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.

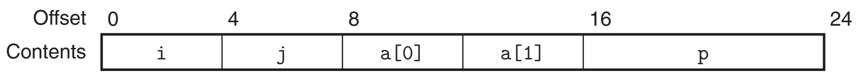
3.9.1 Structures

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`

New to C? Representing an object as a `struct`

The `struct` data type constructor is the closest thing C provides to the objects of C++ and Java. It allows the programmer to keep information about some entity in a single data structure and to reference that information with names.

For example, a graphics program might represent a rectangle as a structure:

```
struct rect {
    long llx;         /* X coordinate of lower-left corner */
    long lly;         /* Y coordinate of lower-left corner */
    unsigned long width;  /* Width (in pixels)                 */
    unsigned long height; /* Height (in pixels)                */
    unsigned color;       /* Coding of color                   */
};
```

We can declare a variable `r` of type `struct rect` and set its field values as follows:

```
    struct rect r;
    r.llx = r.lly = 0;
    r.color = 0xFF00FF;
    r.width = 10;
    r.height = 20;
```

where the expression `r.llx` selects field `llx` of structure `r`.

Alternatively, we can both declare the variable and initialize its fields with a single statement:

```
    struct rect r = { 0, 0, 0xFF00FF, 10, 20 };
```

It is common to pass pointers to structures from one place to another rather than copying them. For example, the following function computes the area of a rectangle, where a pointer to the rectangle `struct` is passed to the function:

```
long area(struct rect *rp) {
    return (*rp).width * (*rp).height;
}
```

The expression `(*rp).width` dereferences the pointer and selects the `width` field of the resulting structure. Parentheses are required, because the compiler would interpret the expression `*rp.width` as `*(rp.width)`, which is not valid. This combination of dereferencing and field selection is so common that C provides an alternative notation using `->`. That is, `rp->width` is equivalent to the expression `(*rp).width`. For example, we can write a function that rotates a rectangle counterclockwise by 90 degrees as

```
void rotate_left(struct rect *rp) {
    /* Exchange width and height */
    long t = rp->height;
    rp->height = rp->width;
    rp->width  = t;
    /* Shift to new lower-left corner */
    rp->llx   -= t;
}
```

New to C? Representing an object as a `struct` *(continued)*

The objects of C++ and Java are more elaborate than structures in C, in that they also associate a set of *methods* with an object that can be invoked to perform computation. In C, we would simply write these as ordinary functions, such as the functions `area` and `rotate_left` shown previously.

of type `struct rec *` is in register `%rdi`. Then the following code copies element `r->i` to element `r->j`:

```
    Registers: r in %rdi
    movl    (%rdi), %eax          Get r->i
    movl    %eax, 4(%rdi)         Store in r->j
```

Since the offset of field `i` is 0, the address of this field is simply the value of `r`. To store into field `j`, the code adds offset 4 to the address of `r`.

To generate a pointer to an object within a structure, we can simply add the field's offset to the structure address. For example, we can generate the pointer `&(r->a[1])` by adding offset $8 + 4 \cdot 1 = 12$. For pointer `r` in register `%rdi` and long integer variable `i` in register `%rsi`, we can generate the pointer value `&(r->a[i])` with the single instruction

```
    Registers: r in %rdi, i %rsi
    leaq    8(%rdi,%rsi,4), %rax    Set %rax to &r->a[i]
```

As a final example, the following code implements the statement

```
r->p = &r->a[r->i + r->j];
```

starting with `r` in register `%rdi`:

```
    Registers: r in %rdi
    movl    4(%rdi), %eax             Get r->j
    addl    (%rdi), %eax              Add r->i
    cltq                              Extend to 8 bytes
    leaq    8(%rdi,%rax,4), %rax      Compute &r->a[r->i + r->j]
    movq    %rax, 16(%rdi)            Store in r->p
```

As these examples show, the selection of the different fields of a structure is handled completely at compile time. The machine code contains no information about the field declarations or the names of the fields.

Practice Problem 3.41 (solution page 379)

Consider the following structure declaration:

```
struct test {
    short *p;
    struct {
        short x;
        short y;
    } s;
    struct test *next;
};
```

This declaration illustrates that one structure can be embedded within another, just as arrays can be embedded within structures and arrays can be embedded within arrays.

