8 Pointers

8.1 Pointer types

A variable is nothing more than a convenient notation for referring to the address of a memory location. A variable name is static and determined at compile-time: distinct names refer to distinct locations and there is no way to "compute" a name except in limited contexts such as array indexing. A value of *pointer type* is an address; a pointer variable contains the address of another variable or constant. The object pointed *to* is called the *designated object*. Pointers are used for computing with the address of a memory location rather than with the contents of the location.

The following example:

C

```
int i = 4;
int *ptr = &i;
```

produces the result shown in Figure 8.1. ptr is itself a variable with its own memory location (284), but its contents are the address (320) of another variable i.

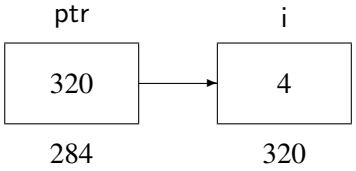

Figure 8.1: Pointer variable and designated variable

The syntax of the declaration is rather confusing because the asterisk "*" refers to the type int and not to the variable ptr. The correct way of reading the declaration is: "ptr is of type pointer to int". The unary operator "&" returns the address of its following operand.[1]

The value of the variable i can of course be accessed by using its name in an expression such as i+1, but it can also be accessed by *dereferencing* the pointer, using the syntax *ptr. When you dereference a pointer you are asking not for the contents of the pointer variable ptr, but the contents of the memory location pointed *to* by the address in ptr, that is for the designated object.

[1] Do not confuse this operator with the use of "&" to indicate a reference parameter in C++.

Typed pointers

The addresses in the example are written like integers, but an address is *not* an integer. The form of an address will depend on the computer architecture. For example, the Intel 8086 computer uses two 16-bit words which are combined to form a 20-bit address. Nevertheless, it is a reasonable assumption that all pointers are represented the same way.

In programming, it is more useful and safer to use typed pointers that are declared to point to a specific type, such as int in the example above. The designated object *ptr is required to be an integer, and after dereferencing it can be used in any context in which an integer is required such as:

C

```
int a[10];
a[*ptr] = a[(*ptr)+5];      /* Dereference and index */
a[i] = 2 * *ptr;            /* Dereference and multiply */
```

It is important to distinguish between a pointer variable and the designated object, and to be very careful when assigning or comparing pointers:

C

```
int i1 = 10;
int i2 = 20;
int *ptr1 = &i1;            /* ptr1 points to i1 */
int *ptr2 = &i2;            /* ptr2 points to i2 */

*ptr1 = *ptr2;              /* Both variables have same value */
if (ptr1 == ptr2) ...       /* False, different pointers */
if (*ptr1 == *ptr2) ...     /* True, designated objects equal */
ptr1 = ptr2;                /* Both point to i2 */
```

Figure 8.2(a) shows the variables after the first assignment statement: dereferencing is used so designated objects are assigned and i1 receives the value 20. After executing the second assignment statement which assigns pointers, not designated objects, the variable i1 is no longer accessible through a pointer as shown in Figure 8.2(b).

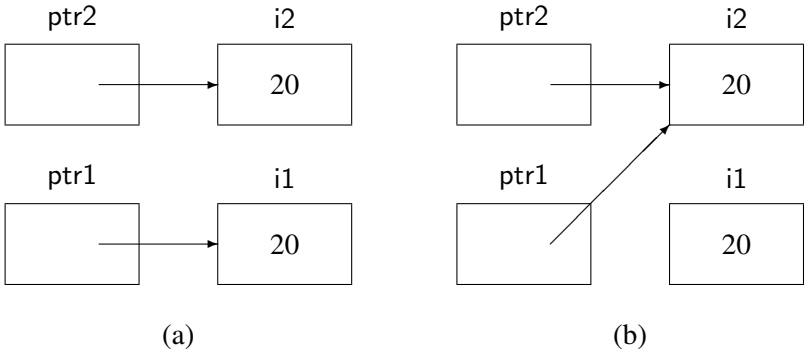

Figure 8.2: Pointer assignment

It is also important to be aware of the difference between a constant pointer and a pointer to a constant designated object. Making a pointer constant does not protect the designated object from modification:

C

```
int i1, i2;
int * const p1 = &i1;       /* Constant pointer */
const int * p2 = &i1;       /* Pointer to constant */
const int * const p3 = &i1;/* Constant pointer to constant */

p1 = &i2;                   /* Error, pointer constant */
*p1 = 5;                    /* OK, designated obj. not constant */
p2 = &i2;                   /* OK, pointer not constant */
*p2 = 5;                    /* Error, designated obj. constant */
p3 = &i2;                   /* Error, pointer constant */
*p3 = 5;                    /* Error, designated obj. constant */
```

