

ECE391: Computer Systems Engineering — Lecture Notes Set 0

Review Material

This set of notes reviews material that you have probably already seen in ECE190 or ECE290 (or CS225). If you took neither ECE190 nor CS225, some of it may be new to you; the TAs might also talk about some of this material in the first couple of discussion sections, and both MP1 and MP2 will require you to make use of it. Regardless of your previous experience, you may want to scan the list of terms at the end of these notes and review the definitions for any terms that seem unclear to you. In order to make the notes more useful as a reference, definitions are highlighted with boldface, and italicization emphasizes pitfalls.

The notes begin with a review of how information is represented in modern digital computers, highlighting the use of memory addresses as pointers to simplify the representation of complex information types. The next topic is the systematic decomposition of tasks into subtasks that can be executed by a computer, and an example based on a pointer-based data structure. Moving on to high-level languages, and particularly the C language, the notes continue with a review of type and variable declarations in C as well as the use of structures, arrays, and new types. The next topic is C operators, with some discussion of the more subtle points involved in their use. After revisiting the decomposition example as written in C, the notes discuss implicit and explicit conversions between types. Finally, the notes conclude with a review of C's preprocessor. We deliberately omit the basics of C syntax as well as descriptions of C conditional (`if/else` and `switch`) and iteration (`while`, `do`, and `for` loops) statements.

Representation as Bits

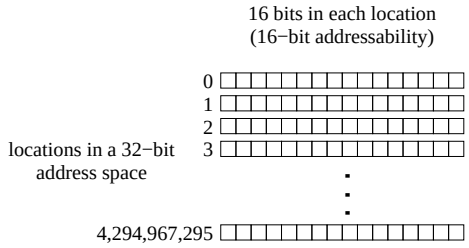
Modern digital computers represent all information as sets of binary digits, *i.e.*, 0s and 1s, or **bits**. Whether you are representing something as simple as an integer or as complex as an undergraduate thesis, the data are simply a bunch of 0s and 1s inside a computer. For any given type of information, a human selects a data type for the information. A **data type** (often called just a **type**) consists of both a size in bits and a representation, such as the 2's complement representation for signed integers, or the ASCII representation for English text. A **representation** is a way of encoding the things being represented as a set of bits, with each bit pattern corresponding to a unique object or thing.

In general, computers do not interpret the bits that they store. A typical instruction set architecture (ISA) supports a handful of data types in hardware in the sense that it provides hardware support for operations on those data types. Most modern arithmetic logic units (ALUs), for example, support addition and subtraction of both unsigned and 2's complement representations, with the specific data type (such as 16- or 64-bit 2's complement) depending on the ISA. Data types and operations not supported by the ISA must be handled in software using a small set of primitive operations, which form the **instructions** available in the ISA. Instructions usually include data movement instructions such as loads and stores and control instructions such as branches and subroutine calls in addition to operations.

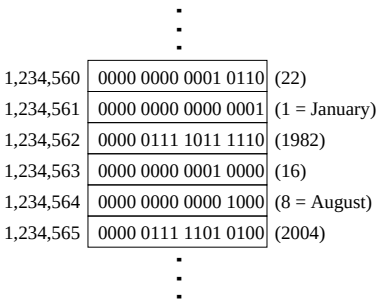
Pointers and Data Structures

One particularly important type of information is memory addresses. Imagine that we have stored some bits representing the number 42 at location 0100100100010100 in a computer's memory. In order to retrieve our stored number, or rather the bits representing it, we must provide the memory with the bits corresponding to the memory location, in this case 0100100100010100. The representation of a memory address as bits is thus straightforward: we simply use the bits that must be provided to the memory in order to retrieve the desired data as the representation for that memory address. Unlike most other types of information, no translation process is necessary; we represent a bunch of bits with a bunch of bits. We might thus think of the bits 0100100100010100 as a **pointer** to the number 42. They point to our stored number in the sense that the memory, when provided with the pointer, produces the stored number.

It is important to recognize, however, that *a pointer is just a bunch of bits, just like any other representation used by a computer*. In particular, we can choose to interpret the bits making up a pointer as an unsigned integer, which provides us with a way of ordering the locations in a memory's address space, as shown to the right for a memory consisting of 2^{32} locations holding 16 bits each, *i.e.*, with an **addressability** of 16 bits.



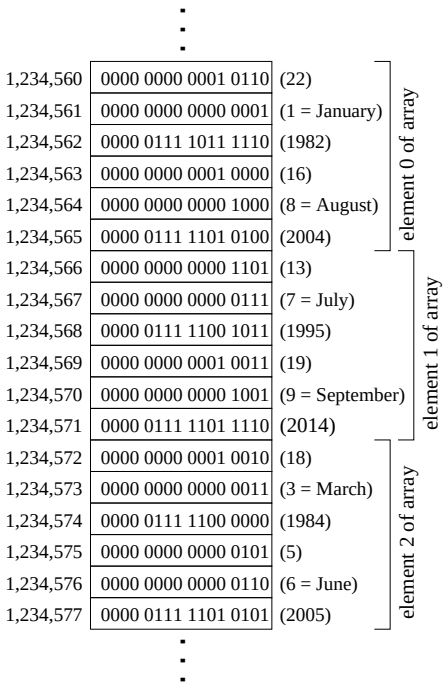
The induced ordering is useful when the representation for some type of information requires more bits than the addressability of the memory. Imagine that you are charged with selecting a representation for the birth and hiring dates of all employees of your company in a memory with 16-bit addressability. Given a reasonable set of assumptions about the range of possible values, representing two dates with only 16 bits is quite challenging. Instead, you might decide to use six memory locations for each person, representing each person with 2's complement integers for the day, month, and year of the two recorded dates. An example appears to the right. Such a multi-location representation is often called a **data structure**, because the information represented is made up of several components, or **fields**, each of which has a data type. In this case, all of the fields are 16-bit 2's complement integers, but the fields of a data structure need not in general have the same type.



In the example above, the value 1,234,560 (*i.e.*, the bit pattern 00000000000100101101011010000000) is a pointer to the employee structure, and all fields of the data structure reside at fixed offsets from this pointer. To find the hiring month, for example, one adds four to the pointer and retrieves the value from the resulting memory address (*i.e.*, treating the sum again as a pointer).