The following procedure (with some expressions omitted) operates on this structure:

```
void st_init(struct test *st) {
    st->s.y  = _________;
    st->p    = _________;
    st->next = _________;
}
```

A. What are the offsets (in bytes) of the following fields?

```
   p:   ________
 s.x:   ________
 s.y:   ________
next:   ________
```

B. How many total bytes does the structure require?

C. The compiler generates the following assembly code for `st_init`:

```
    void st_init(struct test *st)
    st in %rdi
st_init:
  movl    8(%rdi), %eax
  movl    %eax, 10(%rdi)
  leaq    10(%rdi), %rax
  movq    %rax, (%rdi)
  movq    %rdi, 12(%rdi)
  ret
```

On the basis of this information, fill in the missing expressions in the code for `st_init`.

Practice Problem 3.42 (solution page 379)

The following code shows the declaration of a structure of type `ACE` and the prototype for a function `test`:

```
struct ACE {
    short      v;
    struct ACE *p;
};

short test(struct ACE *ptr);
```

When the code for `fun` is compiled, GCC generates the following assembly code:

```
     short test(struct ACE *ptr)
     ptr in %rdi
     test:
       movl    $1, %eax
       jmp     .L2
     .L3:
       imulq   (%rdi), %rax
       movq    2(%rdi), %rdi
     .L2:
       testq   %rdi, %rdi
       jne     .L3
       rep; ret
```

A. Use your reverse engineering skills to write C code for `test`.

B. Describe the data structure that this structure implements and the operation performed by `test`.

3.9.2 Unions

Unions provide a way to circumvent the type system of C, allowing a single object to be referenced according to multiple types. The syntax of a union declaration is identical to that for structures, but its semantics are very different. Rather than having the different fields reference different blocks of memory, they all reference the same block.

Consider the following declarations:

```
struct S3 {
    char c;
    int i[2];
    double v;
};
```

```
union U3 {
    char c;
    int i[2];
    double v;
};
```

When compiled on an x86-64 Linux machine, the offsets of the fields, as well as the total size of data types `S3` and `U3`, are as shown in the following table:

Type	c	i	v	Size
S3	0	4	16	24
U3	0	0	0	8

(We will see shortly why `i` has offset 4 in `S3` rather than 1, and why `v` has offset 16, rather than 9 or 12.) For pointer `p` of type `union U3 *`, references `p->c`, `p->i[0]`, and `p->v` would all reference the beginning of the data structure. Observe also that the overall size of a union equals the maximum size of any of its fields.

Unions can be useful in several contexts. However, they can also lead to nasty bugs, since they bypass the safety provided by the C type system. One application is when we know in advance that the use of two different fields in a data structure will be mutually exclusive. Then, declaring these two fields as part of a union rather than a structure will reduce the total space allocated.

For example, suppose we want to implement a binary tree data structure where each leaf node has two `double` data values and each internal node has pointers to two children but no data. If we declare this as

```
struct node_s {
    struct node_s *left;
    struct node_s *right;
    double data[2];
};
```

then every node requires 32 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as

```
union node_u {
    struct {
        union node_u *left;
        union node_u *right;
    } internal;
    double data[2];
};
```

then every node will require just 16 bytes. If `n` is a pointer to a node of type `union node_u *`, we would reference the data of a leaf node as `n->data[0]` and `n->data[1]`, and the children of an internal node as `n->internal.left` and `n->internal.right`.

With this encoding, however, there is no way to determine whether a given node is a leaf or an internal node. A common method is to introduce an enumerated type defining the different possible choices for the union, and then create a structure containing a tag field and the union:

```
typedef enum { N_LEAF, N_INTERNAL } nodetype_t;

struct node_t {
    nodetype_t type;
    union {
        struct {
            struct node_t *left;
            struct node_t *right;
        } internal;
        double data[2];
    } info;
};
```

This structure requires a total of 24 bytes: 4 for `type`, and either 8 each for `info.internal.left` and `info.internal.right` or 16 for `info.data`. As we will discuss shortly, an additional 4 bytes of padding is required between the field for `type` and the union elements, bringing the total structure size to $4 + 4 + 16 = 24$. In this case, the savings gain of using a union is small relative to the awkwardness of the resulting code. For data structures with more fields, the savings can be more compelling.