In C, a pointer to void is an untyped pointer. Any pointer can be implicitly converted to a pointer to void and conversely, though mixing typed pointer assignments will usually get you a warning message. Fortunately, C++ is much more careful about type checking. Typed pointers can be implicitly converted to void pointers but not conversely:

C

```
void   *void_ptr;           /* Untyped pointer */
int    *int_ptr;            /* Typed pointer */
char   *char_ptr;           /* Typed pointer */

void_ptr = int_ptr;         /* OK */
char_ptr = void_ptr;        /* OK in C, error in C++ */
char_ptr = int_ptr;         /* Warning in C, error in C++ */
```

Since no type checking in done in C, an arbitrary expression may be assigned to a pointer. There is no guarantee that the designated object is of the expected type; in fact, the value of a pointer might not even be an address assigned to the program. At best, this will cause the program to crash because of an addressing fault and you will get an appropriate message from the operating system. At worst, this can cause the operating system data to become corrupted. Mistakes with pointers are very difficult to debug because the absolute addresses that the debugger displays are hard to understand. The solution is stronger type checking on pointers as in Ada and C++.

Syntax

We depart from our usual custom of glossing over syntax because the syntax for dereferencing can be confusing and must be clearly understood. Dereferencing of pointers, indexing of arrays and selection within records are the means by which data within a data structure are accessed. Pascal has the clearest syntax because each of these three operations is denoted by a separate symbol which is written after the variable. In this example, Ptr is declared as a pointer to an array of records with an integer field:

Pascal

```
type Rec_Type =
    record
        Field: Integer;
    end;
type Array_Type = array[1..100] of Rec_Type;
type Ptr_Type = ↑Array_Type;

Ptr: Ptr_Type;
```

Then using the symbol (↑) which denotes dereferencing, each addition of a symbol goes one step deeper in the decomposition of the data structure:

```
Ptr                 (* Pointer to array of records with integer field *)
Ptr↑                (* Array of records with integer field *)
Ptr↑[78]            (* Record with integer field *)
Ptr↑[78].Field      (* Integer *)
```

In C, the dereferencing symbol (*) is a prefix operator so that the above example would be written:

C

```
typedef struct {
    int field;
} Rec_Type;
typedef Rec_Type Array_Type[100];

Array_Type *ptr;

ptr                 /* Pointer to array of records with integer field */
*ptr                /* Array of records with integer field */
(*ptr)[78]          /* Record with integer field */
(*ptr)[78].field    /* Integer */
```

where the parentheses are needed because array indexing has higher precedence than pointer dereferencing. In a complicated data structure it can be confusing to decipher a decomposition that uses dereferencing as a prefix, and indexing and selection as a suffix. Fortunately, the most common sequence of operations, dereferencing followed by selection, has a special elegant syntax. If ptr points to a record, then ptr− >field is a shorthand for (*ptr).field.

The Ada syntax is based on the assumption that dereferencing is almost always followed by selection, so there is no need for a separate notation for dereferencing. You cannot tell if R.Field is simply the selection of a field of an ordinary record called R, or if R is a pointer to a record which is dereferenced before selection. While this can be confusing, it does have the advantage that a data structure can be changed from using records to using pointers to records, without otherwise

modifying the program. On those occasions when pure dereferencing is needed, a rather awkward syntax is used, as shown by repeating the above example in Ada:

Ada

```
type Rec_Type is
   record
      Field: Integer;
   end record;
type Array_Type is array(1..100) of Rec_Type;
type Ptr_Type is access Array_Type;

Ptr: Ptr_Type;

Ptr                 -- Pointer to array of records with integer field
Ptr.all             -- Array of records with integer field
Ptr.all[78]         -- Record with integer field
Ptr.all[78].Field   -- Integer
```

Note that Ada uses a keyword access, rather than a symbol, to denote pointers. The keyword all is used on those few occasions when deferencing without selection is required.