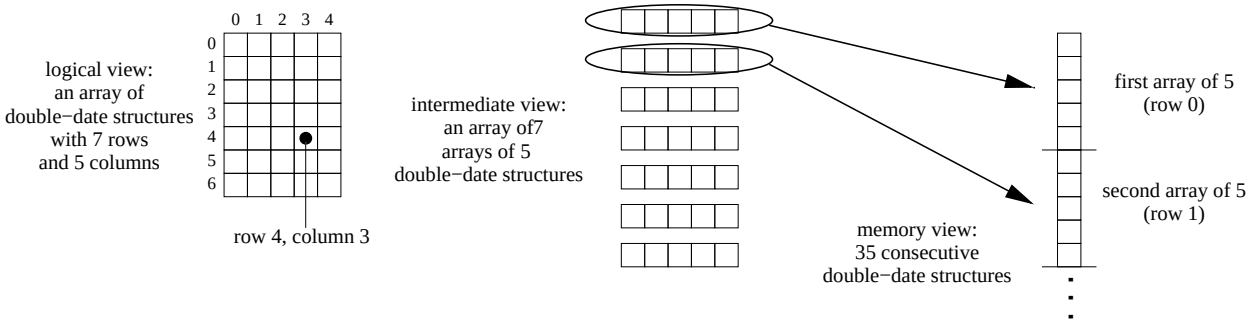
Arrays

Multiple instances of a certain type are often represented as arrays. Again using the ordering induced by treating memory addresses as unsigned integers, the bits for each element of the array are simply laid out sequentially in memory, with each element occupying whatever number of memory locations is necessary for the data type. We refer to this number as the **size** of the type, and (usually) measure it in terms of the addressability of the memory system. Given the address of the first element of an array, we can easily calculate the address of any element by simply adding a multiple of the size of the type. For convenience, we number array elements starting with 0, allowing us to use the product of the array index and the type size as the offset for any given element. *All array elements must be of the same size in order for this calculation to work properly.* In the example shown to the right, we find element 2 by adding 2×6 to 1,234,560. The result, 1,234,572, is a pointer to element 2 of the array. We can also think of the pointer 1,234,572 as a pointer to an array with two fewer elements than the original array.



The **duality** between pointers and arrays, by which any pointer can be treated as an array, and any array can be treated as a pointer, is both useful and a potential source of confusion. If, for example, you create an array of ten of our double-date structures, and write code that takes a pointer to the array of ten and accesses the eleventh (element 10), your code will proceed to read garbage bits from the memory. Adding 10×6 to the pointer to element 0 does produce a valid memory address, but neither the contents of that address nor those of the subsequent five addresses were intended to be a double-date structure. With any pointer (*i.e.*, memory address), your program must keep track of the type of the data to which the pointer points. With a pointer to an array, your code must also keep track of the size of the array. To a computer, bits are just bits.

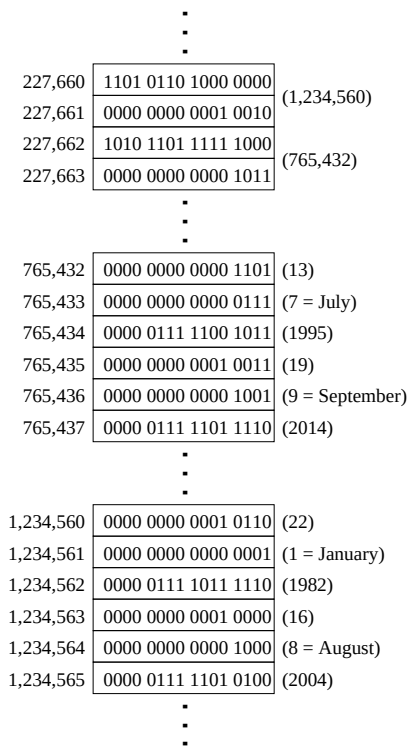
Multidimensional arrays are usually handled by thinking of each of the lower dimensions in the array as a new type of data. For example, let's say that we want to create an array of our double-date structures with seven rows and five columns, as shown in the following figure. We begin by thinking about an array of five double-date structures. Each row in our two-dimensional array is simply an array of five double-date structures. Our entire two dimensional array is then an array of seven of the arrays of five. When we lay it out in memory, the arrays of five are again laid out consecutively.



How do we find an element in our two-dimensional array? Let's say that we want to access row 4 and column 3. The size of the array of five is $5 \times 6 = 30$, and we are looking for row 4, so we multiply 4×30 and add it to the pointer to the start of the array to find a pointer to row 4. We then multiply 3×6 and add it to the pointer to row 4 to obtain a pointer to the desired array element. As you may have already observed, the duality between arrays and pointers also holds for multi-dimensional arrays: we can choose to view our two-dimensional array as an one-dimensional array of 35 structures if it is useful or convenient to do so.

Arrays of Pointers

As mentioned earlier, a pointer is simply a type of information. In practice, it is often useful to associate the type of information to which a pointer points with the pointer itself, but this association is purely an abstraction and has no effect on the bits used to represent the pointer. However, mentally distinguishing a pointer to an integer from a pointer to one of the double-date data structures from our running examples can help when discussing our next step: a pointer may point to a pointer, which may in turn point to another pointer, and so forth.



As a simple example, rather than building an array of our double-date structures in memory, we might instead build an array of pointers to double-date structures, as shown to the right. The upper block in the figure shows the first two pointers in our array of pointers; each pointer occupies 32 bits, or two memory locations. The lower block in the figure shows the first structure referenced by the array; the address of this structure is stored as element 0 in our array of pointers. The middle block in the figure shows the second structure referenced by the array, *i.e.*, the structure to which element 1 of the array points. Given a pointer—227,660—to the array, we obtain a pointer to element 1 by adding 2, the size of a pointer, to the array pointer. The data at that memory address are then the address of the desired structure, in this case the pointer 765,432.

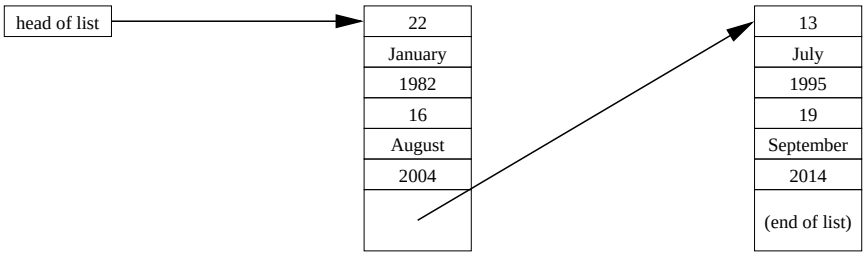
What is the difference between using an array of pointers to structures and using an array of structures? With an array of structures, finding the address of any desired element is easy, requiring only a multiplication and an addition. Using an array of pointers to structures introduces another memory access to retrieve the structure pointer from the array. However, if we want to move elements around within the array, or want to be able to have the same structure appear more than once in the array, using an array of pointers makes our task much easier, as moving and copying pointers is easier than moving and copying whole data structures, particularly if the data structures are large. Similarly, using an array of pointers allows us to use a special value (usually 0) to indicate that some elements do not exist, which may reduce the amount of memory needed to store an array for which the size changes while the program executes.

