

Runtime Environments

Handout written by Maggie Johnson and revised by Julie Zelenski.

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
- booleans: 1 bit (most often at least 1 full byte used)

Characters, shorts, integers, booleans, and other integer-family types are usually stored in a binary polynomial form, with some mechanism such as one's or two's complement used to handle negative values. Floats and doubles are almost universally represented using the standard IEEE floating point representation, although some processor architectures support extensions.

Pointers: are usually represented as unsigned integers. The size of the pointer depends on the range of addresses on the machine. Currently almost all machines use 4 bytes to store an address, creating a 4GB addressable range. There is actually very little distinction between a pointer and a 4 byte unsigned integer. They both just store integers—the difference is whether the number is interpreted as a number or as an address. In semantic analysis, we may quibble about which can be used where, but once we start generating code we manipulate addresses fairly similarly to integers.

One dimensional arrays: are a contiguous block of elements. The size of an array is at least equal to the size of each element multiplied by the number of elements. The elements are laid out consecutively starting with the first element and working from low-memory to high. Given the base-address of the array, the compiler can generate code to compute the address of any element as an offset from the base address:

$$\&\text{arr}[i] = \text{arr} + 4 * i \quad (4\text{-byte integers; arr} = \text{base address})$$

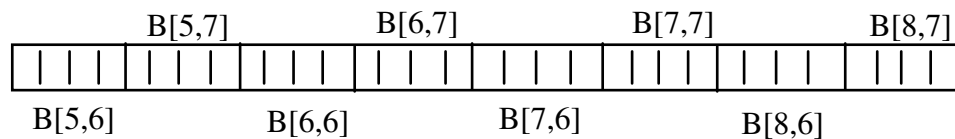
Global static arrays are stored in the data segment. Local arrays may be on the stack if the language supports such a feature. If the language supports dynamic arrays, the memory will be allocated in the runtime heap.

Multi-dimensional arrays: usually, one of two approaches is used: either a true multi-dimensional array or an array of arrays strategy.

A true multi-dimensional array is one contiguous block. In a row-major configuration (C, Pascal), the rows are stored one after another. In a column-major (Fortran), the columns are stored one after another. For a row-major array of m rows \times n columns integer array, we compute the address of an element as $\&arr[i, j] = arr + 4 * (n * i + j)$ or in a more general form for the array:

```
array[L1..U1, L2..U2] (in Pascal form)
&arr[i, j] = arr + (sizeof T) * ((i - L1) * (U2 - L2 + 1) + (j - L2))
```

For example, the array $B[5..8, 6..7]$ of integers would look like this:

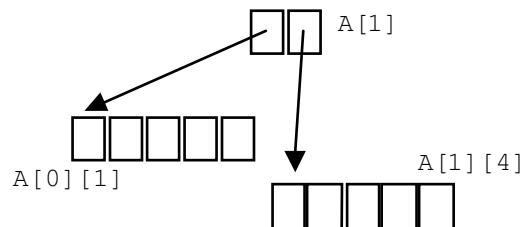


(4 bytes per integer)

$$L_1 = 5; U_2 = 7, L_2 = 6$$

$$\&B[8,6] = B + 4 * ((8-5)*(7-6+1) + (6-6)) = B + 24$$

The alternative is an array of arrays, i.e., we have an array with one pointer element for each row. This scheme is the one Decaf uses. Each element of the outer array holds the address of the memory for the elements in the corresponding row. For example, here is a diagram of the array $A[2][5]$ (indexed from 0):

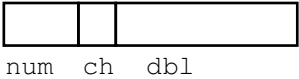


Reading the contents of array element $A[i][j]$ involves dereferencing two pointers, unlike the multi-dimensional array, which required only one. We get the location of the address of the i^{th} row by computing $A + i * \text{sizeof}(\text{pointer})$. We dereference and load that value and add $j * \text{sizeof}(\text{arrayElem})$ to get the address of the element, which then can be dereferenced to load the value of that element. The array of arrays approach requires more memory, since there is the overhead of the additional pointers, whereas the multi-dimensional array approach only requires space for the elements

themselves. Given these inefficiencies, why might it still be considered advantageous to support an array of arrays implementation?

