10

STORAGE MANAGEMENT ISSUES

Regardless of type or lifetime, values must be stored in computer memory during their lifetimes. The purpose of this chapter is to address the issues of designing and implementing storage schemes.

10.1 COMPUTER MEMORY

Each manufacturer has its own detailed implementation; most present a von Neumann architecture to the programmer (Figure 10.1). Memory tends to be passive; information is stored at and retrieved from locations determined external to the memory. Again using the CRUD acronym to guide development, the active agent must

> Create: a place in memory that is not shared by any other storage unit
> Read: values by copying and not destroying content
> Update: content without interfering with other storage units
> Destroy: the content so that the space can be reused

Most commonly available computers can address memory on the byte (8-bit) level, although word (32-bit) or double precision (64-bit) is not unknown. For our discussion here, let's define *primitive storage unit* as the number of bits the memory transfers (either read or write).

Any element that can be represented in a single primitive storage unit is easy to control; however, almost all data, including primitive data types such as integers, take more than one primitive storage unit. Virtually all operating systems provide primitives that `allocate` and `deallocate` memory. In general, `allocate` has one argument, the number of primitive storage units requested and returns a memory address. To return the memory to the free pool, `deallocate` takes a previous `allocated` memory address

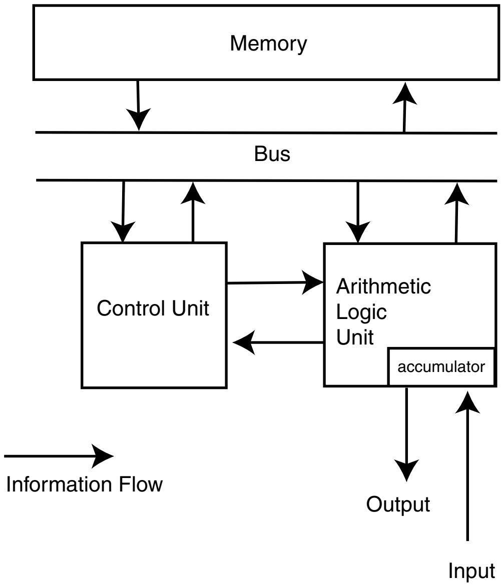

Figure 10.1 Scheme of von Neumann Architecture

and that area of memory is released. Therefore, the CRUD for memory is as follows.

- Create is a call to `allocate`.
- Read is the *dereferencing* of a pointer.
- Update is assigning new information on a pointer.
- Destroy is a call to `deallocate`.

Understanding this is the key to understanding space issues in compilers. For every type, the compiler must

1. Determine the number of primitive storage units required to hold the information in the type
2. Provide for the allocation of the space before it is needed
3. Maintain access and update control of the space during its lifetime
4. Deallocate the space at the end of its lifetime

10.2 DETERMINING THE SIZE REQUIRED

Once again we turn to an example language to explore the issues; in this case, C. In C there are three basic classes of data: primitive data types defined by the compiler, arrays, and structures. C provides a function `sizeof(type)` that returns the size in bytes of its operand. Whether the result of `sizeof` is `unsigned int` or `unsigned long` is implementation defined. The `stdlib.h` header file and `sizeof` are defined consistently in order to declare `malloc` correctly.

CASE 52. DESIGN AN ALGORITHM TO CALCULATE SIZE

The purpose of this case is to develop an understanding of how `sizeof` is defined. As a hint, `sizeof` is defined by structural induction on the type construction expressions in C.

Before reading on, define your version of `sizeof` using structural induction on the type expressions.

10.2.1 Compiler-Defined Primitive Data

Structural induction is recursion. The base cases for the structural induction must be the primitive data types. Use the program shown in Figure 10.2 to explore the values for a computer at your disposal.

10.2.2 Space for Structures

Although arrays came first historically, determining an algorithm for space should consider structures next. For this chapter, *structures* are data structures that have heterogeneous data components. In C, structures are `structs` and `unions`, whereas in C++ and Java we have `structs`, `unions`, and `classes`. Object-oriented languages raise another issue: how to deal with *inheritance*; we'll not address that here.

Turning again to Figure 10.2, consider the answer given for `struct a`: 16 bytes. How can that be?

