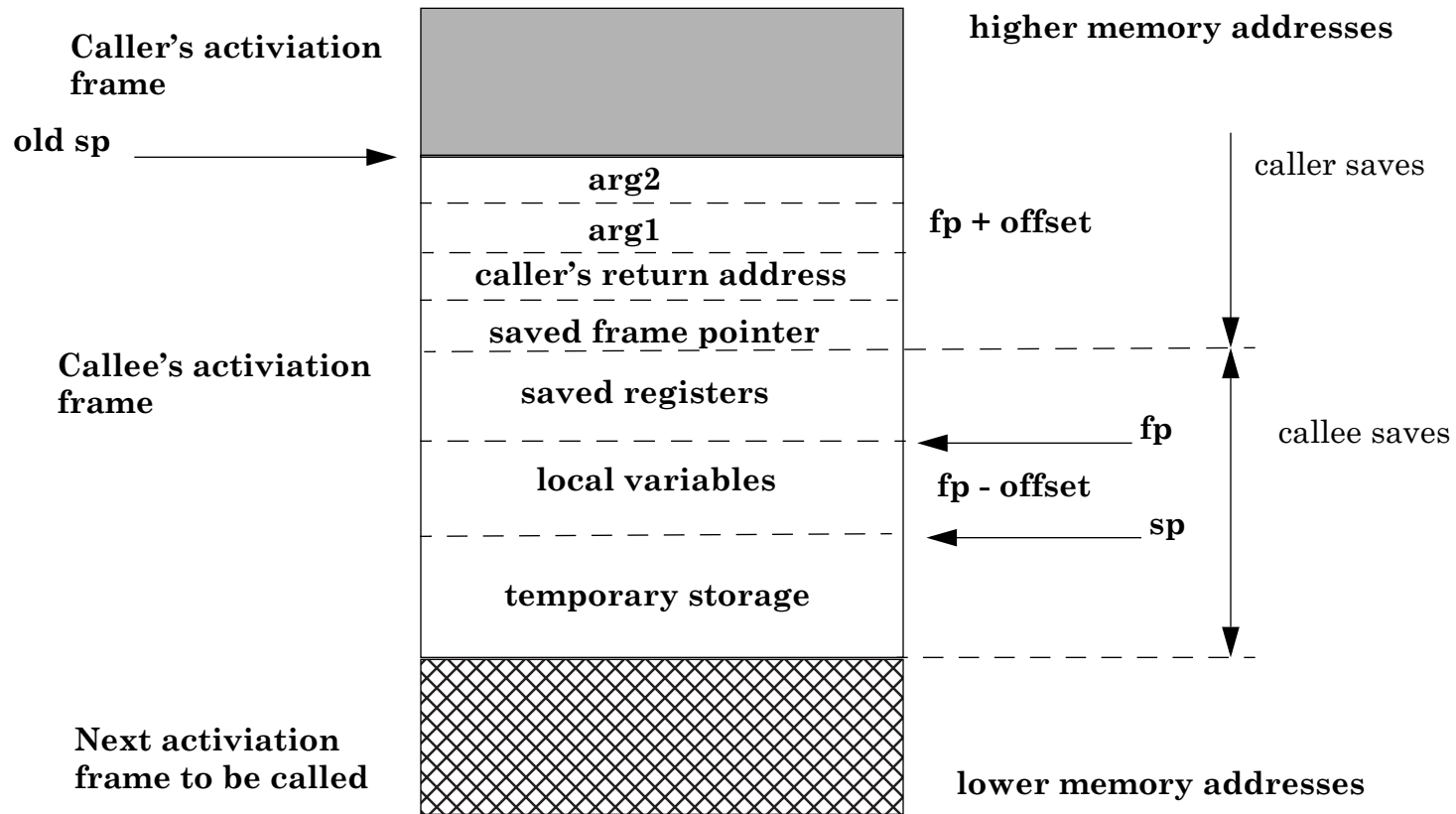
Structure of an Activation Frame

The actual memory structure of an activation frame will depend on the language, and also the target machine. However, all activation frames contain the same type of information. A special frame pointer register is used to reference locations within the frame, which contains all or some of the following:

- A *control link*, or *dynamic link*, which points back to the frame of the previous activation record on the stack. The control link is typically just the value of the frame pointer for the previous frame. When a procedure returns, the fp register
- Possibly (but not always) an *access link* or *static link*, which is a pointer to another set of variable bindings (outer scope) that are accessible to the procedure. Pascal provides this extra link because of nested procedures, C/C++ do not. In C/C++, the only variables visible within a procedure are the formal parameters, the local stack variables, and any external/static data that is declared in the global scope (recall that C/C++ do not permit nested procedures, but do permit nested scopes within a procedure scope).
- Saved *register state* of the caller, allowing the procedure to use the registers without destroying information needed by the caller. The *return address* is also saved. The return address is the address of the instruction in the caller immediately following the procedure call statement. On return from a procedure, the register state is restored, and control is transferred to the return address of the caller.
- Locations corresponding to the formal parameters, which will contain the values/references of actual arguments when the procedure is called. It is typically the caller's responsibility to push the actual arguments onto the activation frame, or they might also be passed in a register for efficiency.
- A location that will contain the result value of the procedure/function. In most language, every attempt is made to return the result in a register, rather than using the stack. However, if the result value is too large to fit in a register (or perhaps a pair of registers) then the result is placed on the stack, and it is the responsibility of the caller to take the result value off of the stack.
- Locations for local variables. There can be as many as you need, and they can take up as much space as necessary as long as the stack does not overflow. For example, it is common to allocate large character buffers on the stack (e.g., `char buf[1024]`), as well as a arrays.

Structure of an Activation Frame

An activation frame is partially constructed by the caller of a procedure and partially constructed by the called procedure. The caller will take care of saving the return address then pushing the arguments onto the stack (possibly in reverse order). The called procedure has the responsibility of saving registers and allocating space on the stack for local variables. Arguments are access by computing positive offsets from the fp, and locals are computed using negative offsets from the fp because the stack grows “downward” from high addresses to low addresses. The stack pointer (sp) points at a temporary storage area, which can be used for additional local storage:



Procedure Prologues, Epilogues, Calls, and Returns

Calling a procedure requires a “handshake” to pass arguments and transfer control from the calling procedure to the called procedure. There are 5 major phases:

1. The *procedure call* computes each argument to the procedure and passes control by branching to the address of the procedure (which in most languages is determined at compile time).
 - a) each argument is “evaluated” and put into a register or pushed onto the stack.
 - b) registers in use by the caller are saved by the callee
 - c) the return address is saved in a register or on the stack, the frame pointer is saved, and a branch instruction transfers control to the procedure’s code
2. The *prologue* is a small number of instructions that are executed on entry to the procedure. It saves registers and enables the environment of the procedure by setting the frame and stack pointers appropriately.
3. The procedure executes, possibly calling other procedures, or itself recursively, thereby repeating these steps.
4. The *epilogue* occurs at the end of the procedure and restores saved register values and the addressing environment of the caller, puts the return value (if any) into a register or onto a stack location, and returns control to the caller.
5. The code in the caller following the call to the procedure finishes restoring the execution environment and extracts the return value from the stack of return register.

NOTE: a **stack smashing attack** is an activation frame exploit that exploits the following:

1. In some operating systems, the stack memory pages are in fact marked executable, meaning that stack addresses can be sent to the instruction cache, and if those locations contain code, it will be executed.
2. During the execution of a procedure, a stack allocated array(buffer) or pointer to a function is overwritten through sloppy input size checking (read, scanf) or during some memory operations (sscanf), causing overwriting of the local variable storage locations up through the return address location in the activation frame to be overwritten. This is why this attack is called “stack smashing”
3. The fact that the stack most often grows from high to low addresses facilitates this kind of attack. Why?