Pointers within Data Structures

Just as we can form arrays of pointers, we can also include pointers as fields within data structures. For example, we can extend our double-date structure by appending a field that points to another double-date structure, as shown to the right. This new structure can then be linked together, from one to the next, to form a list of structures representing the birth and hiring dates for all employees of a company. We call such a structure a **linked list**, with the pointer field forming the link between each element in the list and the subsequent element. A single pointer then suffices to indicate the location of the start of the list, and a special value (again, 0) is used to indicate the end of the list.

birth day
birth month
birth year
hiring day
hiring month
hiring year
pointer to next structure

The figure below illustrates a linked list of two employees. The upper part of the figure is the logical list structure, and the lower part is a possible organization in memory. For this figure, we have deliberately changed both the addresses and bits for each memory location into hexadecimal form. Hexadecimal notation is easier for humans to parse and remember than is binary notation. However, *use of hexadecimal is purely a notational convenience and does not change the bits actually stored in a computer.*



0x0003794C	0xD620	(0x0012D620)
0x0003794D	0x0012	

0x0012D620	0x0016	(22)
0x0012D621	0x0001	(1 = January)
0x0012D622	0x07BE	(1982)
0x0012D623	0x0010	(16)
0x0012D624	0x0008	(8 = August)
0x0012D625	0x07D4	(2004)
0x0012D626	0xADF8	(0x000BADF8)
0x0012D627	0x000B	

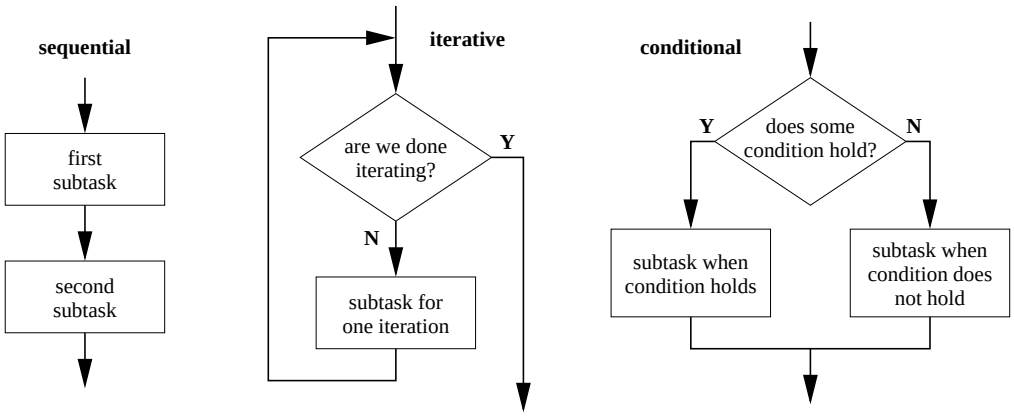
0x000BADF8	0x000D	(13)
0x000BADF9	0x0007	(7 = July)
0x000BADFA	0x07CB	(1995)
0x000BADFB	0x0013	(19)
0x000BADFC	0x0009	(9 = September)
0x000BADFD	0x07DE	(2014)
0x000BADFE	0x0000	(0x00000000)
0x000BADFF	0x0000	

What is the difference between using a linked list and using an array of pointers? A linked list requires only one pointer per element in the list, plus one extra for the list head, and does not require a programmer to guess the maximum size of the list in advance. In contrast, an array of pointers contains a fixed number of pointers; this number must be chosen in advance, and must be large enough for all executions of the program, even if most executions require far fewer list elements than this theoretical maximum. The array itself can be resized dynamically, but doing so requires extra memory accesses and copying. The speed of access in an array is its primary advantage. For comparison, getting to the middle of a list means reading all of the pointers from the head of the list to the desired element. More sophisticated data structures using multiple pointers to provide more attractive tradeoffs between space and operation speed are possible; you will see some of these in ECE391, but are not assumed to have already done so.

Systematic Decomposition

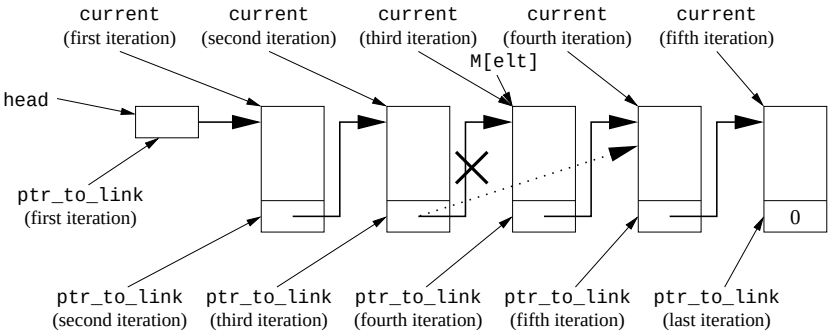
We are now ready to shift gears and talk about the process of turning a specification for a program into a working program. Essentially, given a well-defined task expressed at some suitably high level of abstraction, we want to be able to systematically transform the task into subtasks that can be executed by a computer. We do so by repeatedly breaking each task into subtasks until we reach the desired level, *i.e.*, until each of the remaining tasks requires only a machine instruction or two. The name for this process is **systematic decomposition**, and the ECE190 textbook by Patt and Patel[1] provides a detailed introduction to the ideas; the basics are replicated here for review purposes.

[1] Yale N. Patt, Sanjay J. Patel, *Introduction to Computing Systems: from Bits & Gates to C & Beyond*, second edition, McGraw Hill, New York, New York, 2004, ISBN 0-07-246750-9.



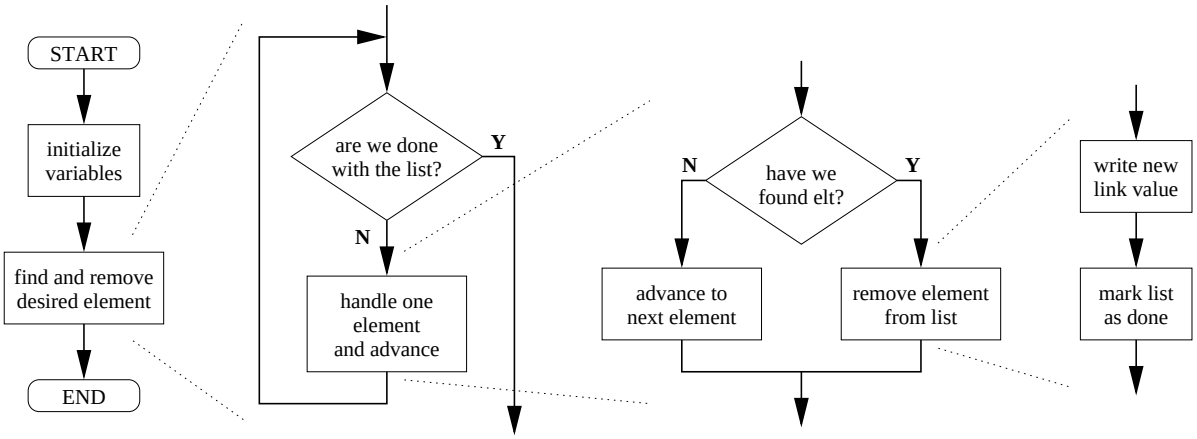
When writing a program, it is often helpful to get out a sheet of paper and draw a few pictures of the data structures before trying to write code. Similarly, it is often useful to explicitly draw the organization of the code, which is where systematic decomposition comes into play. Begin by drawing a single box that implements the entire task. Next, repeatedly select a box that cannot be implemented with a single machine instruction and break it down into smaller tasks using one of the three constructions shown above: sequential, iterative, or conditional. Some tasks require a sequence of subtasks; for these, we use the sequential construction, breaking the original task into two or more subtasks. Some tasks require repetition of a particular subtask. These make use of the iterative construction; determining the condition for deciding when the iterative process should end is critical to using such a construction. Finally, some tasks divide into two (or possibly more) subtasks based on some condition. To make use of the conditional construction, this test must be stated explicitly, and the steps for each outcome clearly stated.

