14 The executing program, linking, loading

Addressing data

There are four main categories of data, from an addressing point of view. These are

static data

dynamic storage in a stack

dynamic storage in a heap

temporary storage in a register, on a stack, in a stack or in a heap

Static data storage

Static data is stored relative to a fixed base address, which remains unchanged throughout the execution of a program. Thus all references to such an item whether from different calls of a procedure or different instances of a class will find the same location. It follows that values stored in static locations are not lost between procedure calls.

In some languages, such as Fortran and Basic, all data is stored statically. In others, such as C, the user may specify that any data item be static, but only those declared globally are static by default. In yet others, such as Pascal, only data items declared at the global level are effectively static and there is no explicit static specifier. Persistent languages, such as PS-Algol, take the idea of static storage one step further and allow values to persist in locations (albeit in the heap) across executions of a program.

Static data items have been the subject of much debate in the language design community over the years. Once they are available they can introduce some interesting problems when mixed with more dynamic elements of the language. Thus, Algol 60 allowed static variables, including arrays, to be declared at any textual level. At the same time arrays were allowed to specify their bounds as variables declared in an enclosing scope (which included parameters for procedures). This led to the possibility (unforeseen) of static arrays with dynamically determined extent. Two blocks could refer to what should be the same location, assuming both the size and bounds of each dimension to be different.

For the compiler, an assumption must be made that the start address of the static area will be available at the start of execution. In crude operating systems and embedded code a fixed address may be used. Such data is *non-relocatable*. In others adresses are only set up at run time and, since the code can run with its data at any address in memory it is known as *relocatable data*.

In most operating systems a piece of software called a *loader* is responsible for plugging the address used into an instruction which the compiler requests, into a location addressable relative to a known address in a register or for loading it into a known register. Such requests must be stored in the *object file* for interpretation by the loader when the program is run.

Dynamic storage in a stack

Stacks may have two uses. Here we refer to their rôle in context switching and calling. When a procedure is called it needs to have access to unique copies of any non-static variables declared

in its scope. This is usually done for straightforward procedural languages by pushing the required data locations onto the top of the runtime program stack. The area at the top of the stack reserved in this way for a procedure is called its *stack frame*. When the program returns from that call these locations are popped again.

In lecture 3 we will look in some detail at calling conventions and the runtime stack. Here we note that, as long as the entry and exit sequences for the procedure operate properly, local variables (usually including parameters) can be stored within the stack frame and referenced relative to its start or, in some cases, parameters may lie one side of the stack base register while locals lie the other. See lecture 3.

As long as the entry and exit sequences associated with procedure calling work correctly addressing of local variables is relatively simple. There are at least two exceptions. In Fortran, where there is no concept of recursion and all variables are static. The compiler may choose to use the stack frame for locals since this saves total space unless all procedures (called subroutines) are somehow called at once. Allocating them statically, however, may result in faster calling. In Pascal there is no separate global declaration list but the entry point is a procedure called Program. Some compilers choose to treat declarations at this first level as statics, as mentioned when discussing externals below. This can have unfortunate effects on the size of unloaded object files.

Dynamic storage in a heap

It is generally considered useful to be able to create an arbitrary number of copies of certain data items, depending on the data read in by a program when running. Linked lists and other dynamic data structures depend on use of such dynamic memory. This is allocated by a part of the runtime system known as the heap manager.

We should all know by now that new heap data locations are typically allocated space by a function call, which returns a pointer or reference. In C malloc and calloc do this. In Pascal new does the same. This pointer may then be stored as a static, stack or heap item. If no pointer is kept this space is no longer accessible from the program and is lost to it, a phenomenon known as *memory leakage*. Eventually memory leakage will use up all the space allocated for the heap unless something can be done to prevent it.

There are two solutions to memory leakage. In C, Pascal and C++ the programmer is required to return the space before the pointer is lost. Thus at any point where a pointer is assigned a new value or is lost due to procedure termination or object deletion the programmer must use the appropriate call to return the space, free in C or Pascal, delete in C++. Before doing so, however, a second condition must be satisfied, i.e. that no other pointer variable remains pointing to the same location. If this is not true a second problem, the *dangling pointer*, is caused. Then it is possible for the space to be reallocated by the heap manager to a later call for space, where it now has a different meaning. Thus two pointer variables in the program seek to use the same location for different purposes.

This explicit memory management strategy has the advantages that:

- freeing of memory is triggered exactly when needed and no checking is done that is not built in by the programmer and visible in the source of her program;
- the heap manager is a simple set of functions, usually simply maintaining a pool of space not used so far and a *free list* of returned areas within the pool allocated to date. A simple improvement is to detect when a returned area is contiguous with the unused pool and/or an item on the free list and to merge it in these cases, creating larger areas for reallocation.

The heap manager avoids awkward pauses and lost processing time, both of which are associated with the alternative, i.e. *garbage collection.*

A garbage collector is an additional part of the runtime system which is invoked when the heap manager has allocated all the space in the heap. It tries to find space with no remaining active references, which have occured due to memory leakage, and return them to the free space part of the heap. This requires, as we shall see, access to considerable information about data structures and current values in the program.

A garbage collector eliminates the phenomenon of memory leakage. It can also be used to replace altogether the explicit return of space to the heap manager and so eliminate dangling pointer errors. This is strictly only true if explicit pointer arithmetic is disallowed.

