6.004 Computation Structures
Spring 2009

6.004 uses the C language, developed at Bell Telephone Laboratories, for the informal presentation of algorithms and program fragments. C is quite typical of modern compiled languages, and any reader with some programming experience should find the examples fairly readable. This overview is provided as an aid to understanding the simple C programs presented in the lectures. It falls far short of a complete introduction to the language, for which the interested reader is referred to Kernighan and Ritchie [1978].

C is a relatively low-level language, designed to be easily translated to efficient object code for many modern computers. Unlike LISP, for example, the semantics of C strongly reflect the assumption of compiled (rather than interpreted) implementations and a bias toward simplicity and efficiency of the translated program rather than flexibility and generality of the source language. In these respects, C is typical of languages in widespread use for production programming; other examples include FORTRAN, Pascal, and PL/1.

Simple Data Types and Declarations

C offers several representations of integer data, differing in the number of bits used. The data types char, short, and long designate, respectively, 8-, 16-, and 32-bit signed integer data. The type int designates signed integer data of an implementation-dependent size; it is equivalent to short in some C implementations (typically those for 16-bit machines) and to long in others.

Every variable in C has an associated compile-time type. Unlike some interpreted languages (for example, LISP), the type information in C is used only to make compile-time decisions regarding the object code (machine instructions) to be generated; the type information is not manipulated during program execution. A variable may be *declared* in C by declarations of the form:

```
short x, y, z;      /* Declare three 16-bit integers. */
long a, b = 13;     /* Declare two 32-bit integers.   */
```

This C program fragment includes two declarations, signaling to the compiler that 16-bit storage locations are to be reserved for the storage of `x`, `y`, and `z` and that `a` and `b` require 32-bit storage. The `= 13` clause in the declaration of `b` specifies an initial value; if absent, the initial value will be random. Note the use of `/* ... */` syntax to incorporate comments into the source program.

The keyword `unsigned` may precede the type in a declaration; it serves as an adjective to notify the compiler that the declared variables represent natural numbers rather than signed integers. This information affects the compiler's choice of instructions in certain operations, such as comparisons and right shifts.

C programs use integer data to represent Boolean (true/false) conditions. In general, zero is taken to mean *false* and nonzero represents *true*.

Typed *pointers* can be manipulated in C. A pointer to a datum is represented at run time by a word containing the machine address of that datum; if the datum occupies multiple (consecutive) bytes, the pointer contains the *lowest* address occupied by the data to which it points. Pointer variables are declared using one or more asterisks preceding the variable name. Thus the declaration

```
long a, *b, **c;
```

notifies the compiler that the variable `a` will have as its value a 32-bit integer, `b` will have as its value a *pointer* to a 32-bit integer (that is, the address of a memory location containing a long integer), and `c`'s value will be a pointer to a location containing a pointer to a long integer.

A *procedure* in C may return a value and is declared with a type corresponding to the value returned. The following is a typical procedure declaration:

```
long add1(a)
 long a;
 {
       return a+1;
 }
```

Note that the name `add1` of the procedure is preceded by the type of its returned value; this may be omitted if the procedure returns no value. Following the procedure name is a list of dummy arguments, separated by commas and surrounded by parentheses; these names become variables whose declaration immediately follows the argument list and whose scope is the *body* of the function, enclosed in braces.

Expressions

The simplest C expressions consist of single constants or variables; `123` and `x` are each valid C expressions (assuming that `x` has been declared as a variable). More complicated expressions can be constructed using C *operators*, such as `+`, `-`, `*`, and `/`, which designate the usual arithmetic operations. Thus `x/3+4` is an expression whose *value* (computed during program execution) is four plus the quotient of the value of `x` and three. Examples of expressions using C operators are given in the following table.