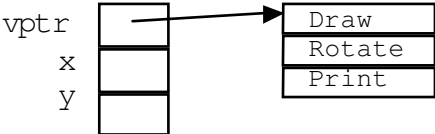
Structs: are laid out by allocating the fields sequentially in a contiguous block, working from low-memory to high. The size of a record is equal to at least the sum of the field sizes. In the example below, if integers require 4 bytes, chars 1 and doubles 8, the struct could occupy 13 bytes with the individual fields at offsets 0, 4 and 5 bytes from the base address. However, on many machines, it will actually reserve 16 bytes for this struct, because it will insert padding between the fields so that each field starts on an aligned boundary.

```
struct example {
  int num;
  char ch;
  double dbl;
};
```



Objects: are very similar to structs, where the fields are the instance variables for the class. Methods are not stored in the object itself. For dynamic dispatch, a hidden extra pointer within each object references a piece of shared class-wide information (often called the *vtable*) that provides information about the class's methods. If a language doesn't support dynamic dispatch—or if a class doesn't define any methods that require it, as would be the case for a class without virtual methods (in C++), or a class with only private and final methods (in Java), there may be no need for the vtable or vpnr element in the object's representation.

```
public class Rectangle {
  private int x, y;
  public void Draw() {}
  public void Rotate(int deg) {}
  public void Print() {}
}
```



Instructions: themselves are also encoded using bit patterns, usually in native word size. The different bits in the instruction encoding indicate such information as what type of instruction it is (load, store, multiply, etc.), what registers are being used in the operation, and so on.

Many compiled languages don't record any type information with the variables being stored. So when examining the contents of address 1000, it is not obvious whether the value should be interpreted as a short, double pointer, instruction, or whatever. Languages such as LISP or Smalltalk that support extensive runtime type identification do mark each variable with a type tag that allows the runtime system to determine the type to allow for runtime type-checking and to enable polymorphic behavior.

Storage Classes

The *scope* of a variable defines the lexical areas of a program in which the variable may be referenced. The *extent* or *lifetime* refers to the different periods of time for which a variable remains in existence.

- global*: exist throughout the lifetime of the entire program and can be referenced anywhere.
- static*: exist throughout lifetime of entire program but can only be referenced in the function (or module) in which they are declared.
- local*: (also called *automatic*) exist only for the duration of a call to the routine in which they are declared; a new variable is created each time the routine is entered (and destroyed on exit). They may be referenced only in the routine in which they are declared. Locals may have even smaller scope inside an inner block within a routine (although they might “exist” for the entirety of the routine for convenience of the compiler, they can only be referenced within their block).
- dynamic*: variables that are created during program execution; usually represented by an address held in a variable of one of the above classes. Extent is the lifetime of the program (unless explicitly returned to the system); scope is the scope of the variable used to hold its address.

Both global and static variables have a single instance that persists throughout the life of the program; these two storage classes differ only in scope. The usual implementation is to group all of these variables together in the global/static data segment of the executable. These locations are fixed at the end of the compilation process.

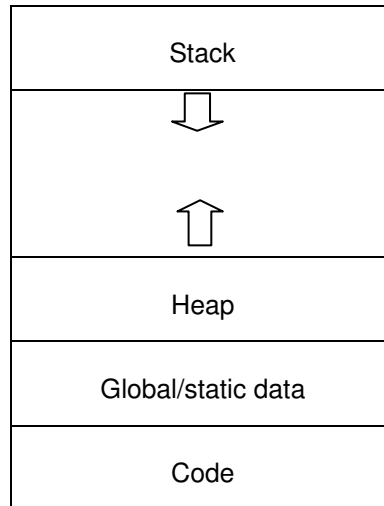
Local variables only come into existence on entry to a routine and persist until its exit. To handle these we use a runtime stack that holds the values of locals. The area of memory used to hold all the locals of a routine is called the stack frame. The stack frame for the routine currently executing will be on top of the stack. The address of the current stack frame is usually held in a machine register and is called the frame pointer.

Some of the first high-level languages (the early versions of Fortran come to mind) did not support a runtime stack of this nature. Local variables and parameters were assigned at compile-time to fixed locations, just like globals. This makes for efficient access to all variables because direct locations can be emitted into the instruction stream without requiring any arithmetic to compute offsets. But for obvious reasons, the same version of Fortran didn't support any form of recursion, direct or otherwise.

Dynamic allocation of storage during the run of a program is typically done by calling library functions (e.g., `malloc`). This storage is obtained from memory in a different

segment than the program code, global/static, or stack. Such memory is called the heap.

