```
long rfun(unsigned long x) {
    if ( __________ )
        return __________;
    unsigned long nx = __________;
    long rv = rfun(nx);
    return __________;
}
```

GCC generates the following assembly code:

```
    long rfun(unsigned long x)
    x in %rdi
rfun:
  pushq   %rbx
  movq    %rdi, %rbx
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L2
  shrq    $2, %rdi
  call    rfun
  addq    %rbx, %rax
.L2:
  popq    %rbx
  ret
```

A. What value does `rfun` store in the callee-saved register `%rbx`?

B. Fill in the missing expressions in the C code shown above.

3.8 Array Allocation and Access

Arrays in C are one means of aggregating scalar data into larger data types. C uses a particularly simple implementation of arrays, and hence the translation into machine code is fairly straightforward. One unusual feature of C is that we can generate pointers to elements within arrays and perform arithmetic with these pointers. These are translated into address computations in machine code.

Optimizing compilers are particularly good at simplifying the address computations used by array indexing. This can make the correspondence between the C code and its translation into machine code somewhat difficult to decipher.

3.8.1 Basic Principles

For data type T and integer constant N, consider a declaration of the form

T `A[`N`];`

Let us denote the starting location as $x_{\mathtt{A}}$. The declaration has two effects. First, it allocates a contiguous region of $L \cdot N$ bytes in memory, where L is the size (in bytes) of data type T. Second, it introduces an identifier `A` that can be used as a pointer to the beginning of the array. The value of this pointer will be $x_{\mathtt{A}}$. The array elements can be accessed using an integer index ranging between 0 and $N-1$. Array element i will be stored at address $x_{\mathtt{A}} + L \cdot i$.

As examples, consider the following declarations:

```
char    A[12];
char   *B[8];
int     C[6];
double *D[5];
```

These declarations will generate arrays with the following parameters:

Array	Element size	Total size	Start address	Element i
A	1	12	$x_{\mathtt{A}}$	$x_{\mathtt{A}} + i$
B	8	64	$x_{\mathtt{B}}$	$x_{\mathtt{B}} + 8i$
C	4	24	$x_{\mathtt{C}}$	$x_{\mathtt{C}} + 4i$
D	8	40	$x_{\mathtt{D}}$	$x_{\mathtt{D}} + 8i$

Array `A` consists of 12 single-byte (`char`) elements. Array `C` consists of 6 integers, each requiring 4 bytes. `B` and `D` are both arrays of pointers, and hence the array elements are 8 bytes each.

The memory referencing instructions of x86-64 are designed to simplify array access. For example, suppose `E` is an array of values of type `int` and we wish to evaluate `E[i]`, where the address of `E` is stored in register `%rdx` and i is stored in register `%rcx`. Then the instruction

```
movl (%rdx,%rcx,4),%eax
```

will perform the address computation $x_{\mathtt{E}} + 4i$, read that memory location, and copy the result to register `%eax`. The allowed scaling factors of 1, 2, 4, and 8 cover the sizes of the common primitive data types.

Practice Problem 3.36 (solution page 377)

Consider the following declarations:

```
int    P[5];
short  Q[2];
int    **R[9];
double *S[10];
short  *T[2];
```

Fill in the following table describing the element size, the total size, and the address of element i for each of these arrays.

Array	Element size	Total size	Start address	Element i
P			$x_{\mathtt{P}}$	
Q			$x_{\mathtt{Q}}$	
R			$x_{\mathtt{R}}$	
S			$x_{\mathtt{S}}$	
T			$x_{\mathtt{T}}$	

3.8.2 Pointer Arithmetic

C allows arithmetic on pointers, where the computed value is scaled according to the size of the data type referenced by the pointer. That is, if `p` is a pointer to data of type T, and the value of `p` is $x_{\mathtt{p}}$, then the expression `p+i` has value $x_{\mathtt{p}} + L \cdot i$, where L is the size of data type T.

The unary operators '`&`' and '`*`' allow the generation and dereferencing of pointers. That is, for an expression *Expr* denoting some object, `&`*Expr* is a pointer giving the address of the object. For an expression *AExpr* denoting an address, `*`*AExpr* gives the value at that address. The expressions *Expr* and `*&`*Expr* are therefore equivalent. The array subscripting operation can be applied to both arrays and pointers. The array reference `A[i]` is identical to the expression `*(A+i)`. It computes the address of the ith array element and then accesses this memory location.