Expression	*Value*
`a + b`	Addition
`a - b`	Subtraction
`-a`	2's complement (negative)
`a * b`	Multiplication
`a / b`	Division
`a % b`	Modulus
`(a)`	Value of `a`; parenthesis used for grouping
`a < b`	True (nonzero) if `a` is less than `b`, else false
`a > b`	True (nonzero) if `a` is greater than `b`, else false
`a <= b`	Less-than-or-equal-to comparison
`a >= b`	Greater-than-or-equal-to comparison

`a == b`	Equal-to comparison (don't confuse with assignment =)
`a != b`	Not-Equal-to comparison
`!a`	True (nonzero) if `a` is false (zero); Boolean *not*
`a && b`	Wordwise AND: false (zero) if either `a` or `b` is false
`a || b`	Wordwise OR: true (nonzero) if either `a` or `b` is true
`~a`	Bitwise complement of `a`
`a & b`	Bitwise AND
`a | b`	Bitwise OR
`a >> b`	Integer `a` shifted right `b` bit positions
`a << b`	Integer `a` shifted left `b` bit positions
`x = a`	Assignment: has value `a`, but sets the current value of `x` to the value of `a` (don't confuse with equal-to comparison ==)
`&x`	Address of the variable `x`
`*p`	Contents of the location whose address is `p`
`f(a,b,...)`	Procedure call
`p[a]`	Array reference: element `a` of array `p`
`x.c`	Component `c` of structure `x`
`p->c`	Component `c` of structure pointed to by `p`
`sizeof x`	Size, in bytes, of the representation of `x`

In this table, `a`, `b`, `f`, and `p` may be replaced by valid expressions, while `x` must be a variable (since the storage location associated with it is referenced). `c` is a structure component name (see Structures). Note that = dictates assignment but is syntactically an operator; thus simple expressions such as `3*(x=x+1)` may have side effects as well as values.

The unary operators `*` and `&` can be used for referencing through and creating pointers. Thus, if `a` is an expression of type `t`, `&a` is an expression of type *pointer to* `t`. Conversely, if `a` is of type *pointer to* `t`, then `*a` is an expression of type `t`.

The form `f(a,b,...)` denotes a *procedure call* and is an expression whose type is that declared for the function `f`. C procedure arguments are passed by value; thus the *values* (rather than the addresses) of the arguments (`a$`, `b`, and so on) are bound to the formal parameters of `f` during its execution.

Statements and Programs

A *statement* in C is a single imperative or declarative command. Unlike an expression, a statement has no value; its purpose in the program stems from the effect it has, either during compilation or during execution. We have seen variable declarations, which typify the declarative class; the simplest example of an imperative statement is

an expression followed by a semicolon:

```
a = b+3;
```

This causes the evaluation of the given expression during program execution. Such a statement is of interest to the programmer, of course, because its execution causes the *side effect* of changing `a`'s value. While a side-effect-free expression such as `a+3` can be made into a syntactically valid C statement by appending a semicolon, the exercise is clearly silly.

Compound statements can be constructed by using braces to enclose a sequence of component statements that are to be treated as a syntactic unit. Thus

```
{ temp=a; a=b; b=temp;
}
```

is a statement that has the effect of exchanging the values of `a` and `b`; we assume, of course, that `a`, `b`, and `temp` have been appropriately declared. Note that the body of a procedure, as specified in a procedure declaration, is just such a compound statement.

Each compound statement is, in fact, a program *block* and may begin with declarations of variables that are local to that block. Thus the exchange of values in the above example might better be written as

```
{ int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Here the temporary location is local to the block, minimizing externally visible side effects. Note that variables declared inside a block are *dynamic*; they are allocated (from the stack) when the block is entered and are deallocated on exit from the block. Variables declared outside any block are *static*; they are effectively allocated at compile time and retain their values during the entire program execution. Variables declared with initial values (using the "`=` *constant*" declaration syntax) are initialized at allocation; a dynamic variable with a specified initial value is thus reinitialized at each entry to the block.

Conditional execution can be specified using an `if` statement, optionally including an `else` clause, as follows:

```
if (a < b) biggest = b;
else biggest = a;
```

The condition in the `if` statement is enclosed in parentheses and immediately follows the keyword `if`. At execution time, the conditional expression is evaluated; if it is true (nonzero), then the statement following the condition (the *then clause* of the `if` statement) is executed; otherwise it is skipped. The body of an `else` clause is executed if and only if the then clause of the immediately preceding `if` statement is skipped. Of course, the

then clause and the `else` clause bodies may each be (and often are) program blocks.

We shall use two C constructs for program loops:

```
while (cond) statement;
```

and

```
for (init; test; increment) statement;
```

The `while` statement repeatedly evaluates the conditional expression *cond*, executing *statement* once after each evaluation of *cond* until *cond* evaluates to false (zero). The `for` statement is semantically equivalent to

```
init;
while (test)
 {      statement;
        increment;
 }
```

and provides a syntactically compact form for small loops.

A C *program* consists of a set of declarations of procedures and data. The following C code is a simple example, using two versions of the Fibonacci function to illustrate these constructs.