Implementation

The indirect access to data via pointers requires an additional instruction in the machine code. Let us compare a direct assignment statement with an indirect assignment; for example:

C

```
int i,j;
int *p = &i;
int *q = &j;

i = j;              /* Direct assignment */
*p = *q;            /* Indirect assignment */
```

The machine instructions for the direct assignment are:

```
load    R1,j
store   R1,i
```

while the instructions for the indirect assignment are:

```
load    R1,&q       Addr(designated object)
load    R2,(R1)     Load designated object
load    R3,&p       Addr(designated object)
store   R2,(R3)     Store to designated object
```

Some overhead is inevitable in indirection, but is usually not serious, because if a designated object is accessed repeatedly, the optimizer can ensure that the pointer is loaded only once. Given:

C

```
p->right = p->left;
```

once the address of p is loaded into a register, subsequent accesses can use the register:

```
load    R1,&p          Addr(designated object)
load    R2,left(R1)    Offset from start of record
store   R2,right(R1)   Offset from start of record
```

A potential source of inefficiency in indirect access to data via pointers is the size of the pointers themselves. In the early 1970's when C and Pascal were designed, computers usually had only 16K or 32K of main memory, and addresses could be stored in 16 bits. Now that personal computers and workstations are delivered with many megabytes of memory, pointers must be stored in 32 bits. In addition, because of memory management schemes based on caches and paging, arbitrary access to data through pointers can be much more expensive than access to arrays that are allocated in contiguous locations. The implication is that optimizing a data structure for efficiency is highly system-dependent, and should never be done without first using a profiler to measure execution times.

The typed pointers in Ada offer one possibility for optimization. The set of designated objects associated with a specific access type is called a *collection*, and you can specify its size:[2]

Ada

```
type Node_Ptr is access Node;
for Node_Ptr'Storage_Size use 40_000;
```

Since the amount of memory requested for Nodes is less than 64K, the pointers can be stored in 16 bits relative to the start of the block of memory, thus saving both space in data structures and CPU time to load and store pointers.

* Aliased pointers in Ada 95

A pointer in C can be used to give an alias to an ordinary variable:

C

```
int i;
int *ptr = &i;
```

Aliases are occasionally useful; for example, they can be used to construct compile-time linked structures. Since pointer-based structures can only be created at run-time in Ada 83, this can cause unnecessary overhead in both time and memory.

Ada 95 has added an aliasing facility, called *general access types*, but it is restricted so that it cannot create dangling pointers (see Section 8.5). A special syntax is needed both for the pointer declaration and for the aliased variable:

[2] Ada 95 has more sophisticated ways of controlling memory allocation; see the Language Reference Manual 13.11.

Ada

```
type Ptr is access all Integer;      -- Ptr can point to alias
I: aliased Integer;                  -- I can be aliased
P: Ptr := I'Access;                  -- Create the alias
```

The first line declares a type that can point to an aliased Integer variable, the second lines declares such a variable and the third line declares a pointer and initializes it with the variable's address. Such general access types and aliased variables can be components of arrays and records so that linked structures can be built without invoking the run-time memory manager.

* Memory mapping

Memory mapping is trivial in C because an arbitrary address may be assigned to a pointer:[3]

C

```
int * const reg = 0x4f00;   /* Address (in hex) */

*reg = 0x1f1f;              /* Value to absolute location */
```

By using a constant pointer, we are assured that the address of the register will not be accidentally modified.

Ada uses the concept of representation specification to explicitly map an ordinary variable to an absolute address:

Ada

```
Reg: Integer;
for Reg use at 16#4f00#; -- Address (in hex)

Reg := 16#1f1f#;         -- Value to absolute location
```

The advantage of the Ada method is that no explicit pointers are used.

8.2 Data structures

Pointers are used to implement dynamic data structures such as lists and trees. In addition to data elements, a node in the structure will contain one or more pointers which point to another node (Figure 8.3).

Nodes must be defined using recursive type definitions, that is, a record of type node must contain a pointer to its own type node. Languages have provisions for declaring such types by allowing a partial declaration of the record to supply a name for the record, followed by a declaration of the pointer which refers to this name, followed by the full declaration of the record which can now reference the pointer type. These three declarations in Ada are:

[3] Recall that an address may not be a simple integer, so use of this construct introduces a machine-dependency.

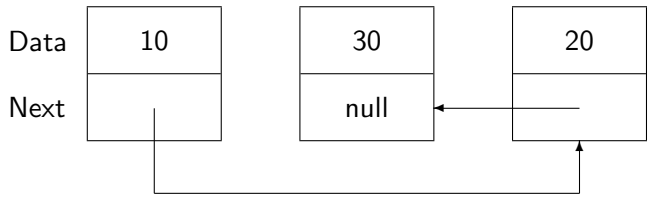

Figure 8.3: Dynamic data structure

Ada

```
type Node;                  -- Incomplete type declaration
type Ptr is access Node;    -- Pointer type declaration
type Node is                -- Complete the declaration
   record
      Data: Integer;        -- Data in the node
      Next: Ptr;            -- Pointer to next node
   end record;
```