CASE 53. WHY DOES `SIZEOF` COMPUTE 16 AND NOT 13?

Using test cases you develop, write an explanation as to why `struct a` is 16 bytes and not 13. Be very cautious—remember, `malloc` is involved as well.

Once you have satisfied yourself that you have the correct size and alignment information, define a recursive function that computes the size of structures.

10.2.3 Arrays

Arrays are comprised of elements. Arrays differ from structures in that all elements of an array are the same size. This requirement is one of the

```
int prog(int x) {
printf"%d"}xint prog(int y)  {
printf("%d"}y);
struct a {
  char c;
  int i;
  double d;
};

union b {
  char c;
  int i;
  double d;
};

int main(int argc, char* argv[] ) {
 printf( "sizeof(char)\t%d\n", sizeof(char));
 printf( "sizeof(int)\t%d\n", sizeof(int));
 printf( "sizeof(int*)\t%d\n", sizeof(int*));
 printf( "sizeof(double)\t%d\n", sizeof(double));
 printf( "sizeof(struct a)\t%d\n", sizeof(struct a));
 printf( "sizeof(union b)\t%d\n", sizeof(union b));
 printf( "sizeof(union b [5])\t%d\n",
                          sizeof(union b [5]));
 printf( "sizeof(union b [5][6])\t%d\n",
                          sizeof(union b [5][6]));
}
```

Figure 10.2 Test Program in C for `sizeof`

justifications for the concept of *variant records* or *unions*. Variant records can have multiple internal organizations but the size of the variant record is fixed.

The design issues with arrays is developing an algorithm to find an arbitrary element of the array. The basic approach is to develop an *addressing polynomial*. The basic model is that the beginning address of an element is computed from some known point, or *base*, plus an *offset* to the beginning of the element. Because all elements are the same size, we have all the information we need.

10.2.3.1 One-Dimensional Array Addressing Polynomials

For a one-dimensional array, the addressing polynomial is easy. Consider the C declaration

```
int array[27];
```

What is the address of the thirteenth element? It is instructive to "count on our fingers," because this is the basis of the more complicated situations.

Element Number	*Offset*
First	`0`
Second	`sizeof(int)`
Third	`2*sizeof(int)`
...	...
Thirteenth	`12*sizeof(int)`

The language design issue that this illustrates is the question, "Is the first element numbered '0' or '1'?" Mathematical practice (Fortran, for example) specifies that the first element is 1; C practice is that the first element is zero. Why did C developers choose zero? Speed. The upshot of this is that there are two possible addressing polynomials:

Label	*Formula*
Zero-origin	`AP(base, number,size) = base + n*size`
Unit-origin	`AP(base, number,size) = base + (n-1)*size`

10.2.3.2 One-Dimensional Arrays with Nonstandard Origins

Many problems are naturally coded using arrays that do not start at their "natural" origin of zero or one. As an example, consider this simple problem:

> Read in a list of pairs of numbers, where the first number is a year between 1990 and 2003 and the second is a dollar amount. Compute the average dollar amounts for each year.

The natural data structure would be a floating point vector (one-dimensional array) with the origin at 1990 and the last index at 2003. Following conventions used by some languages, we would note this *range* as 1990..2003.

CASE 54. DESIGN AN ADDRESSING POLYNOMIAL FOR ARRAYS

Derive an addressing polynomial expression for the arbitrary case where the lowest element is numbered l and the highest numbered element is h.

10.2.4 Multidimensional Arrays

Many problems use higher (than one) dimensional arrays. The general case has dimensions that are denoted $m..n$ and each dimension could have its own lower and upper bounds; for the ith dimension, denote these bounds as l_i and h_i.

Even at the two-dimensional case there is a choice: there are two ways to linearize the four elements of a 2×2 array. The critical insight comes from writing out the two possibilities.

CASE 55. DEVELOP AN EXTENDED ADDRESSING POLYNOMIAL

Complete the following exercise:

1. Draw a 2×2 matrix as a square array.
2. Label each element of the array with the index pair.
3. Linearize in two different ways. **Hint:** The first and last must be the same for each ordering.

How should we interpret these drawings? If it is not clear, construct other examples. Ultimately you should convince yourself that in one case the row index changes more slowly and in the other the column index changes more slowly. We call the first the *row major ordering* and the second the *column major ordering*. Row major ordering is more common but Fortran is column major.