A full blown garbage collector will rquire several passes over the heap. Typically these will:

1. Start from all pointers currently stored in the program, including any temporary values in registers etc. and find the objects to which they point in the heap, marking these as "in use" objects. (This requires that all objects have a spare pointer location.)

2. Repeat step 1 recursively for active pointers within the objects located, computing the closure of active pointer chaining. This requires access to information about the internal structure of all objects (stored in *prototypes* in SIMULA implementations).

3. Compact all "in use" objects to the start of the heap area, eliminating dead space. As each object is moved, all pointers to it must be updated to its new location.

4. If necessary in the language the freed space at the end of the heap may have to be zeroed or set to a special unassigned pattern.

Surprisingly algorithms for this and similar approaches exist and are in use in Stamdard ML, most Lisp versions and SIMULA. Many optimisations are possible to the crude approach outlined above. Much debate has looked at approaches such as *on the fly* garbage collection, where the garbage collector is invoked briefly at times where a dealy is not crucial or when a I/O or other long delay is taking place.

The problems of garbage collection are all concerned with the long, unpredictable delays they are seen to provoke. These may be improved by tuning the heap size and many compilers allow users to request a desired amount of working memory. The relationship is not as simple as it seems, however. In CPU terms increasing the heap size may reduce the number of garbage collections, while making each take longer. This may reduce overall CPU used but may not. PAthological cases may easily be found. Worse, however, is the effect of paging demands in virtual memory systems, where the repeated traversal of the heap may slow down execution while CPU usage improves. In such cases a smaller heap may result in faster execution.

Temporary storage

As well as user defined data items, executing programs inevitably require temporary values to be generated implicitly during expression evaluation (including implicit adress calculations when accessing structured and remote data). These are stored in registers on most modern machines, but some architectures have too few or even no registers. It is usually necessary to have a means of storing extra temporaries in an extension to local space in the stack frame or on the top of the stack. This is combined with a register tracking scheme to make best use of registers. register allocation is considered towards the end of this course.

A more sophisticated use of registers is possible where a variable is identified as capable of replacement by a register held value. This may be done for the whole of a variable's use

or for a period following loading from memory the current value and ending with the storing back of the value. Techniques for finding such cases are also dealt with under the techniques for data flow analysis later in this course. C, of course, allows programmers to suggest that variables be treated in this way, but compilers can typically do a better job on their own.

External variables and libraries

A final consideration in thinking about memory allocation comes where *external* items are allowed. In ISO Pascal, the entire program was assumed to be held in one source file. In such cases all variables, except those in the standard libraries, which did not have to be declared anyway, were declared in that file and the compiler would have full control of their storage. Standard library items, like the record asociated with standard input and output files would be known in advance and be handled as if declared at a suitable point in the source code, allocated storage as best suited the implementation.

Most languages, however, have always supported the building programs from groups of source files. Most of these would typically contain self contained or internally mutually dependent functions. Usually the language definition either says that the effect of combining such files should be the same as having their contents declared together in one file or defines some special *module* or *package* interface through which a client file can see the contents of a library.

For the compiler implementor these separate compilation features pose a problem. The usual solution is to require either a *linker* or a *loader* to match up names defined within library files as being externally visible with names referring to these in client files. Various conventions are then used to store addresses of external items in memory locations, registers or operand fields in the code, creating either a combined object file (where a linker is used) or a combined executable image (where a loader does the job). The required information is added to the linker or loader tables in the initial, unlinked object files output by the compiler.

It is important to see that only statically allocated data items and procedures can be made externally visible.

Mapping languages to storage

Pascal

"Pure" ISO Pascal uses only the stack, for declared data; the heap, for dynamically generated data and an appropriate mixture of inline literals and read only constants.

In practice, some stack items, such as particularly large arrays, may be generated in the heap and only a pointer kept in the stack.

Where initialised globals are allowed, e.g. in ICL Pascal, these must be treated as initialised statics. Similarly, external data must be kept in a static area, as it may be addressed from another module, even when its own main program is not active.

C

Automatic variables in C are normally kept in the stack. Some, however, may be flagged as register variables and, if possible, should be kept loaded. It may be sensible to allocate stack space for register variables, even when they are loaded most of the time. This will provide a place to dump their registers if they are needed in evaluating an expression or if they are hazarded by procedure calls.

Static and external variables must be allocated to static areas.

C has initialisation of both static and automatic variables. Static initialisation is usually done at compile time, but code must be generated as part of the function entry sequence to initialise automatic variables.

The C heap is not part of the language. Access is via pointers, which is a special case of store mapping.

Fortran

Fortran has parameters, locals and common data. Strictly speaking parameters and locals are allocated statically. As there is no recursion (direct or indirect) this causes no problems. In fact, many Fortran compilers allocate them on the stack for memory efficiency.

The main problem with allocating Fortran parameters on the stack is that more than one entry point is allowed to the same sub-routine and these may have different parameters. Thus values set by an earlier call which are not reset in a subsequent call with fewer parameters, should retain their earlier value. As all Fortran parameters are by reference, this leads to fun.

Fortran common blocks are unique to the language. They may be initialised, but only in one of the sub-routines or the program sharing them. The place where they are initialised is the definition of the block with that name. No other reference to that block may be larger. There is one unnamed common area, the block common.

The linker or loader must map all references to a particular named common (or to the block common) onto the same static area. It must check their sizes for consistency.

Access to commons is typically through a pointer relocated by the linker or loader.