The syntax in C requires the use of structure *tags*, which provide an alternative syntax for record declarations:

C

```
typedef struct node *Ptr;   /* Pointer to (tagged) node */
typedef struct node {       /* Declaration of node structure */
   int     data;            /* Data in the node */
   Ptr     next;            /* Pointer to next node */
} node;
```

In C++, there is no need to use a typedef as struct defines both a structure tag and a type name:

C++

```
typedef struct node *Ptr;   /* Pointer to (tagged) node */
struct node {               /* Declaration of node structure */
   int     data;            /* Data in the node */
   Ptr     next;            /* Pointer to next node */
};
```

Algorithms on these data structures use pointer variables to *traverse* the structure. The following statement in C searches for a node whose data field is key:

C

```
while (current− >data != key)
   current = current− >next;
```

The analogous statement in Ada is (using implicit dereferencing):

Ada

```
while Current.Data /= Key loop
   Current := Current.Next;
end loop;
```

Data structures are characterized by the number of pointers per node and the directions in which they point, as well as by the algorithms used to traverse and modify the structure. All the methods that you studied in a course on data structures can be directly programmed in C or Ada using records and pointers.

The null pointer

In Figure 8.3, the next field of the final element of the list does not point to anything. By convention, such a pointer is considered to have a special value called *null*, which is different from any valid pointer. The null value is denoted in Ada by a reserved word null. The search in the previous section would actually be programmed as follows so as not to run off the end of the list:

Ada

```
while (Current /= null) and then (Current.Data /= Key) loop
   Current := Current.Next;
end loop;
```

Note that it is essential that short-circuit evaluation be used (see Section 6.2).

C uses the ordinary integer literal zero to indicate the null pointer:

C

```
while ((current != 0) && (current->data != key))
   current = current->next;
```

The zero literal is just a syntactic convention; if you use a debugger to examine a null pointer, the bit string *may or may not* be all zeros depending on your computer. To improve readability, a symbol NULL is defined in the C library:

C

```
while ((current != NULL) && (current->data != key))
   current = current->next;
```

When a variable such as an integer is first declared, its value is undefined. This does not cause too much difficulty because every possible bit string that might be stored in such a variable is a valid integer. However, pointers that are not null, and yet do not point to valid blocks of memory, can cause serious bugs. Therefore, Ada implicitly initializes every pointer variable to null. C implicitly initializes every *global* variable to zero; global pointer variables are initialized to null, but you have to take care to explicitly initialize local pointer variables.

One must be very careful not to dereference a null pointer, because the null value doesn't point to anything (or rather probably points to system data at address zero):

Ada

```
Current: Ptr := null;
Current := Current.Next;
```

In Ada this error causes an exception (see Chapter 11), but in C the result of attempting to dereference null is potentially catastrophic. Operating systems which protect programs from each other will be able to abort the program; without such protection, the dereference could interfere with another program or even crash the system.

Pointers to subprograms

In C a pointer can point to a function. This is extremely useful in two programming situations:

- Passing a function as a parameter.
- Creating a data structure which associates a procedure with a key or an index.

For example, one of the parameters of a numerical integration package will be the function to be integrated. This is easily programmed in C by creating a data type which is a pointer to a function; the function will receive a parameter of type float and return a value of type float:

C

```
typedef float (*Func) (float);
```

The syntax is rather confusing because the name of the type is Func which is deep within the declaration, and because precedence in C requires the extra set of parentheses.

Once the type is declared, it can be used as the type of a formal parameter:

C

```
float integrate(Func f, float upper, float lower)
{
   float u = f(upper);
   float l = f(lower);
   ...
}
```

Note that dereferencing is automatic when the function parameter is called, otherwise we would have to write (*f)(upper). Now, if a function of the appropriate signature is defined, it can be used as an actual parameter to the integrate subprogram:

C

```
float fun(float parm)
{
...        /* Definition of "fun" */
}

float x = integrate(fun, 1.0, 2.0);  /* "fun" as actual parameter */
```

Data structures with pointers to functions are used when creating *interpreters*, which are software programs that receive a sequence of codes and perform actions according to those codes. While a static interpreter can be implemented using a case-statement and ordinary procedure calls, a dynamic interpreter will have the association between codes and actions determined at run-time. Modern windowing systems use a similar programming technique: the programmer is required to provide a *callback* which is a procedure that provides the appropriate action for each event. This is a pointer to a subprogram that will be executed when a code is received indicating that an event has occurred:

C

```
typedef enum {Event1, ..., Event10} Events;
typedef void (*Actions)(void);
        /* Pointer to procedure */
Actions action[10];
        /* Array of ptr's to procedures */
```

At run-time, a procedure is called to create the association between the event and the action:

C

```
void install(Events e, Actions a)
{
   action[e] = a;
}
```

Then when an event occurs, its code can be used to index and call the appropriate action:

C

```
action[e]( );
```

Since Ada 83 does not have pointers to subprograms, this technique cannot be programmed without using non-standard features. When the language was designed, pointers to subprograms were left out because it was assumed that generics (see Section 10.3) would suffice for creating mathematical libraries, and the callback technique was not yet popular. Ada 95 has corrected this deficiency and allows pointers to subprograms. The declaration of a mathematical library function is as follows:

Ada

```
type Func is access function(X: Float) return Float;
        -- Pointer to function type
function Integrate(F: Func; Upper, Lower: Float);
        -- Parameter is a pointer to a function
```

and the callback is declared as follows:

Ada

```
type Events is (Event1, ..., Event10);
type Actions is access procedure;
        -- Pointer to procedure type
Action: array(Events) of Actions;
        -- Array of pointers to procedures
```

Pointers and arrays

In the strongly typed framework of Ada, the only possible operations on pointers are assignment, equality and dereferencing. C, however, considers pointers to be implicit sequential addresses and arithmetic is possible on pointer values. This is most clearly seen in the relationship between pointers and arrays: pointers are considered to be the more primitive concept, while array accesses are defined in terms of pointers. In the following example:

C

```
int *ptr;              /* Pointer to integer */
int a[100];            /* Array of integer */

ptr = &a[0];           /* Explicitly address first element */
ptr = a;               /* Implicitly do the same thing */
```

the two assignment statements are equivalent because an array name is considered to be just a pointer to the first element of the array. Furthermore, if addition or subtraction is done on the pointer, the result is not the numerical result but the result of incrementing or decrementing the pointer by the size of the designated type. If an integer requires four bytes and p contains the address 344, then p+1 is not 345 but rather 348 which is the "next" integer. Accessing an element of an array is done by adding the index to the pointer and dereferencing, so that the two following expressions are equivalent:

C

```
*(ptr + i)
a[i]
```

Despite this equivalence, there is still an important distinction in C between an array and a pointer:

C

```
char s1[] = "Hello world";
char *s2 = "Hello world";
```

s1 is the location of a sequence of 12 bytes of memory containing the string, while s2 is a pointer variable containing the address of a similar sequence of bytes (Figure 8.4). However, s1[i] is the same as *(s2+i) for any i in range, because an array is automatically converted to a pointer when used.

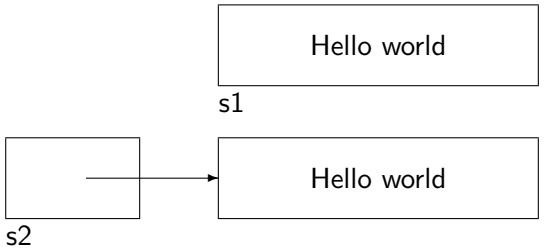

Figure 8.4: Array versus pointer in C

The problem with pointer arithmetic is that there is no guarantee that the result of the expression actually points to an element of the array. While it is relatively easy to understand indexing notation and be convinced of its correctness, pointer arithmetic should in general be avoided. It can be very useful for improving efficiency in loops if your optimizer is not good enough.

8.3 Dynamic data structures in Java

How can you create lists and trees without pointers?! The declarations for linked lists in C++ and Ada given in Section 8.2 would seem to indicate that we need a pointer type to describe the type of the next field:

C

```
typedef struct node *Ptr;
typedef struct node {
    int     data;
    Ptr     next;
} node;
```

But in Java, every object of non-primitive type is automatically a pointer:[4]

Java

```
class Node {
    int data;
    Node next;
}
```

The field next is merely a pointer to a Node, not a Node, so there is no circularity in the declaration. Declaration of a list is simply:

Java

```
Node head;
```

which creates a pointer variable whose value is null (Figure 8.5 (a)). Assuming that there is an appropriate constructor (Section 15.4) for Node, the following statement creates a node at the head of the list (Figure 8.5 (b)):

Java

```
head = new Node(10, head);
```

8.4 Equality and assignment in Java

The behavior of the assignment statement and the equality operator in languages with reference semantics can be a source of surprises for programmers with experience in a language with value semantics. Consider the Java declarations:

[4] C++ retains the struct construct for compatibility with C. However, the construct is identical with a class all of whose members are declared public. Java does not retain compatibility with C; instead, every type declaration must be a class.