Calling Conventions— Real Code Examples

The following are simple C++ procedures that illustrates the different effects of call-by-value, call-by-reference, and call-by-value using pointers to effect call-by-reference by passing addresses (l-values) by value.

```
void swap_by_val(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
void swap_by_ref(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
void swap_by_ptr(int* a, int* b) // not that a and b could be NULL.
{
    int temp = *a; // should we do a run-time check for a != NULL and b != NULL ??
    *a = *b;
    *b = temp;
}
```

```
int main()
{
    int x = 5, y = 10;
    swap_by_val (x, y);
    swap_by_ref (x, y);
    swap_by_ptr (&x, &y);
}
```

Procedure Call Setup (Sparc)

Notice the difference in how the actual arguments are computed before each procedure call. In the case of `swap_by_val`, r-values are obtained from locations on the stack and then placed in (output) registers to be passed to the activation frame of the procedure. In the other two cases, l-values are computed and passed as arguments. The 'call' instruction takes care of saving the return address and the current frame pointer.

```
! 12    swap_by_val (x, y);
        ld        [%fp-8],%l0
        ld        [%fp-12],%l1
        mov       %l0,%o0
        mov       %l1,%o1
        call     __0FLswap_by_valiTB
        nop

.L4:
! 13    swap_by_ref (x, y);
        add       %fp,-8,%l0
        add       %fp,-12,%l1
        mov       %l0,%o0
        mov       %l1,%o1
        call     __0FLswap_by_refRiTB
        nop

.L5:
! 14    swap_by_ptr (&x, &y);
        add       %fp,-8,%l0
        add       %fp,-12,%l1
        mov       %l0,%o0
        mov       %l1,%o1
        call     __0FLswap_by_ptrPiTB
        nop
```

Debugger Trace for Procedure Calls

The previous program is compiled (with `-g` flag for debugging) and the GNU debugger (gdb) is used to set *breakpoints* to allow the programmer to examine what is really going on in each case. Notice the differences, especially for reference the parameters.

```
(gdb) break main
Breakpoint 1 at 0x10458: file ref.C, line 10.
...
(gdb) run
Breakpoint 1, main () at ref.C:10
(gdb) p &x
$15 = (int *) 0xeffffbac
(gdb) p &y
$16 = (int *) 0xeffffba8
Breakpoint 2, swap_by_val (a=5, b=10) at swap.C:3
(gdb) p &a
$17 = (int *) 0xeffffb8c
(gdb) p &b
$18 = (int *) 0xeffffb90
Breakpoint 3, swap_by_ref (a=@0xeffffbac, b=@0xeffffba8) at swap.C:10
(gdb) p &a
$19 = (int *) 0xeffffbac
(gdb) p &b
$20 = (int *) 0xeffffba8
Breakpoint 4, swap_by_ptr (a=0xeffffbac, b=0xeffffba8) at swap.C:17
(gdb) p *a
$21 = 10
(gdb) p &a
$22 = (int **) 0xeffffb8c
(gdb) p &b
$23 = (int **) 0xeffffb90
```

Compiled Call-by-Reference Code (Sparc)

Every procedure has an *activation frame*, which is referenced by a special register, called the **frame pointer** (fp). All arguments and local variable for a procedure (or function) are usually found at some (even) offset from the fp register.

```

!   8  void swap_by_ref(int& a, int& b)
!   9  {
        save    %sp,-104,%sp
        st      %i1,[%fp+72]    !! a and b are really passed in register, then stored
        st      %i0,[%fp+68]    !! at stack locations in the activation frame (fp+offset)
.L9:
!   10  int temp = a;
        ld      [%fp+68],%i0    !! load the r-value of a (an l-value) into a register
        ld      [%i0+0],%i0     !! load the r-value of a's r-value into a register
        st      %i0,[%fp-8]    !! store the r-value into the fp offset for 'temp'
.L10:
!   11  a = b;
        ld      [%fp+72],%i0    !! load the r-value of b (an l-value) into a register
        ld      [%i0+0],%i1     !! load the r-value of b's r-value into a register
        ld      [%fp+68],%i0    !! load the r-value of a's r-value into a register
        st      %i1,[%i0+0]    !! store the r-value into the l-value obtained from a
.L11:
!   12  b = temp;
        ld      [%fp-8],%i1     !! load the r-value of temp into a register
        ld      [%fp+72],%i0    !! load the r-value of b (an l-value) into a register
        st      %i1,[%i0+0]    !! store the r-value of temp into the l-value given by b
.L12:
!   13  }
.L15:
        jmp     %i7+8          !! return to caller
        restore                !! insruction pipelining causes one more instruction
                                !! to execute, which restores the caller's saved registers

