## Run Time Environments

Before we get into the **low-level details** of final **code generation**, we first take a look at the **layout of memory and the runtime data structures** for managing the **stack and heap**.

### Data Representation

Simple variables: are usually represented by sufficiently large memory locations to hold them:

- **characters**: 1 or 2 bytes
- **integers**: 2, 4 or 8 bytes
- **floats**: 4 to 16 bytes
- **boolean**: 1 bit (most often at least 1 full byte used)

During execution, allocation must be maintained by the generated code that is compatible with the scope and lifetime rules of the language.
Typically there are **three choices** for allocating variables and parameters:

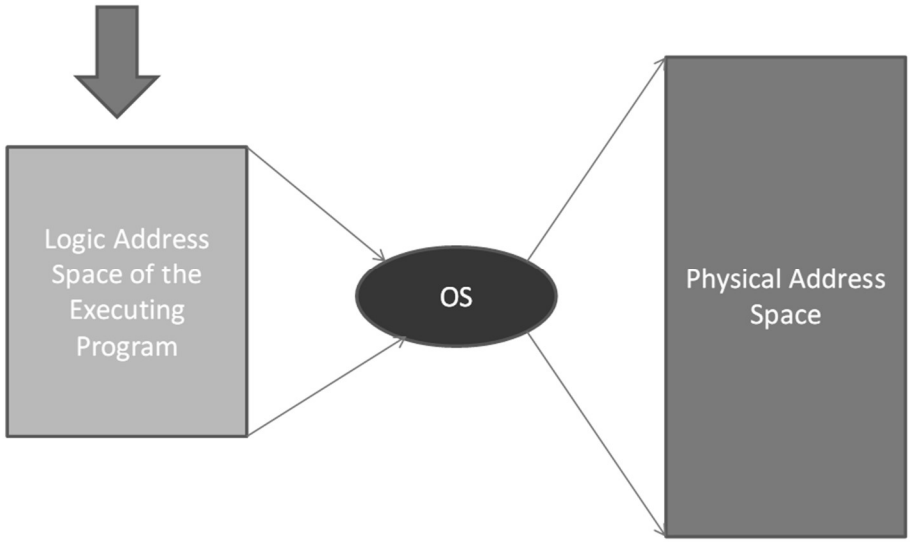
– **Assign them fixed locations in global memory ("static" allocation).**
**– Put them on the processor stack.**
**– Allocate them dynamically in memory managed by the program (the "heap").**

### Issues in Storage Organization

- Recursion
- Block structure and nesting (nested procedures).
- Parameter passing (by value, reference, name).
- Higher order procedures (procedures as parameters to other procedures).
- Dynamic Storage Management (malloc, free).

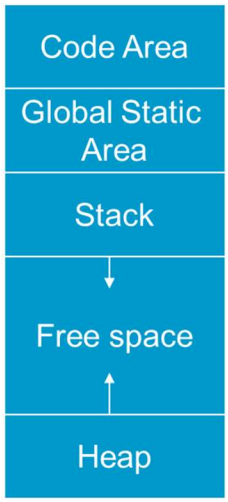
## Storage Organization

From the perspective of the compiler writer, the executing target program runs in its own **logical address space** in which each program value has a location. The management and organization of this logical address space is shared between the **compiler, operating system, and target machine**. The operating system maps the **logical addresses** into **physical addresses**, which are usually spread throughout memory.




**Compiler-writer Perspective**

The **run-time representation** of an **object program** in the logical address space consists of **data and program areas as shown in figure below**. The runtime storage organization might subdivide memory in this way.




**Storage for code and data:**

**Code Area**: Procedures, functions and methods
**Static Data Area**: "Permanent" data with statically known size.
**Stack**: Temporary Data with known lifetime (lifetime refers to the different periods of time for which a variable remains in existence)
**Heap**: Temporary Data with unknown lifetime (dynamically allocated).

We assume the **run-time storage** comes in blocks of **contiguous bytes**, where a **byte** is the **smallest unit** of addressable memory. A byte is **eight bit** and **four bytes** form a machine **word**. **Multi byte** objects are stored in consecutive bytes and given the address of the first byte.

As discussed previously, the amount of storage needed for a name is determined from its **type**. An **elementary data type**, such as a **character**, **integer**, or **float**, can be stored in an **integral number of bytes**. Storage for an aggregate type, such as an **array or structure**, must be large enough to hold all its components.