Example: Decomposing Linked List Removal



As an example, let's decompose the task of removing one of our double-date data structures from a linked list. For this task, we are given two input variables. The variable `head` holds a pointer to the list head, and the variable `elt` holds a pointer to the element to be removed from the list. The variable names represent the addresses at which these values are stored. Thus, at the **register transfer language (RTL)** level, we obtain a pointer to the first list element by reading from `M[head]`, the memory location to which `head` points. The figure to the right shows a picture of the list with one possible value of `elt`, for which the dotted line and cross indicate the change to be made to the data structure in order to remove `elt` from the list.

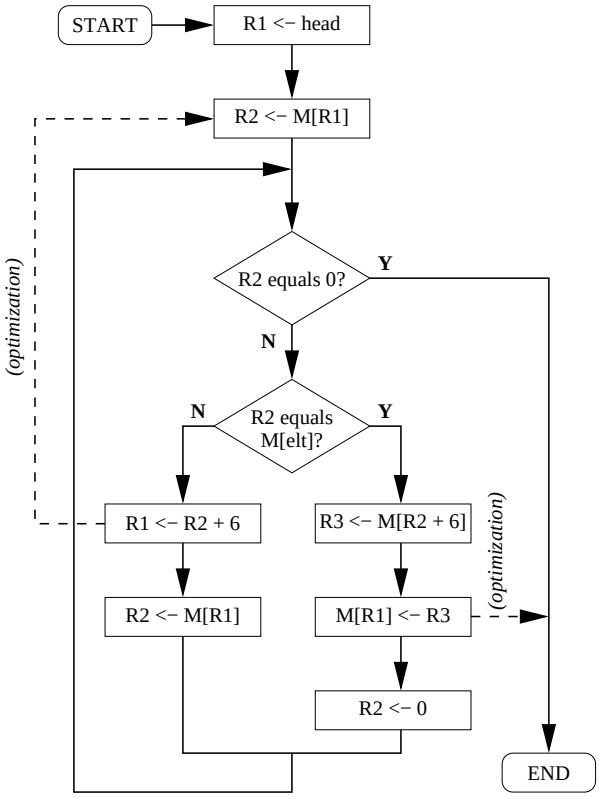
The basic operation for removal is thus to iterate over the list elements until we find the element that we want to remove, then change the element's predecessor's link to point to the element's successor, as illustrated in the diagram. For this purpose, we make use of two additional variables that we keep in registers. The first, `ptr_to_link`, points to the link that might be changed, *i.e.*, either the link field of a list element or to the memory location in which the pointer to the first element is held. The second, `current`, is simply the value to which `ptr_to_link` points. When `current` has the value 0, we have reached the end of the list. The `current` variable is thus a pointer to a double-date structure, while the `ptr_to_link` variable is a pointer to a pointer to a double-date structure.



We are now ready to systematically decompose the problem. The **flow chart** on the left above shows the process after one step of refinement in which we have dedicated a subtask to initializing the `ptr_to_link` and `current` variables. For the next step, we decompose the task of finding and removing the desired element from the list. As discussed, this step requires that we iterate over the elements of the list, so we use an iterative construction, first testing whether we have reached the end of the list, then handling one element as the iterated subtask. We next decompose the process for handling a single list element into two cases using a conditional construction. If the current list element is not equal to `elt`, we move on to the next element in the list. If the current list element is equal to `elt`, we remove it from the list by overwriting the link pointer, at which point we are done with the task and can leave the loop. To do so, we update our variables to indicate that the loop has completed. The last refinement shown, at the right side of the figure above, shows a sequential decomposition of the task of removing an element into overwriting the link pointer and marking the list as done (by setting the `current` variable to 0, *NOT by changing the list*).

At this point, we are ready to transform the tasks into RTL-level instructions. We use three registers in our code. Register R1 holds `ptr_to_link`, *i.e.*, the pointer to the pointer to the element being considered for removal as we traverse the list. Register R2 holds `current`, a pointer to the element being considered. We find `current` by **dereferencing** `ptr_to_link`, *i.e.*, by reading from the memory location to which `ptr_to_link` points. Finally, register R3 is a temporary used to hold a pointer to the successor of `elt` when removing it.

We initialize our variables by setting R1 equal to `head` and dereferencing R1 to find R2. We compare R2 with 0 to decide whether or not we have finished the list. If not, we check whether we have found `elt` by comparing it with R2. If not (the left side of the conditional), we must advance R1 to point to the link field of the next element in the list. Since R2 already points to the data structure, we need merely add the offset of the link field, which is 6 assuming our example memory, and store the result in R1. We then dereference the new value of R1 to find the new value of R2 and continue with the next iteration. Once we find `elt` (the right side of the conditional), we read the value of the link field in the current element (again at offset 6 from R2) and write it into the predecessor's link field (pointed to by R1), completing the removal of `elt`. We then set R2 to 0 to end the loop. Once we have fully decomposed the problem, two minor optimizations become apparent (dashed lines in figure).



As an exercise, try to systematically decompose the problem of inserting a new element into a linked list, given the address of the head of the list and the address of the new list element. The simple version inserts the element at the front of the list, but try writing a second version that inserts the element at the end of the list.

Designing Iterations

Systematic decomposition provides a sound basis for one's first steps in programming, but addresses only the procedural aspects of a program. While low-level programming tasks are almost always procedural in nature, many high-level decisions involve reasoning about the choice of interfaces between large code modules and the data structures used to implement those code modules. In practice, success with these latter design problems requires substantial programming experience and exposure to previous approaches, and in that sense is more the topic of our course than review material.

However, systematic decomposition can also sometimes lead to code that favors structure over clarity and simplicity, and you should already be aware of alternative systematic approaches (or, perhaps, extensions) that can help you to build your programming skills. Towards this end, we outline how a more experienced programmer designs an iterative task by asking a sequence of questions about the iteration:

1. What is the task to be repeated?
2. What **invariants** hold at the start of each iteration, *i.e.*, what statements do we know to be true at these times?
3. Under what **stopping conditions** should we end the iteration?
4. What should be done when we stop? The action might be different for each stopping condition.
5. How do we prepare for the first iteration?
6. How do we update variables between iterations?

You might notice that the early description and figure used to describe the linked list removal problem hinted at the answers to these questions, which reflects the degree to which this process becomes internalized and automatic. The concept of an invariant alone is quite powerful and useful in many other contexts, such as the design of data structures and algorithms, while stopping conditions are fully analogous to the base cases in recursive constructs.

We end by explicitly reviewing the answers for the case of linked list removal and showing that this approach leads directly to the optimized form of the process. The task to be repeated is consideration of a single list element for removal. We define two variables to help with the removal, defining their contents as our invariants. In particular `ptr_to_link` points to the link that might be changed, and `current` is the value to which `ptr_to_link` points. Note that these definitions *do not necessarily hold in the middle of the iteration*. We end the loop when we either (1) reach the end of the list, in which case we did not find the element to be removed (and might choose to return an error condition), or (2) find the element to be removed, in which case we do so. To prepare for the first iteration, we need merely initialize our two variables to match their defining invariants. To update, we change the variables to point to the next element. As `current` is defined to be the value to which `ptr_to_link` points, we can always use one subtask to set it up once `ptr_to_link` is ready, whether we are preparing for the first iteration or any other.