Expanding on our earlier example, suppose the starting address of integer array `E` and integer index i are stored in registers `%rdx` and `%rcx`, respectively. The following are some expressions involving `E`. We also show an assembly-code implementation of each expression, with the result being stored in either register `%eax` (for data) or register `%rax` (for pointers).

Expression	Type	Value	Assembly code
`E`	`int *`	$x_{\mathtt{E}}$	`movl %rdx,%rax`
`E[0]`	`int`	$M[x_{\mathtt{E}}]$	`movl (%rdx),%eax`
`E[i]`	`int`	$M[x_{\mathtt{E}} + 4i]$	`movl (%rdx,%rcx,4),%eax`
`&E[2]`	`int *`	$x_{\mathtt{E}} + 8$	`leaq 8(%rdx),%rax`
`E+i-1`	`int *`	$x_{\mathtt{E}} + 4i - 4$	`leaq -4(%rdx,%rcx,4),%rax`
`*(E+i-3)`	`int`	$M[x_{\mathtt{E}} + 4i - 12]$	`movl -12(%rdx,%rcx,4),%eax`
`&E[i]-E`	`long`	i	`movq %rcx,%rax`

In these examples, we see that operations that return array values have type `int`, and hence involve 4-byte operations (e.g., `movl`) and registers (e.g., `%eax`). Those that return pointers have type `int *`, and hence involve 8-byte operations (e.g., `leaq`) and registers (e.g., `%rax`). The final example shows that one can compute the difference of two pointers within the same data structure, with the result being data having type `long` and value equal to the difference of the two addresses divided by the size of the data type.

Practice Problem 3.37 (solution page 377)

Suppose $x_{\mathtt{P}}$, the address of short integer array `P`, and long integer index i are stored in registers `%rdx` and `%rcx`, respectively. For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in register `%rax` if it is a pointer and register element `%ax` if it has data type `short`.

Expression	Type	Value	Assembly code
`P[1]`			
`P + 3 + i`			
`P[i * 6 - 5]`			
`P[2]`			
`&P[i + 2]`			

3.8.3 Nested Arrays

The general principles of array allocation and referencing hold even when we create arrays of arrays. For example, the declaration

```
int A[5][3];
```

is equivalent to the declaration

```
typedef int row3_t[3];
row3_t A[5];
```

Data type `row3_t` is defined to be an array of three integers. Array `A` contains five such elements, each requiring 12 bytes to store the three integers. The total array size is then $4 \cdot 5 \cdot 3 = 60$ bytes.

Array `A` can also be viewed as a two-dimensional array with five rows and three columns, referenced as `A[0][0]` through `A[4][2]`. The array elements are ordered in memory in *row-major* order, meaning all elements of row 0, which can be written `A[0]`, followed by all elements of row 1 (`A[1]`), and so on. This is illustrated in Figure 3.36.

This ordering is a consequence of our nested declaration. Viewing `A` as an array of five elements, each of which is an array of three `int`'s, we first have `A[0]`, followed by `A[1]`, and so on.

To access elements of multidimensional arrays, the compiler generates code to compute the offset of the desired element and then uses one of the `MOV` instructions with the start of the array as the base address and the (possibly scaled) offset as an index. In general, for an array declared as

T `D[`R`][`C`];`

array element `D[i][j]` is at memory address

$$\&\mathtt{D[i][j]} = x_{\mathtt{D}} + L(C \cdot i + j) \tag{3.1}$$

Figure 3.36
Elements of array in row-major order.