```
/* Fibonacci -- recursive version                */
long rfib(n)
long n;
{       if (n>1) return rfib(n-1) + rfib(n-2);
        else return n;
}

/* Fibonacci -- iterative version                */
long ifib(n)
long n;
{
        if (n<2) return n;
        else    { long val0, val1=0, val2=1, i;
                  for (i=1; i<n; i = i+1)
                        {       val0 = val2;
                                val2 = val2 + val1;
                                val1 = val0;
                        }
                  return val2;
```

```
                }
}
```

C provides a multiway branch construct, the `switch` statement, allowing a given expression to be tested for a fixed number of constant values. The syntax of `switch` is illustrated by the next program, yet another Fibonacci function, which uses `switch` to handle certain commonly occurring arguments quickly. The `case` labels identify statements in the `switch` block to which control is transferred for various values of the `switch` expression; the value in each `case` label must be a compile-time constant. Control is transferred to the statement bearing the label `default:` if the switch value matches none of the `case` constants; in the example, the default action is to call one of the alternative Fibonacci procedures.

```
/* Fibonacci -- "quick" (for small argument) version */
long qfib(n)
long n;
{       switch (n)
         {      case 0:
                case 1:  return n;
                case 2:  return 1;
                case 3:  return 2;
                case 4:  return 3;
                case 5:  return 5;
                case 6:  return 8;
                default: return ifib(n);
         }
}
```

Several additional C statements are provided for low-level control. We have seen that `return` *expr*`;` returns from the innermost enclosing procedure with the optional value *expr*; `break;` can be used in a similar way to exit the innermost enclosing `for` or `while` loop or `switch` block. A `continue;` statement within a loop terminates the current iteration, effectively jumping to the end of the loop body to perform the indicated *increment* and *test* operations and continue with the next iteration, if appropriate. A `goto` *tag*`;` statement, where the label *tag*`:` identifies a statement local to the procedure, provides the lowest-level control mechanism. It is occasionally convenient (for example, to exit several nested loops), as illustrated in by the following program.

```
/* See if an x,y pair exists for which f(x) = g(y);
 * search each function's domain between 1 and the value for
 * which it returns 0.                                          */
Search(f, g)
 int f(), g();
 {      int x, y, fx, gy;
        for (x=0;  (fx = f(x)) != 0; x = x+1)
        {       for (y=0; ; y=y+1)
                {       if ((gy = g(y)) == 0) break;
                        if (fx != gy) continue;
```