Data Types in C

We're now ready to move up a level of abstraction and to review the same set of concepts as they appear in the language C. High-level languages typically associate types with data in order to reduce the chance that the bits making up an individual datum are misused or misinterpreted accidentally. If, for example, you store your name in a computer's memory as a sequence of ASCII characters and write an assembly program to add consecutive groups of four characters as 2's complement integers and to print the result to the screen, the computer will not complain about the fact that your code produces meaningless garbage. In contrast, most high-level languages will give you at least a warning, since such implicit re-interpretations of the bits are rarely intentional and thus rarely correct. The compiler can also generate the proper conversion code automatically when the transformations are intentional, as is often the case with arithmetic.

Some high-level languages, such as Java, go a step further and actually prevent programmers from changing the type of a given datum (in most cases). If you define a type that represents one of your favorite twenty colors, for example, you are not allowed to turn a color into an integer, despite the fact that the color is represented as a handful of bits. Such languages are said to be **strongly typed**.

The C language is not strongly typed, and programmers are free to interpret any bits in any manner they see fit. Taking advantage of this ability in any but a few exceptional cases, however, usually results in arcane and non-portable code, and is thus considered to be bad programming practice. We discuss conversion between types in more detail later in these notes.

Each high-level language defines a number of **primitive** data types, *i.e.*, types that are always available in the language, as well as ways of defining new types in terms of these primitives. The primitive data types in C are signed and unsigned integers of various sizes, single- and double-precision floating-point numbers, and pointers. New types can be defined as arrays of an existing type, structures with named fields, and enumerations (*e.g.*, red, yellow, or blue).

The primitive integer types in C include both unsigned and 2's complement representations. These types were originally defined so as to give reasonable performance when code was ported. In particular, the `int` type is intended to be the native integer type for the target ISA, which was often faster than trying to use larger or smaller integer types on a given machine. When C was standardized, these types were defined so as to include a range of existing C compilers rather than requiring all compilers to produce uniform results. At the time, most workstations and mainframes were 32-bit machines, while most personal computers were 16-bit, thus flexibility was somewhat desirable. With the GCC compiler on Linux, the C types are defined in the table above. Although the `int` and `long` types are usually the same, there is a semantic difference in common usage. In particular, on most architectures and most compilers, a `long` has enough bits to hold a pointer, while an `int` may not. When in doubt, the **size in bytes** of any type or variable can be found using the built-in C function `sizeof`.

	2's complement	unsigned
8 bits	`char`	`unsigned char`
16 bits	`short` `short int`	`unsigned short` `unsigned short int`
32 bits	`int`	`unsigned` `unsigned int`
32 or 64 bits	`long` `long int`	`unsigned long` `unsigned long int`
64 bits	`long long` `long long int`	`unsigned long long` `unsigned long long int`

Over time, the flexibility of size in C types has become less important (except for the embedded markets, where one often wants even more accurate bit-width control), and the fact that the size of an `int` can vary from machine to machine and compiler to compiler has become more a source of headaches than a helpful feature. In the late 1990s, a new set of fixed-size types were recommended for inclusion in the C library, reflecting the fact that many companies had already developed and were using such definitions to make their programs platform-independent. Although these types—shown in the table above—are not yet commonplace, we encourage you to make use of them. In Linux, they can be used by including the `stdint.h` header file in user code or the `linux/types.h` header file in kernel code.

	2's complement	unsigned
8 bits	`int8_t`	`uint8_t`
16 bits	`int16_t`	`uint16_t`
32 bits	`int32_t`	`uint32_t`
64 bits	`int64_t`	`uint64_t`

Floating-point types in C include `float` and `double`, which correspond respectively to single- and double-precision IEEE floating-point values. Although the 32-bit `float` type can save memory compared with use of 64-bit `double` values, C's math library works with double-precision values, and single-precision data are uncommon in scientific and engineering codes. Single-precision floating-point currently dominates the graphics industry, but even there support for double-precision will be prevalent soon.

Pointer types in C are formed by appending asterisks to the name of a type. For example, an `int*` is a pointer to an `int`, and a `double*` is a pointer to a `double`. The `void*` type is a generic pointer, and serves as a generic memory address. Pointers can, of course, also point to pointers. For example, a `short***` is a pointer to a pointer to a pointer to a `short` (read it from right to left). Using more than two levels of indirection in this manner is rare, however.

C Variables

Variables can be defined in one of two places in C: outside of any function, or at the start of a compound statement (a sequence of statements enclosed by braces). Historically, and in contrast with C++, variables in C could not be defined in the middle of compound statements, nor in expressions (*e.g.*, `for` loops). Although permitted by recent standards, *declaring variables haphazardly tends to obscure type information*.

The **scope** of a variable specifies what parts of a program can access the variable, while its **storage class** specifies when and where in memory the variable is stored. For C variables, both scope and storage class depend on where the variable is defined and upon whether or not the variable definition is preceded by the qualifier `static`.

Variables defined in compound statements, including function bodies, are accessible only within the compound statement in which they are defined. Defining a variable with the same name as another variable with a broader (but enclosing) scope **shadows** the second variable, making it inaccessible within the first variable's scope. Shadowing variables makes your code harder to read and more likely to contain bugs.

A variable defined outside of a function is accessible by any code in the file after its definition. If the variable is not preceded by the `static` qualifier, it is a **global variable**, and is also accessible from other files provided that the variable has been declared before being used. Here we distinguish between a **variable definition**, which results in the creation of a new variable, and a **variable declaration**, which makes a variable defined elsewhere accessible. In C, variable declarations look the same as definitions, but are preceded by the keyword `extern`. Header files usually include declarations, for example, but not definitions, since *a variable definition in a header file produces a separate copy of the variable for every file that includes the header file*. Global variables should be used sparingly, if at all.

Variables in C are stored either on the stack or in a static data region allocated when the program is loaded into memory. Variables declared outside of functions are placed in the static data region, as are variables declared in compound statements *and preceded by the* `static` *qualifier*. For any variable declared in a compound statement and without the `static` qualifier, space is allocated on the stack as part of the function's **stack frame** whenever the enclosing function begins execution, and is discarded when the function returns.[2] Such variables have **automatic** storage class. Multiple copies may thus exist if a function can call itself recursively, but only one copy is accessible by name from any given execution of the function. Note that if a `static` variable is declared inside a compound statement, *only one copy exists.* That copy is then accessible from all executions of the enclosing function, even if the function calls itself recursively, and the copy persists even when the function is not currently executing, although no code outside of the compound statement can access the variable by name.

The short sample code below illustrates variable declarations and definitions.

```
extern int    a_declaration;        /* must be defined elsewhere */
       int*   a_global_variable;
static double a_file_scope_variable;

int a_function (short int arg)
{
    static long int function_var;         /* unique copy in static data area */
           short    function_var_on_stack; /* one copy per call to a_function */
    {
        unsigned char another_stack_var;
        if (arg == 0)
            return 0;
    }
    /* another_stack_var is not accessible here.  */
    function_var_on_stack = arg - 1;
    return a_function (function_var_on_stack);
}
```

[2] Stack frames, also known as **activation records**, are covered in detail for the x86 ISA in the next set of notes.