Row	Element	Address
A[0]	A[0][0]	x_A
	A[0][1]	$x_A + 4$
	A[0][2]	$x_A + 8$
A[1]	A[1][0]	$x_A + 12$
	A[1][1]	$x_A + 16$
	A[1][2]	$x_A + 20$
A[2]	A[2][0]	$x_A + 24$
	A[2][1]	$x_A + 28$
	A[2][2]	$x_A + 32$
A[3]	A[3][0]	$x_A + 36$
	A[3][1]	$x_A + 40$
	A[4][2]	$x_A + 44$
A[4]	A[4][0]	$x_A + 48$
	A[4][1]	$x_A + 52$
	A[4][2]	$x_A + 56$

where L is the size of data type T in bytes. As an example, consider the 5×3 integer array `A` defined earlier. Suppose x_A, i, and j are in registers `%rdi`, `%rsi`, and `%rdx`, respectively. Then array element `A[i][j]` can be copied to register `%eax` by the following code:

```
  A in %rdi, i in %rsi, and j in %rdx
  leaq    (%rsi,%rsi,2), %rax     Compute 3i
  leaq    (%rdi,%rax,4), %rax     Compute x_A + 12i
  movl    (%rax,%rdx,4), %eax     Read from M[x_A + 12i + 4]
```

As can be seen, this code computes the element's address as $x_A + 12i + 4j = x_A + 4(3i + j)$ using the scaling and addition capabilities of x86-64 address arithmetic.

Practice Problem 3.38 (solution page 377)

Consider the following source code, where M and N are constants declared with `#define`:

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

In compiling this program, GCC generates the following assembly code:

```
long sum_element(long i, long j)
i in %rdi, j in %rsi
sum_element:
  leaq    0(,%rdi,8), %rdx
  subq    %rdi, %rdx
  addq    %rsi, %rdx
  leaq    (%rsi,%rsi,4), %rax
  addq    %rax, %rdi
  movq    Q(,%rdi,8), %rax
  addq    P(,%rdx,8), %rax
  ret
```

Use your reverse engineering skills to determine the values of M and N based on this assembly code.

3.8.4 Fixed-Size Arrays

The C compiler is able to make many optimizations for code operating on multi-dimensional arrays of fixed size. Here we demonstrate some of the optimizations made by GCC when the optimization level is set with the flag `-O1`. Suppose we declare data type `fix_matrix` to be 16×16 arrays of integers as follows:

```
#define N 16
typedef int fix_matrix[N][N];
```

(This example illustrates a good coding practice. Whenever a program uses some constant as an array dimension or buffer size, it is best to associate a name with it via a `#define` declaration, and then use this name consistently, rather than the numeric value. That way, if an occasion ever arises to change the value, it can be done by simply modifying the `#define` declaration.) The code in Figure 3.37(a) computes element i, k of the product of arrays `A` and `B`—that is, the inner product of row i from `A` and column k from `B`. This product is given by the formula $\sum_{0 \le j < N} a_{i,j} \cdot b_{j,k}$. GCC generates code that we then recoded into C, shown as function `fix_prod_ele_opt` in Figure 3.37(b). This code contains a number of clever optimizations. It removes the integer index `j` and converts all array references to pointer dereferences. This involves (1) generating a pointer, which we have named `Aptr`, that points to successive elements in row i of `A`, (2) generating a pointer, which we have named `Bptr`, that points to successive elements in column k of `B`, and (3) generating a pointer, which we have named `Bend`, that equals the value `Bptr` will have when it is time to terminate the loop. The initial value for `Aptr` is the address of the first element of row i of `A`, given by the C expression `&A[i][0]`. The initial value for `Bptr` is the address of the first element of column k of `B`, given by the C expression `&B[0][k]`. The value for `Bend` is the index of what would be the $(n+1)$st element in column j of `B`, given by the C expression `&B[N][k]`.

(a) Original C code

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele (fix_matrix A, fix_matrix B, long i, long k) {
    long j;
    int result = 0;

    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

(b) Optimized C code

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k) {
    int *Aptr = &A[i][0];    /* Points to elements in row i of A    */
    int *Bptr = &B[0][k];    /* Points to elements in column k of B */
    int *Bend = &B[N][k];    /* Marks stopping point for Bptr       */
    int result = 0;
    do {                         /* No need for initial test */
        result += *Aptr * *Bptr; /* Add next product to sum  */
        Aptr ++;                 /* Move Aptr to next column */
        Bptr += N;               /* Move Bptr to next row    */
    } while (Bptr != Bend);      /* Test for stopping point  */
    return result;
}
```

Figure 3.37 Original and optimized code to compute element i, k of matrix product for fixed-length arrays. The compiler performs these optimizations automatically.

The following is the actual assembly code generated by GCC for function `fix_prod_ele`. We see that four registers are used as follows: `%eax` holds `result`, `%rdi` holds `Aptr`, `%rcx` holds `Bptr`, and `%rsi` holds `Bend`.