CASE 56. GENERAL TWO-DIMENSIONAL ARRAY ADDRESSING POLYNOMIAL

Develop the addressing polynomial for the general case of two-dimensional arrays.

10.3 RUNTIME STORAGE MANAGEMENT

Runtime storage management implements scoping, allocating, and deallocating local storage. The three concepts are tightly coupled.

10.3.1 Scoping as a Visibility Function

The fundamental concept in scoping is to answer the question, "When can a data object be created, retrieved, updated, and destroyed?" The λ-calculus provides the prototypical rules for scoping in programming languages (not to mention logic and other areas of mathematics). The so-called *α-conversion rule* states that the name of bound variables is not important; that as long as I consistently rewrite the names, I will get the same result when I evaluate an expression.

An easy example is the C programs shown in Figures 10.2 and 10.3; when compiled and run, these two provide identical results. As trivial as this may seem, it is a fundamental concept in compiling functions with local variables. Consider now the C program in Figure 10.3. Although this may not seem like a program someone would write, it demonstrates a point: the y on lines (5) to (9) is different from the y on lines (1) to (4) and (10). How should we model this? It is also clear that the same variable z is used in both sections of code. How do we model this?

We will use a visibility function to model the behavior. The visibility function considers the runtime versus definitions. You can think of this as a graph with time on the abscissa and variable names on the ordinate axes (see Figure 10.4).

10.3.2 Stack Frame

A *stack frame* or *activation record* is a data structure used to create temporary storage for data and saved state in functions. The word *stack* gives away the secret: the stack frames are pushed onto the *activation stack*

```
int main(void) {
(1)   int y;
(2)   int z = 10;
(3)   for( y=0; y<=5; y++)
(4)     printf("%d ",y+z);
(5)     {
(6)     double y;
(7)     for( y=0; y<=5; y++)
(8)       printf("%f ",y+z);
(9)     }
(10)  printf("%d\n",y+z);
}
```

Figure 10.3 Simple C Program Illustrating Visibility

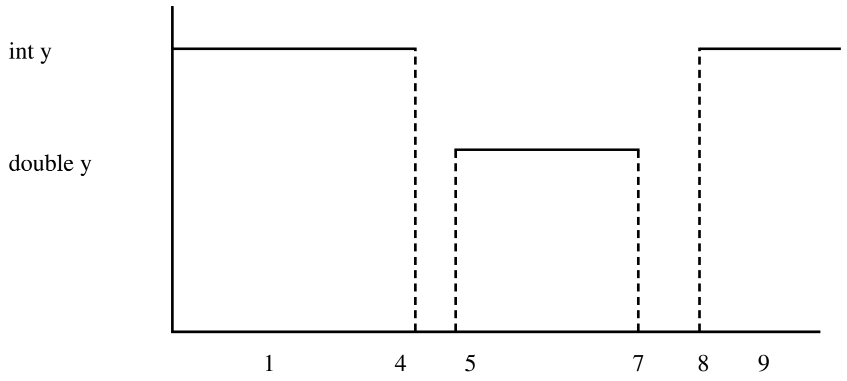

Figure 10.4 Illustration of Visibility (The graph shows statement execution on the abscissa and visibility on the ordinate axis. Solid lines indicate when a variable is visible.)

(hence the synonym *activation record*). The term *stack frame* is somewhat misleading in one sense: that the entire frame exists on the stack all at once; this is not necessarily so because this can waste stack space.

CASE 57. STACK FRAME MANIPULATION

Develop a function vocabulary to manipulate stack frames and the activation stack. This vocabulary is necessary for compiling functions and local variables to functions.

The stack frame will have space for all user-defined local variables, but there may be many other frame slots used by the compiler to save such things as temporary variables generated during expression compilation and administrative information.

From the example program, Figure 10.3, it is clear that the activation stack cannot be a *true* stack because we can see into stack frames other than the top. This indicates a list structure with a stack structure imposed. The performance issue is that we cannot afford to search for the correct stack frame.

CASE 58. LISTING ACTIVATION RECORDS

Expand the vocabulary developed above to incorporate the list requirement.