The **storage layout for data objects** is strongly influenced by the addressing **constraints of the target machine**. On many machines, instructions to add integers may expect integers to be aligned, that is, placed at an address **divisible by 4**. Although an **array of ten characters** needs only enough **bytes** to hold **ten characters**, a compiler may **allocate 12 bytes** to get the **proper alignment**, leaving **2 bytes unused**. Space left unused due to alignment considerations is referred to as **padding**. When **space is at a premium**, a compiler may pack data so that no padding is left; additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned.

## Areas (segments) of Memory

**1.** The code (often called text in OS-speak) is fixed size and unchanging (self-modifying code is long out of fashion). If there is OS support, the text could be marked execute only (or perhaps read and execute, but not write). All other areas would be marked non-executable (except for systems like lisp that execute their data).
**2.** There is likely data of **fixed size** whose need can be determined by the compiler by examining the program's structure (and not by determining the program's execution pattern). **One example is global data**. Storage for this data would be allocated in the **next area right after the code**. A key point is that since the code and this area are of fixed size that does not change during execution, they, unlike the next two areas, no need for an expansion region.
**3.** The **stack** is used for memory whose **lifetime is stack-like**. It is organized into activation records that are created as a procedure is called and destroyed when the procedure exits. It **abuts** the area of unused memory so can grow easily. Typically the **stack is stored at the highest virtual addresses and grows downward** (toward small addresses). However, it is sometimes easier in describing the

activation records and their uses to **pretend** that the addresses are increasing (so that increments are positive).
**4.** The **heap** is used for data whose **lifetime is not as easily described**. This data is allocated by the **program itself**, typically either with a language construct, such as **new**, or via a **library function call**, such as **malloc()**. It is deallocates either by another executable statement, such as a call to free(), or automatically by the system.

## Static versus Dynamic Storage Allocation

**Much (often most) data cannot be statically allocated. Either its size is not known at compile time or its lifetime is only a subset of the program's execution.**
Early versions of **FORTRAN** used only **statically allocated data**. This required that each array had a constant size specified in the program. Another consequence of supporting only static allocation was **that recursion was forbidden** (otherwise the compiler could not tell how many versions of a variable would be needed).

**Modern languages**, including newer versions of **FORTRAN**, support both **static and dynamic allocation of memory**.

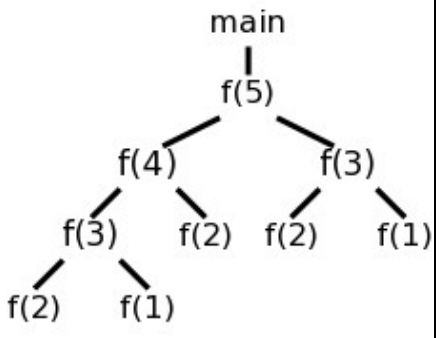
The **advantage** supporting **dynamic storage allocation** is the **increased flexibility and storage efficiency possible** (instead of declaring an array to have a size adequate for the largest data set; **just allocate what is needed**). The **advantage of static storage allocation** is that it **avoids the runtime costs** for allocation/deallocation and may permit faster code sequences for referencing the data.
An (unfortunately, all too common) error is a so-called **memory leak** where a long **running program repeated allocates memory that it fails to delete**, even after it can no longer be referenced. To **avoid memory leaks** and **ease programming**, several programming language systems employ **automatic garbage collection**. That means the runtime system itself determines when data can no longer be referenced and automatically deallocates it.

## Stack Allocation of Space

Almost all **compilers for languages that use procedures**, **functions, or methods** as units of user-defined actions manage at **least part of their run-time memory as a stack**. Each time a **procedure is called, space for its local variables is pushed onto a stack**, and when the **procedure terminates**, that **space is popped off the stack**. As we shall see, this arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.

## Activation Trees

Recall the Fibonacci sequence 1,1,2,3,5,8, ... defined by **f(1)=f(2)=1** and, for **n>2**, **f(n)=f(n-1)+f(n-2)**. Consider the function calls that result from a main program calling **f (5)**. On the **left they are shown in a linear fashion** and, on the **right, we show them in tree form.** The latter is sometimes called the activation tree or call tree.

| | |
|---|---|
| System starts main<br>**enter f(5)**<br>**enter f(4)**<br>**enter f(3)**<br>**enter f(2)**<br>**exit f(2)**<br>**enter f(1)**<br>**exit f(1)**<br>**exit f(3)**<br>**enter f(2)**<br>**exit f(2)**<br>**exit f(4)**<br>**enter f(3)**<br>**enter f(2)**<br>**exit f(2)**<br>**enter f(1)**<br>**exit f(1)**<br>**exit f(3)**<br>**exit f(5)**<br>main ends | int a[5];<br>int f (int n)<br>{<br>if (n<3) return 1;<br>return f(n-1)+f(n-2);<br>}<br>int main()<br>{<br>int i;<br>for (i=1; i<=5; i++)<br>{<br>a[i] = f(i); }<br>  <br> |

We can make the following observation about these **procedure calls**.