```
                        else goto GotOne;     /* Found an answer!    */
                }
        }
        return 0;                             /* No answer found.    */
GotOne: return 1;                             /* Found an x,y pair.  */
 }
```

This program includes several notable features. First, it illustrates the use of *procedure* names as arguments; the procedures passed as `f` and `g` are presumed to be defined elsewhere in the program. (The actual arguments are, in fact, *pointers* to procedures; some C implementations require the somewhat inscrutable declaration syntax "`int(*f)()`;" to reflect this fact.) Second, the end test in the outer `for` loop requires application of one of the argument procedures; to avoid redundant evaluation of `f(x)` within the loop body, the resulting value is assigned to a local variable. The end test of this loop thus contains an *assignment*; it is important to distinguish between the assignment `fx=f(x)` and the syntactically similar equality test `fx==f(x)`. The corresponding end test of the inner `for` statement is left blank, resulting in two consecutive semicolon characters delimiting an empty expression. In this context, the missing conditional expression is equivalent to a nonzero constant expression, causing the loop to iterate forever unless other provisions are made. Execution of this inner loop is terminated by the explicit `if` statement within its body, which performs the assignment and test functions of the end test in the outer loop. The `break` statement in this conditional exits one level (back to the body of the outer loop) when `g(y)` evaluates to zero. The `continue` statement in the following `if` ends execution of the current iteration of the containing loop, continuing that loop with the next value of `y` if the `fx` and `gy` values differ. The `goto` in the `else` statement is used to exit both loops when a solution is found. In this example, the statement `goto GotOne;` could be replaced by a simple `return 1;`. Indeed, the latter coding is to be preferred since proliferation of `goto` statements can make program logic difficult to follow. In more complicated cases (for example, those involving substantial computation at `GotOne:` that is to be entered from a variety of different places), `goto` may be the most natural solution.

The following table summarizes a set of useful C statement types.

Statement type	*Use*
`expr;`	Evaluate expression *expr*
`if (`*`test`*`)` *`statement`*`;`	Conditional test
`else` *`statement`*`;`	Optionally follows `if` statement

`switch(`*`expr`*`) {`
`  case` *`C1`*`: ...`
`  case` *`C2`*`: ...`
`  ...`
`}`
	N-way branch (dispatch)

`while (`*`test`*`)` *`statement`*`;`	Iteration
`for (`*`init`*`;` *`test`*`;` *`incr`*`)` *`statement`*`;`	

`return` *`expr`*`;`	Procedure return; *expr* is optional
`break;`	Break out of loop or `switch`
`continue;`	Continue to next loop iteration
`tag``:`	Define a label for `goto`
`goto` *`tag`*`;`	Transfer control

Arrays and Pointers

In addition to simple scalar and pointer data, C provides two mechanisms for handling aggregate data objects stored in contiguous memory locations. The first of these is the single-dimensioned *array* or *vector*, declared using a bracketed constant following the variable name:

```
/* 100-employee payroll record          */
long    Salary[100];
char    *Name[100];
```

This fragment declares two 100-element arrays, comprising 100 consecutive 32-bit and 8-bit locations, respectively. The brackets serve also as the C operator for referencing an array element; thus `Salary[37]` designates element 37 of the `Salary` array. C arrays are *zero-indexed*, meaning that the first element of `Salary` is `Salary[0]`; thus `Salary[37]` is, in fact, the thirty-eighth element. `Salary[100]` is an out-of-bounds array reference since it refers to the 101st element of a 100-element array.

Using the array indexing operator, the `Salary` array might be cleared as follows:

```
{ int i;
  for (i=0; i<100; i=i+1) Salary[i] = 0;
}
```

From a type standpoint, C treats arrays as pointers; thus `Salary` will be treated in subsequent statements as a variable of type *pointer to* `long`. However, the declaration `long Salary[100];` has the additional effect of reserving 400 bytes of storage for the `Salary` array and permanently binding the value of `Salary` to the beginning (lowest) address of the reserved storage. Similarly, the declaration `char *Name[100];` causes `Name` to be treated in subsequent statements as if it had been declared `char **Name`, except that `Name` permanently identifies the start of a reserved memory block of sufficient size to hold 100 pointers. Since C views an array as a pointer to the first element of consecutive similarly typed data, it also allows a pointer to a block of memory to be used as an array. Thus, if a variable `p` is declared `char *p` and is set to point to a block of consecutive bytes of memory, references of the form `p[`*`i`*`]` can be used to access the *i*th element of `p`.

Arrays declared using the "`...`" syntax must have sizes that are compile-time constants. However, available library procedures allocate variable-size contiguous regions of memory dynamically (that is, during program execution) and return pointers to them; such library routines allow more flexible and efficient use of memory. The `Salary` array in the above example could be allocated at run time rather than at compile time by rewriting

it as