Structures and Arrays

Structures can be defined in C using the `struct` keyword. Structure names technically occupy a separate **name space** from variables, *i.e.*, one can use a variable and a structure with identical names without syntactic errors, but overloading the meaning of a symbol in this way makes your code hard to read and should be avoided. If not explicitly defined as a new type, as discussed in the next section, structures used to define variables must include the `struct` keyword in the type name. For example, one can define a structure to represent two dates and then define double-date variables as shown to the right.

```
struct two_date_t {
    short birth_day;
    short birth_month;
    short birth_year;
    short hiring_day;
    short hiring_month;
    short hiring_year;
};

struct two_date_t my_data, your_data;
```

Defining a structure does not cause the compiler to allocate storage for such a structure. Rather, the definition merely tells the compiler the types and names of the structure's **fields**. Variable declarations for variables with these new types look and are interpreted no differently than variable declarations for primitive data types such as `int`.

A structure definition enables the compiler to generate instructions that operate on the fields of the structure. For example, given a pointer to a double-date structure, the compiler must know how much to add to the pointer to find the address of the `hiring_month` field. The compiler must ensure that the instructions do not violate rules imposed by the architecture, such as address alignment. For example, most ISAs use 8-bit-addressable memory but require that 32-bit loads and stores occur as if the memory were 32-bit-addressable, *i.e.*, only using addresses that are multiples of four. A processor for such an ISA is in fact not capable of performing an unaligned load, and the compiler must avoid generating code that requires any. It does so by inserting padding bytes into the structure to guarantee proper field alignment for the target ISA. *Making assumptions about fields offsets and size for a structure is thus risky business and should be avoided.* Use the `sizeof` function to find a structure's size and the field names to access its fields.

As with pointers, arrays of any type can be defined in C by simply appending one or more array dimensions to a variable definition:

```
int array[10];                  /* an array of 10 integers                          */
int multi[5][9];                /* an array of 5 arrays of 9 integers               */
struct two_date_t example[7][5]; /* an array of 7 arrays of 5 double-date structures */
```

Arrays are laid out sequentially in memory, and the name of the array is equivalent to a pointer to the first subarray (*i.e.*, of one fewer dimensions). With the examples above, `array` is a pointer to the first of ten `int`s, `multi` is a pointer to the first of five arrays of nine `int`s, and `example` is a pointer to the first of seven arrays of five `two_date_t` structures.

Defining New Types in C

As we've just seen, defining and using simple structures and arrays does not require the creation of new named types, but building complex data structures without intermediate names for the sub-structures or arrays can be confusing and error-prone. New types in C can be defined using the **type definition** command `typedef`, which looks exactly like a variable declaration, but instead defines a new data type.

Examples appear to the right. The first line creates two new types to represent the width and height of a screen, both of which are simply `int`s. The second line defines our double-date structure as a structure in its own right, allowing us to define variables, pass parameters, *etc.*, without writing `struct` over and over again. The last line defines a chess board as an 8-by-8 array of integers. The use of "_t" in the type names is a convention often used to indicate to programmers that the name is a type rather than a variable.

```
typedef int screen_width_t, screen_height_t;
typedef struct two_date_t two_date_t;
typedef struct int chessboard_t[8][8];
```

Enumerations are another convenient method for defining symbolic names for various sets or one-bit flags. By default, constants in an `enum` start at 0 and count upwards by 1, but any can be overridden by assignment, and the symbols need not have unique values.

A few examples follow. The left example defines three constants and a fourth symbol that can be used to size arrays to hold one value (*e.g.*, a human-readable name) for each of the constants. Constants can also be added or removed, and the last value changes automatically, enabling the compiler to check that arrays have also been updated when such changes are made. The symbolic names are also passed into the debugger, allowing you to use them with variables of type `constant_t`. The middle example defines specific numeric values, some of which are identical. The right example defines a set of values for use as one-bit flags.

```
typedef enum {
    CONST_A,     /* 0 */
    CONST_B,     /* 1 */
    CONST_C,     /* 2 */
    NUM_CONST    /* 3 */
} constant_t;
```

```
typedef enum {
    VALUE_A = 7, /* 7 */
    VALUE_B,     /* 8 */
    VALUE_C = 6, /* 6 */
    VALUE_D      /* 7 */
} value_t;
```

```
typedef enum {
    FLAG_A = 1,  /* 1 */
    FLAG_B = 2,  /* 2 */
    FLAG_C = 4,  /* 4 */
    FLAG_D = 8   /* 8 */
} flag_t;
```

C Operators

Basic **arithmetic operators** in C include addition (`+`), subtraction (`-`), negation (a minus sign not preceded by another expression), multiplication (`*`), division (`/`), and modulus (`%`). No exponentiation operator exists; instead, library routines are defined for this purpose as well as for a range of more complex mathematical functions.

C also supports **bitwise operations** on integer types, including AND (`&`), OR (`|`), XOR (`^`), NOT (∼), and left (<<) and right (>>) bit shifts. Right shifting a signed integer results in an **arithmetic right shift** (the sign bit is copied), while right shifting an unsigned integer results in a **logical right shift** (0 bits are inserted).

A range of **relational** or **comparison operators** are available, including equality (==), inequality (`!=`), and relative order (<, <=, >=, and >). All such operations evaluate to 1 to indicate a true relation and 0 to indicate a false relation. Any non-zero value is considered to be true for the purposes of tests (*e.g.*, in an `if` statement or a `while` loop) in C.

Assignment of a new value to a variable or memory location uses a single equal sign (=) in C. *The use of two equal signs for an equality check and a single equal sign for assignment is a common source of errors*, although modern compilers generally detect and warn about such mistakes. For an assignment, the compiler produces instructions that evaluate the right-hand expression and copy the result into the memory location corresponding to the left-hand expression *The value to the left must thus have an associated memory address.* Binary operators extended with an equal sign can be used in C to imply the use of the current value of the left side of the operator as the first operand. For example, "`A += 5`" adds 5 to the current value of the variable `A` and stores the result back into the variable.

Increment (`++`) and decrement (`--`) operators change the value of a variable (or memory location) and produce a result. If the operator is placed before the expression (a **pre-increment**), the expression produces the new, incremented value. If the operator is placed after the expression (a **post-increment**), the expression producs the original, unincremented value. The decrement operator works identically, and both of course require that the expression be stored in memory (*e.g.*, use a variable, such as `A++`).

The address of any expression with an address can be obtained with the **address operator** (`&`), and the contents of an address can be read using the **dereference operator** (`*`). Combining pointers and addresses with assignments confuses many. Shown to the right are a few examples translating C assignments on variables `A` and `B` into RTL. Note that the variable names in the RTL correspond to the memory addresses at which the variables are stored.

```
  A = B;   /* M[A] <- M[B]                          */
  A = *B;  /* M[A] <- M[M[B]]                       */
  A = **B; /* M[A] <- M[M[M[B]]]                    */
  A = &B;  /* M[A] <- B                             */
  A = &&B; /* not legal: &B has no address          */
  A = *&B; /* M[A] <- M[B]                          */
  A = &*B; /* M[A] <- M[B] (B must be a pointer)    */
 *A = B;   /* M[M[A]] <- M[B]                       */
**A = B;   /* M[M[M[A]]] <- M[B]                    */
 &A = B;   /* not legal: &A has no address          */
```

The C language supports **pointer arithmetic**, meaning that addition and subtraction are defined for pointer types. The compiler multiplies the value added (or subtracted) by the size of the type to which the pointer points, then adds (or subtracts) the result from the address to produce the new pointer. Thus adding one to a pointer into an array produces a pointer to the next element of the array.