```
  int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k)
  A in %rdi, B in %rsi, i in %rdx, k in %rcx
fix_prod_ele:
  salq    $6, %rdx                Compute 64 * i
  addq    %rdx, %rdi              Compute Aptr = x_A + 64i = &A[i][0]
  leaq    (%rsi,%rcx,4), %rcx     Compute Bptr = x_B + 4k = &B[0][k]
  leaq    1024(%rcx), %rsi        Compute Bend = x_B + 4k + 1024 = &B[N][k]
  movl    $0, %eax                Set result = 0
.L7:                            loop:
  movl    (%rdi), %edx            Read *Aptr
  imull   (%rcx), %edx            Multiply by *Bptr
  addl    %edx, %eax              Add to result
```

```
11      addq    $4, %rdi              Increment Aptr ++
12      addq    $64, %rcx             Increment Bptr += N
13      cmpq    %rsi, %rcx            Compare Bptr:Bend
14      jne     .L7                   If !=, goto loop
15      rep; ret                      Return
```

Practice Problem 3.39 (solution page 378)

Use Equation 3.1 to explain how the computations of the initial values for `Aptr`, `Bptr`, and `Bend` in the C code of Figure 3.37(b) (lines 3–5) correctly describe their computations in the assembly code generated for `fix_prod_ele` (lines 3–5).

Practice Problem 3.40 (solution page 378)

The following C code sets the diagonal elements of one of our fixed-size arrays to `val`:

```c
/* Set all diagonal elements to val */
void fix_set_diag(fix_matrix A, int val) {
    long i;
    for (i = 0; i < N; i++)
        A[i][i] = val;
}
```

When compiled with optimization level `-O1`, GCC generates the following assembly code:

```
fix_set_diag:
void fix_set_diag(fix_matrix A, int val)
A in %rdi, val in %rsi
  movl    $0, %eax
.L13:
  movl    %esi, (%rdi,%rax)
  addq    $68, %rax
  cmpq    $1088, %rax
  jne     .L13
  rep; ret
```

Create a C code program `fix_set_diag_opt` that uses optimizations similar to those in the assembly code, in the same style as the code in Figure 3.37(b). Use expressions involving the parameter N rather than integer constants, so that your code will work correctly if N is redefined.

3.8.5 Variable-Size Arrays

Historically, C only supported multidimensional arrays where the sizes (with the possible exception of the first dimension) could be determined at compile time.

Programmers requiring variable-size arrays had to allocate storage for these arrays using functions such as `malloc` or `calloc`, and they had to explicitly encode the mapping of multidimensional arrays into single-dimension ones via row-major indexing, as expressed in Equation 3.1. ISO C99 introduced the capability of having array dimension expressions that are computed as the array is being allocated.

In the C version of variable-size arrays, we can declare an array

```
int A[expr1][expr2]
```

either as a local variable or as an argument to a function, and then the dimensions of the array are determined by evaluating the expressions *expr1* and *expr2* at the time the declaration is encountered. So, for example, we can write a function to access element i, j of an $n \times n$ array as follows:

```
int var_ele(long n, int A[n][n], long i, long j) {
    return A[i][j];
}
```

The parameter `n` must precede the parameter `A[n][n]`, so that the function can compute the array dimensions as the parameter is encountered.

Gcc generates code for this referencing function as