```

Compiled Call-by-Value with Pointers Code (Sparc)

As we have seen, call-by-value using pointers has the disadvantage of the possibility of a pointer being passed to the procedure which is NULL. Dereferencing a NULL pointer will crash our program. NOTE that the code for call-by-value using pointers is IDENTICAL to the code for call-by-reference, which is **much safer** since the compiler can determine l-values statically at compile-time, whereas pointers are determined dynamically at run-time.

```
! 15 void swap_by_ptr(int* a, int* b)
! 16 {
    save    %sp, -104, %sp
    st      %i1, [%fp+72]
    st      %i0, [%fp+68]
.L16:
! 17     int temp = *a;
    ld      [%fp+68], %10
    ld      [%10+0], %10
    st      %10, [%fp-8]
.L17:
! 18     *a = *b;
    ld      [%fp+72], %10
    ld      [%10+0], %11
    ld      [%fp+68], %10
    st      %11, [%10+0]
.L18:
! 19     *b = temp;
    ld      [%fp-8], %11
    ld      [%fp+72], %10
    st      %11, [%10+0]
.L19:
! 20 }
.L22:
    jmp     %i7+8
    restore
```

Optimized Call-by-Reference Code (Sparc)

A good optimizing compiler can usually omit the frame pointer and use only registers for arguments that will fit in a register (i.e., int, char, pointers). If there are not too many local variables, and they are the right size, they will be put into registers as well.

```

!      8          !void swap_by_ref(int& a, int& b)
!      9          !{
!     10          !  int temp = a;
/* 000000      10 */          ld      [%o0],%g1 ! volatile
!     11          !  a = b;
/* 0x0004      11 */          ld      [%o1],%g2 ! volatile
/* 0x0008          */          st      %g2,[%o0] ! volatile
/* 0x000c          */          retl
!     12          !  b = temp;
!     13          !}
/* 0x0010          */          st      %g1,[%o1] ! volatile

```

Note that the optimized code for swap_by_ptr is identical, which is what we would expect. But the call-by-reference code is safer! Why?

```

!     15          !void swap_by_ptr(int* a, int* b)
!     16          !{
!     17          !  int temp = *a;
/* 000000      17 */          ld      [%o0],%g1 ! volatile
!     18          !  *a = *b;
/* 0x0004      18 */          ld      [%o1],%g2 ! volatile
/* 0x0008          */          st      %g2,[%o0] ! volatile
/* 0x000c          */          retl
!     19          !  b = temp;
!     20          !}
/* 0x0010          */          st      %g1,[%o1] ! volatile

```

Optimized Code (Sparc)

By the way, what about the swap-by-val example that does not result in its two arguments being swapped?

```
!   1 void swap_by_val(int a, int b)
!   2 {
!       save    %sp,-104,%sp
!       st      %i1,[%fp+72]
!       st      %i0,[%fp+68]
!   3   int temp = a;
!       ld      [%fp+68],%l0
!       st      %l0,[%fp-8]
!   4   a = b;
!       ld      [%fp+72],%l0
!       st      %l0,[%fp+68]
!   5   b = temp;
!       ld      [%fp-8],%l0
!       st      %l0,[%fp+72]
!   6   }
!       jmp     %i7+8
!       restore
```

However, the optimized code just returns to the caller. The optimizer in the compiler is smart enough to figure out that the procedure doesn't have any effect, so the optimization that results is to do nothing!!

```
!   1           !void swap_by_val(int a, int b)
!   2           !{
!   3           ! int temp = a;
!   4           ! a = b;
!   5           ! b = temp;
!   6           !}
/* 0x0000      */           retl    !! do nothing, just return to the caller
/* 0x0004      */           nop      !! latent no-op instruction for pipelining
```