Two operators are used for **field access** with structures. With a structure, appending a period (`.`) followed by a field name accesses the corresponding field. With a structure pointer, one can dereference the structure and then access the field, but the notation is cumbersome, so a second operator (`->`) is available for this purpose. The code to the right gives a few examples using our double-date structure.

```
two_date_t  my_data;
two_date_t* my_data_ptr = &my_data;

my_data.hiring_day = 21;
(*my_data).hiring_month = 7;
my_data->hiring_year = 1998;
```

Several **logical operators** are available in C to combine the results of multiple test conditions: logical AND (`&&`), OR (`||`), and NOT (`!`). As with relational operators, logical operations evaluate to either true (1) or false (0). The AND and OR operators use **shortcut evaluation**, meaning that the code produced by the compiler stops evaluating operands as soon as the final result is known. As you know, ANDing any result with 0 (false in C) produces a false result. Thus, if the first operand of a logical AND is false, the second operand is not evaluated. Similarly, if the first operand of a logical OR is true, the second operand is not evaluated. Shortcutting simplifies code structure when some expressions may only be evaluated safely assuming that other conditions are true. For example, if you want to read from a stream `f` into an uninitialized buffer `buf` and then check for the presence of the proper string, you can write:

```
if (NULL != fgets (buf, 200, f) && 0 == strcmp (buf, "absquatulate")) {
    /* found the word!  */
}
```

Calling the string comparison function `strcmp` on an uninitialized buffer is not safe nor meaningful, thus if the file is empty, *i.e.*, if `fgets` returns `NULL`, the second call should not be made. Shortcutting guarantees that it is not.

Example: Linked List Removal Function in C

Let's now revisit our instruction-level linked list example and develop it as a function in C. First, we'll need to extend our double-date structure to include a pointer.

As shown to the right, the type definition can precede the structure definition—in fact, by putting the type definition in a header file and keeping the structure definition in the source file containing all structure-related functions, you can prevent other code from making assumptions about how you have implemented the structure.

```
typedef struct two_date_t two_date_t;

struct two_date_t {
    short birth_day;
    short birth_month;
    short birth_year;
    short hiring_day;
    short hiring_month;
    short hiring_year;

    /* pointer to next structure in linked list */
    two_date_t* link;
};
```

The next step is to develop the **function signature** for the linked list removal function. A function signature or **prototype** specifies the return type, name, and argument types for a function, but does not actually define the function. This information is enough for the compiler to generate calls to the function as well as to perform basic checks for correct use and to generate necessary type conversions, as discussed later.

The information needed as input to our function consists of a pointer to the element to be removed and a pointer to the head of the linked list. The element being removed might be the first in the list, and our function will be more useful if it can also update the variable holding this pointer rather than requiring every caller to do so.

As you may recall, C arguments are **passed by value**, meaning that the arguments are evaluated and that the resulting values are copied (usually onto the stack) when a function is called. No C function can change the contents of a variable based only on the value of the variable, thus we must instead pass a pointer to the pointer to the head of the linked list, allowing the function to use this address to change the head as necessary:

```
/* function returns 0 on success, -1 on failure (e.g., element not in list) */
int remove_two_date (two_date_t** head_ptr, two_date_t* elt);
```

The code for the function is not much different structurally from that developed earlier for an instruction-level implementation, but allows us to express the approach a little more clearly, and with explicit information about data types:

```
    /* function returns 0 on success, -1 on failure (e.g., element not in list) */
    int remove_two_date (two_date_t** head_ptr, two_date_t* elt)
    {
        two_date_t** ptr_to_link;
        two_date_t*  current;
        int          ret_val = -1;

        ptr_to_link = head;
        current = *ptr_to_link;
        while (NULL != current) {
            if (elt == current) {
                *ptr_to_link = elt->next;
                current     = NULL;
                ret_val     = 0;
            } else {
                ptr_to_link = &current->next;
                current    = *ptr_to_link;
            }
        }
        return ret_val;
    }
```

A more succinct version is also possible using similar optimizations as before, but the syntax of multiple levels of indirection, *i.e.*, pointers to pointers, does tend to be a little confusing at first:

```
/* function returns 0 on success, -1 on failure (e.g., element not in list) */
int remove_two_date (two_date_t** head_ptr, two_date_t* elt)
{
    two_date_t** ptr_to_link;

    for (ptr_to_link = head; NULL != *ptr_to_link; ptr_to_link = &(*ptr_to_link)->next) {
        if (elt == *ptr_to_link) {
            ptr_to_link = elt->next;
            return 0;
        }
    }
    return -1;
}
```

Changing Types in C

Changing the type of a datum is necessary from time to time, but sometimes a compiler can do the work for you. The most common form of **implicit type conversion** occurs with binary arithmetic operations. Integer arithmetic in C always uses types of at least the size of `int`, and all floating-point arithmetic uses `double`. If either or both operands have smaller integer types, or differ from one another, the compiler implicitly converts them before performing the operation, and the type of the result may be different from those of both operands. In general, the compiler selects the final type according to some preferred ordering in which floating-point is preferred over integers, unsigned values are preferred over signed values, and more bits are preferred over fewer bits. The type of the result must be at least as large as either argument, but is also at least as large as an `int` for integer operations and a `double` for floating-point operations.

Modern C compilers always extend an integer type's bit width before converting from signed to unsigned. The original C specification interleaved bit width extensions to `int` with sign changes, thus *older compilers may not be consistent, and implicitly require both types of conversion in a single operation may lead to portability bugs.*

The implicit extension to `int` can also be confusing in the sense that arithmetic that seems to work on smaller integers fails with larger ones. For example, multiplying two 16-bit integers set to 1000 and printing the result works with most compilers because the 32-bit `int` result is wide enough to hold the right answer. In contrast, multiplying two 32-bit integers set to 100,000 produces the wrong result because the high bits of the result are discarded before it can be converted to a larger type. For this operation to produce the correct result, one of the integers must be converted explicitly (as discussed later) before the multiplication.

Implicit type conversions also occur due to assignments. Unlike arithmetic conversions, the final type must match the left-hand side of the assignment (*e.g.*, a variable to which a result is assigned), and the compiler simply performs any necessary conversion. *Since the desired type may be smaller than the type of the value assigned, information can be lost.* Floating-point values are truncated when assigned to integers, and high bits of wider integer types are discarded when assigned to narrower integer types. *Note that a positive number may become a negative number when bits are discarded in this manner.*

Passing arguments to functions can be viewed as a special case of assignment. Given a function prototype, the compiler knows the type of each argument and can perform conversions as part of the code generated to pass the arguments to the function. Without such a prototype, or for functions with variable numbers of arguments, the compiler lacks type information and thus cannot perform necessary conversions, leading to unpredictable behavior. By default, however, the compiler extends any integers to the width of an `int`.

Pointer types can also be automatically converted, but such conversions are much more restrictive than those for numeric types. Automatic conversion of pointer types occurs only for assignment (and argument passing), and is only allowed when one of the pointers has type `void*`.