```
    int var_ele(long n, int A[n][n], long i, long j)
    n in %rdi, A in %rsi, i in %rdx, j in %rcx
1   var_ele:
2     imulq   %rdx, %rdi              Compute n · i
3     leaq    (%rsi,%rdi,4), %rax     Compute x_A + 4(n · i
4     movl    (%rax,%rcx,4), %eax     Read from M[x_A + 4(n · i) + 4j]
5     ret
```

As the annotations show, this code computes the address of element i, j as $x_A + 4(n \cdot i) + 4j = x_A + 4(n \cdot i + j)$. The address computation is similar to that of the fixed-size array (Section 3.8.3), except that (1) the register usage changes due to added parameter `n`, and (2) a multiply instruction is used (line 2) to compute $n \cdot i$, rather than an `leaq` instruction to compute $3i$. We see therefore that referencing variable-size arrays requires only a slight generalization over fixed-size ones. The dynamic version must use a multiplication instruction to scale i by n, rather than a series of shifts and adds. In some processors, this multiplication can incur a significant performance penalty, but it is unavoidable in this case.

When variable-size arrays are referenced within a loop, the compiler can often optimize the index computations by exploiting the regularity of the access patterns. For example, Figure 3.38(a) shows C code to compute element i, k of the product of two $n \times n$ arrays `A` and `B`. Gcc generates assembly code, which we have recast into C (Figure 3.38(b)). This code follows a different style from the optimized code for the fixed-size array (Figure 3.37), but that is more an artifact of the choices made by the compiler, rather than a fundamental requirement for the two different functions. The code of Figure 3.38(b) retains loop variable `j`, both to detect when

(a) Original C code

```
/* Compute i,k of variable matrix product */
int var_prod_ele(long n, int A[n][n], int B[n][n], long i, long k) {
    long j;
    int result = 0;

    for (j = 0; j < n; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

(b) Optimized C code

```
/* Compute i,k of variable matrix product */
int var_prod_ele_opt(long n, int A[n][n], int B[n][n], long i, long k) {
    int *Arow = A[i];
    int *Bptr = &B[0][k];
    int result = 0;
    long j;
    for (j = 0; j < n; j++) {
        result += Arow[j] * *Bptr;
        Bptr += n;
    }
    return result;
}
```

Figure 3.38 Original and optimized code to compute element i, k of matrix product for variable-size arrays. The compiler performs these optimizations automatically.

the loop has terminated and to index into an array consisting of the elements of row i of A.

The following is the assembly code for the loop of `var_prod_ele`:

```
Registers: n in %rdi, Arow in %rsi, Bptr in %rcx
           4n in %r9, result in %eax, j in %edx
.L24:                                   loop:
  movl    (%rsi,%rdx,4), %r8d             Read Arow[j]
  imull   (%rcx), %r8d                    Multiply by *Bptr
  addl    %r8d, %eax                      Add to result
  addq    $1, %rdx                        j++
  addq    %r9, %rcx                       Bptr += n
  cmpq    %rdi, %rdx                      Compare j:n
  jne     .L24                            If !=, goto loop
```

We see that the program makes use of both a scaled value $4n$ (register `%r9`) for incrementing `Bptr` as well as the value of n (register `%rdi`) to check the loop bounds. The need for two values does not show up in the C code, due to the scaling of pointer arithmetic.

We have seen that, with optimizations enabled, GCC is able to recognize patterns that arise when a program steps through the elements of a multidimensional array. It can then generate code that avoids the multiplication that would result from a direct application of Equation 3.1. Whether it generates the pointer-based code of Figure 3.37(b) or the array-based code of Figure 3.38(b), these optimizations will significantly improve program performance.

3.9 Heterogeneous Data Structures

C provides two mechanisms for creating data types by combining objects of different types: *structures*, declared using the keyword `struct`, aggregate multiple objects into a single unit; *unions*, declared using the keyword `union`, allow an object to be referenced using several different types.

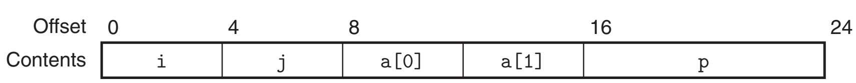
3.9.1 Structures

The C `struct` declaration creates a data type that groups objects of possibly different types into a single object. The different components of a structure are referenced by names. The implementation of structures is similar to that of arrays in that all of the components of a structure are stored in a contiguous region of memory and a pointer to a structure is the address of its first byte. The compiler maintains information about each structure type indicating the byte offset of each field. It generates references to structure elements using these offsets as displacements in memory referencing instructions.

As an example, consider the following structure declaration:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

This structure contains four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte integer pointer, giving a total of 24 bytes:



Observe that array `a` is embedded within the structure. The numbers along the top of the diagram give the byte offsets of the fields from the beginning of the structure.

To access the fields of a structure, the compiler generates code that adds the appropriate offset to the address of the structure. For example, suppose variable `r`