Here's a map depicting the *address space* of an executing program:



Runtime Stack

Each active function call has its own unique stack frame. In a stack frame (activation record) we hold the following information:

- 1) frame pointer: pointer value of the previous stack frame so we can reset the top of stack when we exit this function. This is also sometimes called the dynamic link.
- 2) static link: in languages (like Pascal but not C or Decaf) that allow nested function declarations, a function may be able to access the variables of the function(s) within which it is declared. In the static link, we hold the pointer value of the stack frame in which the current function was declared.
- 3) return address: point in the code to which we return at the end of execution of the current function.
- 4) values of arguments passed to the function and locals and temporaries used in the function.

Here is what typically happens when we call a function (every machine and language is slightly different, but the same basic steps need to be done):

Before a function call, the calling routine:

- 1) saves any necessary registers
- 2) pushes the arguments onto the stack for the target call
- 3) set up the static link (if appropriate)
- 4) pushes the return address onto the stack
- 5) jumps to the target

During a function call, the target routine:

- 1) saves any necessary registers
- 2) sets up the new frame pointer
- 3) makes space for any local variables
- 4) does its work
- 5) tears down frame pointer and static link
- 6) restores any saved registers
- 7) jumps to saved return address

After a function call, the calling routine:

- 1) removes return address and parameters from the stack
- 2) restores any saved registers
- 3) continues executing

Parameter Passing

The above description deliberately is vague on what mechanism is used for parameter-passing. Some of the common parameter-passing variants supported by various programming languages are:

Pass by value: This is the only mechanism supported by C and Decaf. Values of parameters are copied into called routine. Given the routine has its own copies, changes made to those values are not seen back in the calling function.

Pass by value-result: This interesting variant supported by languages such as Ada copies the value of the parameter into the routine, and then copies the (potentially changed) value back out. This has an effect similar to pass-by-reference, but not exactly. How could you write a test program to determine whether a language was using value-result versus reference for parameters?

Pass by reference: No copying is done, but a reference (usually implemented as a pointer) is given to the value. In addition to allowing the called

routine to change the values, it is also efficient means for passing large variables (such as structs). How does this compare to explicitly passing a pointer in C?

Pass by name: This rather unusual mechanism acts somewhat like C preprocessor macros and was introduced in Algol. Rather than evaluating the parameter value, the name or expression is actually substituted into the calling sequence and each access to the parameter in the calling body re-evaluates it. This is not particularly efficient, as you might imagine.

A few other orthogonal parameter-passing features that appear in various mainstream languages:

A read-only property such as C++ `const`, which allows you to use a more efficient pass-by-reference passing style, which however is only used for its efficiency, not for its ability to mutate the value passed.

Default parameters, which allow a routine to specify what value to use when the programmer doesn't supply an argument (i.e., calls routine with fewer than all parameters).

Position-independent parameters, ala Ada and LISP, allow parameters to be tagged with names rather than requiring them to be listed in some enforced arbitrary ordering.

Memory management

Typically, the program requests memory from the operating system, which usually returns it in pages. A page is a fairly large chunk usually around the size 4-8 KB, sometimes larger. The program then divides up this memory on its own. Quite a bit of research has been done what makes a fast allocator that minimizes fragmentation on a page. The system allocator quite likely is very smart, and keeps track of how much memory is allocated per pointer and so on.

```
a = malloc(12); // give me 12 bytes
a = b;         // oops, lost it!
```

Languages such as ML and Java have garbage collection, where the programmer doesn't have to manually free dynamically allocated memory. The runtime environment detects if an allocated chunk of memory can no longer be referenced by the program and reclaims the memory automatically. There are several methods for implementing garbage collection. The two most common are reference counting and mark and sweep.

Automatic storage management is very convenient for the programmer, but sometimes causes problems. Why isn't garbage collection a panacea for memory management? What semantic rules are needed for a garbage-collected language?

Bibliography

- A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.
- J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.
- J.P. Hayes, Computer Architecture and Organization. New York, NY: McGraw-Hill, 1988.
- A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.
- G.M.Schneider, J. Gersting, An Invitation to Computer Science, New York: West, 1995.
- J. Wakerly, Microcomputer Architecture and Programming. New York, NY: Wiley, 1989.