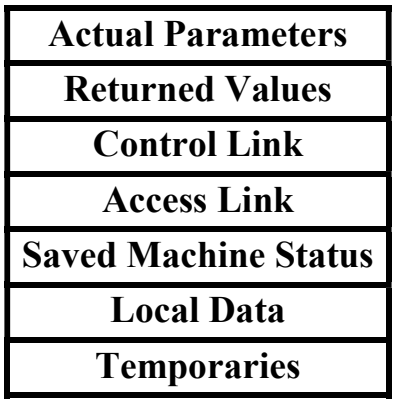
**1.** If an **activation** of procedure **p** calls procedure **q**, then that activation of **q** must **end before the activation of p can end**.
**2.** The **order of activations (procedure calls)** corresponds to a **preorder traversal of the call tree. (root, left, and right)**
**3.** The order of deactivations (**procedure returns**) corresponds to **postorder traversal of the call tree**. **(left, right, and root)**
**4.** If execution is currently in an activation corresponding to a **node N** of the activation tree, then the activations that are currently live are those corresponding to **N** and its ancestors in the tree. **These live activations were called in the order given by the root-to-N path in the tree**, and the returns will occur in the **reverse order**.

## Activation Records (ARs)

**Procedure calls and returns** are usually managed by a **run-time stack** called the **control stack**. Each **live activation has an activation record** (sometimes called a **frame**) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides. The latter activation has its record at the top of the stack.

Note that this is **memory used by the compiled program**, not by the compiler. The compiler's job is to **generate code that obtains the needed memory.**
At any point in time the number of frames on the stack is the **current depth of procedure calls**. For example, in the Fibonacci execution shown above when **f(4)** is active there are **three activation records on the control stack**.

ARs vary with the language and compiler implementation. Typical components are described with figure below. In the diagrams the stack grows down the page.

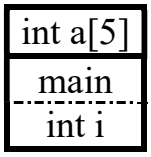



**A general activation record**

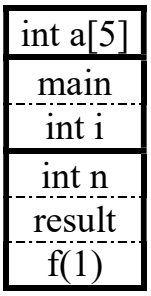
Here is a list of the kinds of data that might appear in an activation record

1. **Temporaries**. For example, recall the temporaries generated during expression evaluation. Often these can be held in **machine registers**. When that is not possible (e.g., there are more temporaries than registers), the temporary area is used.
2. **Local Data** to the procedure being activated.
3. **A Saved machine status** from the **caller**, which typically includes the return address and the machine registers. The register values are restored when control returns to the caller.
4. **The access link** may be needed to **locate data needed by the called procedure** but found elsewhere, e.g., in another activation record
5. **The control link** connects the **ARs** by pointing to the **AR of the caller**.
6. **The returned value**. This is often **placed in a register** if it is a scalar.
7. **The actual parameters** used by the **calling procedure**. Commonly, these values are not placed in the activation record but rather in **registers**, when possible, for greater efficiency. However, we show a space for them to be completely general.

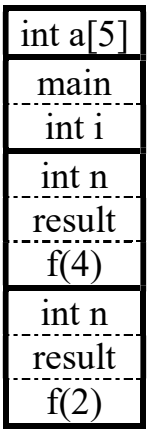
The **diagram below** on the right shows (part of) the control stack for the Fibonacci example at three points during the execution. In the **upper left** we have the initial state, we show the **global variable** a, although it is not in an activation record and actually is allocated before the program begins execution (it is statically allocated; recall that the stack and heap are each dynamically allocated). Also shown is the activation record for main, which contains storage for the local variable i.




(a) Frame for main




(b) f(1) is activated




( c) f(4), f(2) are activated

Below the initial state we see the next state when main has called **f (1)** and there are **two activation records**, one for **main and one for f**. The activation record for **f** contains space for the argument **n** and also for the result. There are **no local variables in f**.

At the far right is a later state in the execution when **f (4)** has been called by main and has in turn called **f (2)**. There are **three activation records**, one for main and **two for f**. It is these multiple activations for **f** that permits the recursive execution. **There are two locations for n and two for the result**.