```
long    *Salary; /* Simple pointer variable */
...
/* Allocate storage (N elements) for array: */
Salary = malloc(N*sizeof *Salary);
/* Use array: */
for (i=0; i=100; i=i+1) Salary[i] = 0;
...
/* Return array to free storage pool:       */
free(Salary);
```

In this example, the storage for the `Salary` array is allocated by an executable statement (that is, at run time) rather than by a declaration. This technique makes convenient use of C's happy confusion between pointers and arrays; the identical syntax (for example, `Salary[i]`) can be used to reference the elements of `Salary` independently of the means by which its storage is allocated. The library procedures `malloc` and `free` are used in this example to deal with a program-managed heap storage pool: `p=malloc(s)` allocates an `s`-byte region and returns a pointer to its first location, while `free(p)` returns a previously allocated block to the pool.

Textual data are naturally represented in C as arrays of characters. Thus a text-string variable `text` would be declared `char *text`, and the *i*th character of its current (string) value can be referred to as `text[i]`. Constant text strings can be designated in C using the double-quote character `"`; thus `"fungus"` is a C constant of type `char *`.

Structures

In addition to arrays, which have homogeneous type but variable size, C supports *structures*, which are fixed-size records of heterogeneous data. A structure declaration provides a useful way to localize related data; our payroll data, for example, might naturally be given the type:

```
struct Employee {
  char *Name;     /* Employee's name.      */
  long Salary;    /* Employee's salary.    */
};
```

This fragment describes a structure type and gives it the name `Employee`. Subsequent declarations can treat `struct Employee` as a valid C type; for example,

```
struct Employee Payroll[100];
```

declares an array of 100 structures, each devoted to some employee's records.

It should be understood that a structure declaration provides the compiler with a prototype for the interpretation

of blocks of memory. In particular, each region of memory identified to the compiler as a `struct Employee` will begin with a pointer to a character string that holds the employee's name (occupying, say, the first 4 bytes of the region) and will contain in the immediately following 4 bytes a binary integer giving the employee's salary. The C syntax for referring to the components of a structure is "*`str.cname`*" where *str* designates a structure and *cname* is the name of one of its components. The salary of the fourth employee, for example, is referenced by `Payroll[3].Salary`.

It is often both convenient and efficient to manipulate pointers to structures rather than structures themselves. This is particularly true when some `struct` type must contain another value of the same type. We might, for example, wish to expand our little payroll data base to include the supervisor of each employee:

```
struct Employee {
  char *Name;                  /* Employee's name.      */
  long Salary;                 /* Employee's salary.    */
  struct Employee *Supervisor; /* Employee's boss.      */
} Payroll[100];
```

Our goal here is to identify in each payroll record the record of the employee's supervisor. While it is perfectly legal in C to declare a structure component that is itself a `struct` type, the result is that the containing structure type is enlarged by the size of the contained structure. If the supervisor component were declared to be type `struct Employee` rather than a pointer to `struct Employee`, C would complain. We would have asked it to allocate a structure large enough for several elements, one of which is the same size as itself!

We circumvent such difficulties by referencing the supervisor through a pointer to the appropriate payroll record. In addition, it is usually far preferable from a performance standpoint to pass pointers to structures (for example, as procedure arguments) rather than copies of the structures themselves. Since access to structures through pointers is so common, C provides the special syntax "*`p->cname`*" to reference component *cname* of the structure that *p* points to. Use of structure pointers is illustrated by the silly program shown below.

```
struct Employee {
  char *Name;                  /* Employee's name.     */
  long Salary;                 /* Employee's salary.   */
  long Points;                 /* Brownie points.      */
  struct Employee *Supervisor; /* Employee's boss.     */
}

/* Annual raise program.   */
Raise(p)
  struct Employee p[100];
 {      int i;
        for (i=0; i<100; i=i+1)    /* Consider each employee. */
         { p->Salary =
               p->Salary           /* Salary adjustment:      */
               + 100               /*  cost-of-living term,   */
               + p->Points;        /*  merit term.            */
```

```
            p->Points = 0;          /* Start over next year!   */
            p = p+1;                /* On to next record!      */
            Check(p);              /* Make sure no disparities. */
           }
 }

/* Make sure employee is getting less than boss:              */
Check(e)
  struct Employee *e;               /* Pointer to record.    */
 {
        if (e == e->Supervisor)     /* Ignore the president  */
                return;             /*  (pres. is own boss). */
        if (e->Salary <             /* Problem here?         */
            (e->Supervisor)->Salary)
                return;             /* Nope, leave happy.    */
        /* When e's boss is making no more than e is,
         * give boss a raise, then check that boss's
         * new salary causes no additional problems:
         */
        (e->Supervisor)->Salary =
                1 + e->Salary;      /* Now boss makes more.  */
        Check(e->Supervisor);       /* Check further.         */
 }
```