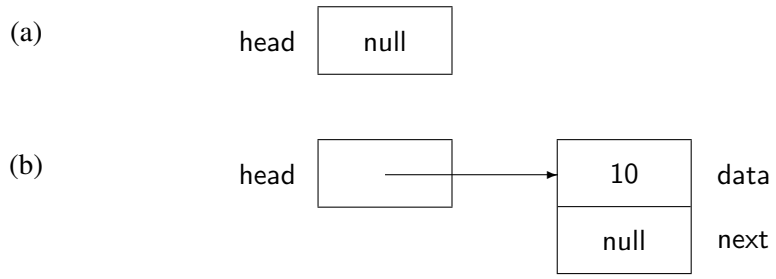

Figure 8.5: Linked list in Java

```
String s1 = new String("Hello");
String s2 = new String("Hello");
```
Java

This results in the data structure shown in Figure 8.6.

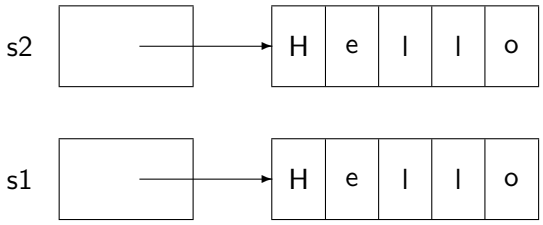

Figure 8.6: Assignment and equality

Now suppose that we compare the string variables:

```
if (s1 == s2) System.out.println("Equal");
else System.out.println("Not equal");
```
Java

The program will print Not equal! The reason is actually clear from Figure 8.6: the variables are pointers with different values, and the fact that they point to equal arrays is irrelevant. Similarly, if we assign one string to the other s1 = s2, the pointers will be assigned and no copying of values will be done. In this case, of course, s1 == s2 will be true.

Java distinguishes between *shallow* copy and comparison, and *deep* copy and comparison. The latter are declared in the common ancestor class Object and are called clone and equals. The predefined class String, for example, overrides these operations, so s1.equals(s2) is true. You can override the operations to create deep operations for your classes.

To summarize the use of reference semantics in Java:

- Flexible data structures can be safely manipulated.
- Programming is simpler because no explicit pointers need be used.
- There is an overhead associated with indirect access to data structures.

8.5 Memory allocation

During the execution of a program, memory is used to store both code and a variety of data structures such as the stack. While the allocation and release of memory is more properly discussed in the context of compilers and operating systems, it is appropriate to survey the topic here because the choice of language constructs and style can be profoundly affected by the implementation.

There are five types of memory that need to be allocated:

Code The machine instructions that are the result of the compilation of the program.

Constants Small constants like 2 and 'x' can often be included within an instruction, but memory must be allocated for large constants, in particular for floating point constants and strings.

Stack Stack memory is used mainly for activation records which contain parameters, variables and links. It is also used as a source of temporary variables when evaluating expressions.

Static Data Variables declared in the main program and elsewhere: in Ada, data declared directly within library packages; in C, data declared directly within a file or declared `static` in a block.

Heap The heap is the term used for the data area from which data is dynamically allocated by `malloc` in C and by `new` in Ada and C++.

Code and constants are similar in that they are fixed at compile-time and are not modified. Thus we will group the two types together in the following discussion. Note that code and constants can be allocated in read-only memory (ROM) if the system supports it. The stack was discussed in great detail in Section 7.6.

We mentioned there that static (global) data could be considered to be allocated at the beginning of the stack. However, static data are usually allocated separately. For example, on the Intel 8086, each area of data (called a *segment*) is limited to 64K bytes of memory. Thus a separate segment would be allocated for the stack in addition to one or more segments for static data.

Finally, we must allocate memory for the heap. A heap differs from a stack in that the allocation and deallocation can be very chaotic. The run-time system must employ sophisticated algorithms to ensure optimal use of the heap.

A program is usually allocated a single, contiguous area in memory which must be divided up to accommodate the required memory areas. Figure 8.7 shows the way that this is done. Since

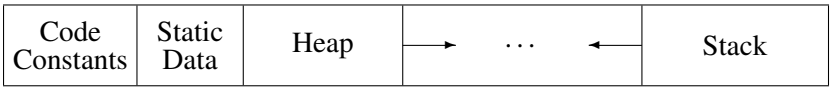

Figure 8.7: Memory allocation: code, data, stack and heap

the code, constant and static data areas are fixed, they are allocated at the beginning of the memory. The two variable-length areas, the heap and the stack, are allocated at opposite ends of the remaining memory.[5] This way if a program uses a large amount of stack during one phase of a

[5]Note that the arrangement is symmetric in that the stack could grow up and the heap down instead of as shown.

computation and a large amount of heap during another, there is less chance that there will not be enough memory.