Occasionally it is convenient to use an **explicit type cast** to force conversion from one type to another. *Such casts must be used with caution, as they silence many of the warnings that a compiler might otherwise generate when it detects potential problems.* One common use is to promote integers to floating-point before an arithmetic operation, as shown to the right. The type to which a value is to be converted is placed in parentheses in front of the value. In most cases, additional parentheses should be used to avoid confusion about the precedence of type conversion over other operations.

```
{
    int numerator = 10;
    int denominator = 20;

    printf ("%f\n", numerator / (double)denominator);
}
```

Explicit type casts can also be used to force the compiler to treat pointers as integers, and to treat integers as addresses (cast them back to pointers). Obviously, since the number of bits required for a pointer as well as structure sizes can vary based on ISA, operating system, and compiler, such casts must be used with caution.

The C Preprocessor

The C language uses a preprocessor to support inclusion of common information (stored in header files) within multiple source files as well as a macro facility to simplify unification of source code with multiple intended purposes, such as portability across instruction set architectures.

The most common use of the preprocessor is to enable the unique definition of new types, structures, and function prototypes within header files that can then be included by reference within source files that make use of them. This capability is based on the **include directive**, `#include`, as shown here:

```
#include <stdio.h>       /* search in standard directories                    */
#include "my_header.h"   /* search in current followed by standard directories */
```

The preprocessor also supports integration of compile-time constants into the original source files before compilation. For example, many software systems allow the definition of a symbol such as `NDEBUG` (no debug) to compile without additional debugging code included in the sources.

Two directives are necessary for this purpose: the **define directive**, `#define`, which provides a text-replacement facility, and **conditional inclusion** (or exclusion) of parts of a file within `#if`/`#else`/`#endif` directives.

These directives are also useful in allowing a single header file to be included multiple times without causing problems, as C does not generally allow redefinition of types, variables, *etc.*, even if the definitions match. Most header files are thus wrapped as shown to the right.

```
#if !defined(MY_HEADER_H)
#define MY_HEADER_H
/* actual header file material goes here */
#endif /* MY_HEADER_H */
```

The preprocessor performs a simple linear pass on the source and does not parse or interpret any C syntax. Definitions for text replacement are valid as soon as they are defined and are performed until they are undefined or until the end of the original source file. The preprocessor does recognize spacing and will not replace part of a word, thus "`#define i 5`" will not wreak havoc on your `if` statements.

Using the text replacement capabilities of the preprocessor does have drawbacks, most importantly in that almost none of the information is passed on for debugging purposes. Lists of defined constants have thus been deprecated; use an `enum` instead, which in combination with `typedef` enables your debugger to support symbolic names rather than forcing you to repeatedly translate.

In contrast, many systems still make use of parametrized text replacement, also known as **preprocessor macros**. A macro definition includes a comma-separated list of arguments after the macro name. The arguments can then be used in the replacement text. Consider, for example, the following definition of a macro to calculate the cube of a number:

```
#define CUBE(a)    ((a) * (a) * (a))    /* correct definition                      */
#define CUBE (a)   ((a) * (a) * (a))    /* no space allowed before arguments       */
#define CUBE(a)    (a * a * a)          /* CUBE (4 + 3) -> (4 + 3 * 4 + 3 * 4 + 3) */
#define CUBE(a)    (a) * (a) * (a)      /* similar potential precedence problems   */
```

Macros provide two main advantages in C. First, as C provides little support for variable types, macros provide a way to emulate such support. For example, to define the same arithmetic algorithm on both integers (faster) and floating-point numbers (better dynamic range), one has to either copy the code or make use of macros to produce the two versions. Similar usages are attractive when developing types of objects with some shared properties, such as instructions in a simulator or widgets in a drawing tool. However, as with other uses of define directives, most systems do not pass information about macros to debuggers. In languages like C++ and Java, templates or simply class hierarchies can be used instead, but sometimes at the cost of performance.

Some of the potential pitfalls with macros are shown in the examples above. More subtle errors are also possible, which is why people generally use a different convention for macro names (*e.g.*, all capital letters) than for function names. Consider, for example, "`CUBE(i++)`"—after text replacement, the variable `i` is incremented not once but three times, with unpredictable results.

Defining macros for more than expressions can be problematic due to interference between text replacement and C syntax. For example: unlike functions, macros can change the values of their arguments by simply replacing the macro call with an assignment, which can be confusing for programmers not familiar with macros.

Consider the range-limiting variable update shown to the right. The backslashes at the end of each line are necessary to tell the preprocessor that the macro definition continues on the next line. This macro avoids the argument issue by requiring a pointer to the variable to be changed. However, writing a multi-line macro such as that shown above poses several additional problems.

```
#define BACKOFF(_w)          \
        *(_w) *= 0.5;        \
        if (*(_w) < 1E-6) {  \
            *(_w) = 1E-6;    \
        }
```

For example, the macro as defined breaks if used as a simple statement:

```
if (should_backoff)
    BACKOFF (&var);            /* semicolon is NOT part of the macro */
else
    SOMETHING_ELSE (&var);
```

Wrapping the macro up as a compound statement (*i.e.*, in braces: { ... }) does not solve the problem because of the spurious semicolon in the user's code.

Instead, we wrap the entire macro as a `do {} while (0)` loop, as shown to the right. This approach works well in the sense that the macro is fully equivalent to a function call returning `void`: it is a single statement from the compiler's point of view, but must be followed by a semicolon when used. Also, most modern compilers will not generate additional instructions for the wrapper code, even when compiling without optimization.

```
#define BACKOFF(_w)             \
        do {                    \
            (_w) *= 0.5;        \
            if ((_w) < 1E-6) {  \
                (_w) = 1E-6;    \
            }                   \
        } while (0)
```

One final use of macros is worth discussing here: their use in supporting assertions for debugging. The GNU preprocessor used with `gcc` supports additional symbols that identify the current file and line of the macro instance, thus an assertion defined as follows can print the file name and line number of the failed assertion:

```
#define ASSERT(x)                                                           \
    do {                                                                    \
        if (!(x)) {                                                         \
            fprintf (stderr, "ASSERTION FAILED: %s:%d\n", __FILE__, __LINE__);  \
            abort ();                                                       \
        }                                                                   \
    } while (0)
```

Terminology

You should be familiar with the following terms after reading this set of notes.

- information storage in computers
  - bits
  - representation
  - data type
  - data structure
  - fields
  - size of a type
  - addressability of memory
- using memory addresses
  - pointer
  - dereference
  - pointer-array duality
  - linked list
- transforming tasks into programs
  - systematic decomposition (sequential, iterative, and conditional tasks)
  - flow chart
  - register transfer language (RTL)
  - invariant
  - stopping condition
- high-level language data types
  - strongly typed languages
  - primitive data types
  - sizeof built-in function
- structures, arrays, and new types in C
  - data structure in C
  - `struct` keyword
  - fields
  - enumerations
  - type definition
  - name space
- variables in C
  - declaration
  - definition
  - scope
  - storage class
  - shadowing
  - `static` qualifier
  - global variable
  - automatic (stack) variable
  - stack frame/activation record
- C operators
  - arithmetic
  - bitwise
  - comparison/relational
  - assignment
  - pre- and post-increment and decrement
  - address and dereference
  - pointer arithmetic
  - structure field access
  - logical
  - shortcutting
- functions in C
  - function signature/prototype
  - pass by value
- type conversion
  - implicit type conversion
  - explicit type cast
- the C preprocessor
  - include directive
  - define directive
  - preprocessor macro