## Calling Sequences

The **calling sequence**, executed when **one procedure** (the **caller**) calls another (the **callee**), allocates an **activation record** (AR) on the stack and fills in the fields. **Part of this work is done by the caller; the remainder by the callee**. Although the work is shared, the AR is called the callee's AR.
Since the procedure being called is **defined in one place**, but **normally called from many places**, we would expect to find more instances of the caller activation code than of the **callee activation code**. Thus it is wise, all else being equal, to assign as much of the work to the callee as possible.

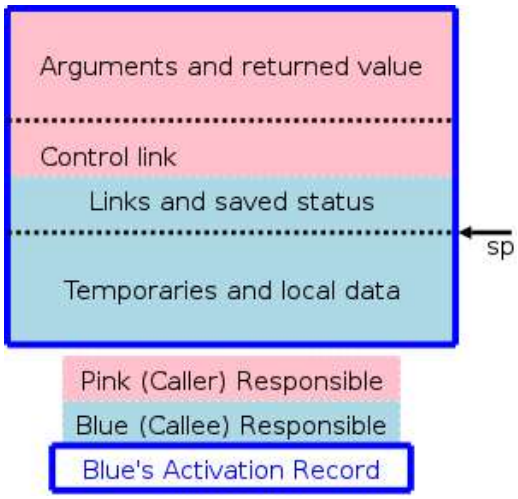
**1.** Values computed by the caller are placed before any items of size unknown by the caller. This way they can be referenced by the caller using fixed offsets. One possibility is to place values computed by the caller at the beginning of the **activation record (AR)**, i.e., near the **AR** of the caller. The number of arguments may not be the same for different calls of the same function (so called varargs, e.g.

printf () in C). However the (compiler of the) caller knows how many arguments there are so, **where pink calls blue**, the compilers knows how far the return values is from the beginning of the **blue AR**. Since this beginning of the **blue AR** is the end of the **pink AR** (or is one more depending on how you count), the caller knows (but only at **run time**) the offset of the return value location from **its own stack pointer** (**sp**, see below).

**2.** Fixed length items are placed next. Their sizes are known to the caller and callee at compile time. Examples of fixed length items include the links and the saved status.

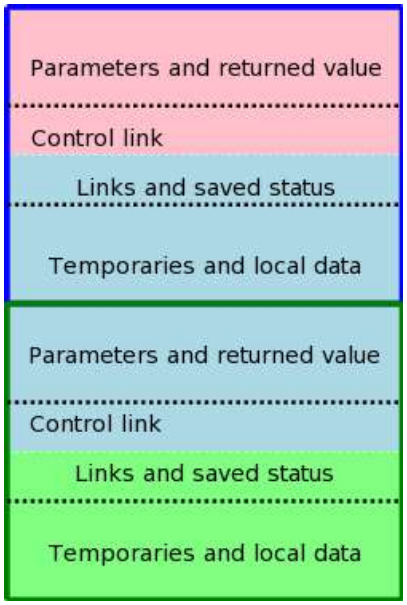
**3.** Finally come items allocated by the **callee whose size is known** only **at run-time**, e.g., **arrays whose size depends on the parameters**.

**4.** The **stack pointer sp** is between the last two. One consequence of this location is that the temporaries and local data are actually above the stack. Fixed length data can be referenced by fixed offsets (known to the intermediate code generator) from the sp.




The top picture illustrates the situation where a **pink procedure** (**the caller**) calls a **blue procedure** (**the callee**). Also shown is **Blue's AR**. Note that responsibility for this single **AR** is shared by both procedures. The picture is just an approximation: For example, the **returned value** is actually **Blue's responsibility**, although the space might well be allocated by **Pink**. Also some of the saved status, e.g., **the old sp**, **is saved by Pink**.

The bottom picture show what happens when **Blue**, **the callee**, itself **calls a green procedure** and thus **Blue is also a caller**. You can see that Blue's responsibility includes part of its AR as well as part of Green's.




**1.** The caller evaluates the arguments. (We use **arguments for the caller**, **parameters for the callee**.)
**2.** The caller stores the **return address** and the (soon-to-be-updated) **sp** in the **callee's AR**.
**3.** The **caller increments sp** so that instead of pointing into its AR, it points to the corresponding point in the **callee's AR**.
**4.** The callee saves the registers and other (system dependent) information.
**5.** The callee allocates and initializes its local data.
**6.** The callee begins execution.

**Important Registers**

- **Program Counter** (**pc**): set this for procedure calls
- **Stack Pointer** (**sp**): for activation frame info in stack-based environments
- **Base Pointer** (**bp**): similar to the stack pointer, but points to the beginning of the activation record