What is important to understand is that each allocation of memory on the stack or the heap (that is each procedure call and each execution of a storage allocator) can potentially fail for lack of memory. A well-designed program should gracefully recover from lack of memory, but this is not an easy situation to handle, because even more memory might be needed for the procedure which executes the recovery! Therefore it is recommended that lack of memory be signalled when a substantial reserve remains.

Allocation and deallocation

Imperative programming languages have explicit expressions or statements for allocating and deallocating memory. C uses malloc which is dangerous because no checking is done that the amount of memory returned is correct for the size of the designated object. sizeof should be used even though it is not required:

C

```
int *p = (int *) malloc(1);            /* Error */
int *p = (int *) malloc(sizeof(int)); /* Better */
```

Note that malloc returns an *untyped* pointer that should be explicitly converted to the required type.

The size of the block need not be given when memory is freed:

C

```
free(p);
```

The allocated memory block includes a few extra words which are used to store the size of the block. The size is used in algorithms for heap management as described below.

C++ and Ada use a notation which clearly brings out the fact that a designated object of a specific type is being created. There is no danger of incompatibility between the type and size of the object:

```
typedef Node *Node_Ptr;
Node_Ptr *p = new Node;          // C++

type Node_Ptr is access Node;
P: Node_Ptr := new Node;         -- Ada
```

The delete operator frees memory in C++. Ada would prefer that you do not free memory allocated from the heap, because freeing memory is inherently dangerous (see below). Of course this is impractical, so a method for releasing memory is supplied called *unchecked deallocation*, named as such to remind you that any use is dangerous. Note that the memory freed is that of the designated object (the object that is pointed *to*) and not the memory used to store the pointer itself.

Dangling pointers

A serious danger with pointers is the possibility of creating *dangling pointers* when releasing a block of memory:

C++

```
int *ptr1 = new int;
int *ptr2;

ptr2 = ptr1;              // Both point to the same block
result = delete ptr1;     // ptr2 now points to released block
```

After executing the first assignment statement, the allocated block of memory is pointed to by two pointers. When the memory is freed, the second pointer still retains a copy of the address but this address is now meaningless. In a complex data structure algorithm, a double reference of this type is quite easy to construct by mistake.

Dangling pointers can also be created in C and C++ without any explicit deallocation on the part of the programmer:

C

```
char *proc(int i)         /* Returns pointer to char */
{
   char c;                /* Local variable */
   return &c;             /* Pointer to local char */
}
```

The memory for c is implicitly allocated on the stack when the procedure is called and implicitly deallocated upon return, so the pointer value returned no longer points to a valid object. This is easy to see in a two-line procedure but may not be so easy to notice in a large program.

Ada tries to avoid dangling pointers:

- Pointers to objects (named variables, constants and parameters) are forbidden in Ada 83; in Ada 95, they are subject to the special alias construct whose rules prevent dangling pointers.
- Explicit allocation cannot be avoided, but it is given a name Unchecked Deallocation that is intended to warn the programmer of the dangers.

8.6 Algorithms for heap allocation

The heap manager is a component of the run-time system that allocates and releases memory. It does this by maintaining a list of *free* blocks. The list is searched when an allocation request is made, and a released block is relinked to the free list. The run-time system designer must make many choices such as the order in which the blocks are maintained, the data structure used, the order of search and so on.

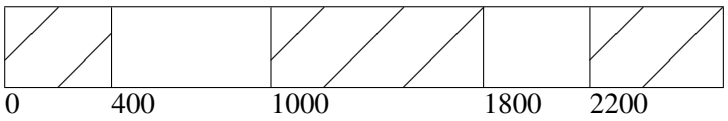

Figure 8.8: Heap fragmentation

The problem with heap allocation is *fragmentation*. In Figure 8.8, five blocks of memory have been allocated and the second and fourth have been released. Now even though there are 1000 bytes available, it is impossible to successfully allocate more than 600 bytes because the memory is fragmented into small blocks. Even if the third block were released, it does not immediately follow that there will be enough memory, unless the heap manager has been designed so that it can merge adjacent free blocks.

In addition to merging, the heap manager can help prevent fragmentation by searching for an appropriate-sized block rather than simply taking the first available one, or by allocating large blocks from one area in the heap and small blocks in another. There is a clear trade-off between the sophistication of the heap manager and the run-time overhead required.

The programmer should be aware of the heap algorithms used and can write a program that takes account of this knowledge.[6] Alternatively, it is possible to refrain from dependence of the heap algorithm by maintaining a cache of deallocated blocks. When a block is released, simply link it onto the cache. When a block is needed, the cache is checked first to avoid both the overhead and the fragmentation dangers of calling the heap manager.

Ada has a facility that allows the programmer to specify different heap sizes, one for each type of pointer. This can prevent fragmentation but runs the risk of one heap running out of memory while others have plenty of free blocks.

Virtual memory

There is one case where heap allocation is essentially risk-free and that is when *virtual memory* is being used. In a virtual memory system, the programmer receives an extremely large address space, so large that memory overflow is practically impossible. The operating system takes care of allocating the logical address space to physical memory as needed. When physical memory is exhausted, blocks of memory called *pages* are swapped to disk.

With virtual memory, the heap allocator can continue allocating from the heap almost forever, so no fragmentation problems are encountered. The only risk is the standard risk of virtual memory called *thrashing*, which occurs when the code and data required for a phase of a computation are spread over so many pages that there is not enough room for all of them in memory. So much time is spent swapping pages that the computation does not progress.

Garbage collection

The final problem with heap allocation is the creation of *garbage*:

[6] This is another case where the program can be portable, but its efficiency may be highly system-dependent.

C

```
int *ptr1 = new int;        // Allocate first block
int *ptr2 = new int;        // Allocate second block
ptr2 = ptr1;                // Now, second block inaccessible
```

Following the assignment statement, the second block of memory is accessible through either of the pointers, but there is no way to access the first block (Figure 8.9). This may not be a bug

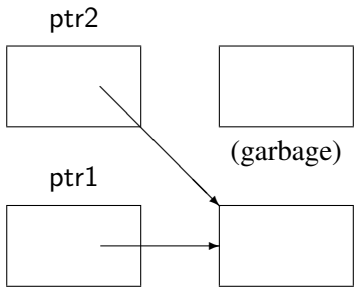

Figure 8.9: Creation of garbage

because memory you can't access, called garbage, can't hurt you. However, if memory continues to *leak*, that is if garbage creation continues, eventually the program will fail because of lack of memory. It is extremely difficult to locate the cause of memory leakage, because there is no direct connection between the cause and the symptom (lack of memory).

The obvious solution is not to create garbage in the first place by being careful to deallocate every block before it becomes inaccessible. Alternatively, the run-time system of a programming language may contain a *garbage collector*. The task of a garbage collector is to "recycle" garbage by identifying inaccessible memory blocks and returning them to the heap manager. There are two basic garbage collection algorithms: one maintains a count of the number of pointers currently pointing to each block and automatically deallocates the block when the count goes to zero. The other algorithm marks all accessible blocks and then collects unmarked (and hence inaccessible) blocks. The first algorithm is problematical because a group of blocks, all of which are garbage, may point at one another so the count may never decrease to zero. The second algorithm requires that the computation be interrupted for long periods of time, so that the marking and collecting can be done without interference from the computation. This of course is not acceptable in interactive systems.

Garbage collection is traditionally done in languages like Lisp and Icon which create large amounts of temporary data structures that quickly become garbage. Extensive research has been done into garbage collection, with emphasis on concurrent and incremental methods that will not disrupt an interactive or real-time computation. Eiffel is one of the few imperative languages that include garbage collectors in their run-time systems.

8.7 Exercises

1. How is a pointer represented on your computer? How is the null pointer represented on your computer?

2. Write an array algorithm in C using indexing and then change it to use explicit pointer manipulation. Compare the resulting machine instructions and then compare the run time of the two programs. Does optimization make a difference?

3. Show how sentinels can be used to make searching a list more efficient.

4. For an actual parameter of type pointer to function, why wasn't the addressing operator used:

```
float x = integrate(&fun, 1.0, 2.0);
```
C

5. Show how dangling pointers can be used to break the type system.

6. Study the Ada 95 definition of *accessibility* and show how the rules prevent dangling pointers.

7. Write a program that uses a dynamic data structure such as a linked list. Modify the program to use a cache of nodes.

8. Study your compiler's documentation; what algorithms does the run-time system use for heap allocation? Is there any memory overhead when allocating from the heap, that is, are extra words allocated in addition to those you requested? If so, how many?

9. If you have access to a computer that uses virtual memory, see how long you can continue to allocate memory. What limit finally terminates allocation?