Unions can also be used to access the bit patterns of different data types. For example, suppose we use a simple cast to convert a value `d` of type `double` to a value `u` of type `unsigned long`:

```
    unsigned long u = (unsigned long) d;
```

Value `u` will be an integer representation of `d`. Except for the case where `d` is 0.0, the bit representation of `u` will be very different from that of `d`. Now consider the following code to generate a value of type `unsigned long` from a `double`:

```
unsigned long double2bits(double d) {
    union {
        double d;
        unsigned long u;
    } temp;
    temp.d = d;
    return temp.u;
};
```

In this code, we store the argument in the union using one data type and access it using another. The result will be that `u` will have the same bit representation as `d`, including fields for the sign bit, the exponent, and the significand, as described in

Section 3.11. The numeric value of `u` will bear no relation to that of `d`, except for the case when `d` is 0.0.

When using unions to combine data types of different sizes, byte-ordering issues can become important. For example, suppose we write a procedure that will create an 8-byte `double` using the bit patterns given by two 4-byte `unsigned` values:

```
double uu2double(unsigned word0, unsigned word1)
{
    union {
        double d;
        unsigned u[2];
    } temp;

    temp.u[0] = word0;
    temp.u[1] = word1;
    return temp.d;
}
```

On a little-endian machine, such as an x86-64 processor, argument `word0` will become the low-order 4 bytes of `d`, while `word1` will become the high-order 4 bytes. On a big-endian machine, the role of the two arguments will be reversed.

Practice Problem 3.43 (solution page 380)

Suppose you are given the job of checking that a C compiler generates the proper code for structure and union access. You write the following structure declaration:

```
typedef union {
    struct {
        long   u;
        short  v;
        char   w;
    } t1;
    struct {
        int a[2];
        char  *p;
    } t2;
} u_type;
```

You write a series of functions of the form

```
void get(u_type *up, type *dest) {
    *dest =  expr;
}
```

with different access expressions *expr* and with destination data type *type* set according to type associated with *expr*. You then examine the code generated when compiling the functions to see if they match your expectations.

Suppose in these functions that up and dest are loaded into registers %rdi and %rsi, respectively. Fill in the following table with data type *type* and sequences of one to three instructions to compute the expression and store the result at dest.

expr	*type*	Code
up->t1.u	long	movq (%rdi), %rax movq %rax, (%rsi)
up->t1.v	______	______ ______ ______
&up->t1.w	______	______ ______ ______
up->t2.a	______	______ ______ ______
up->t2.a[up->t1.u]	______	______ ______ ______
*up->t2.p	______	______ ______ ______

3.9.3 Data Alignment

Many computer systems place restrictions on the allowable addresses for the primitive data types, requiring that the address for some objects must be a multiple of some value K (typically 2, 4, or 8). Such *alignment restrictions* simplify the design of the hardware forming the interface between the processor and the memory system. For example, suppose a processor always fetches 8 bytes from memory with an address that must be a multiple of 8. If we can guarantee that any double will be aligned to have its address be a multiple of 8, then the value can be read or written with a single memory operation. Otherwise, we may need to perform two memory accesses, since the object might be split across two 8-byte memory blocks.

The x86-64 hardware will work correctly regardless of the alignment of data. However, Intel recommends that data be aligned to improve memory system performance. Their alignment rule is based on the principle that any primitive object of K bytes must have an address that is a multiple of K. We can see that this rule leads to the following alignments:

K	Types
1	char
2	short
4	int, float
8	long, double, char *

Alignment is enforced by making sure that every data type is organized and allocated in such a way that every object within the type satisfies its alignment restrictions. The compiler places directives in the assembly code indicating the desired alignment for global data. For example, the assembly-code declaration of the jump table on page 271 contains the following directive on line 2:

```
.align 8
```

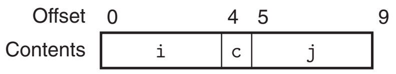
This ensures that the data following it (in this case the start of the jump table) will start with an address that is a multiple of 8. Since each table entry is 8 bytes long, the successive elements will obey the 8-byte alignment restriction.

For code involving structures, the compiler may need to insert gaps in the field allocation to ensure that each structure element satisfies its alignment requirement. The structure will then have some required alignment for its starting address.

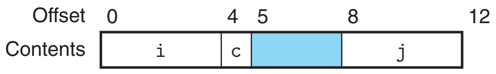
For example, consider the structure declaration

```
struct S1 {
    int  i;
    char c;
    int  j;
};
```

Suppose the compiler used the minimal 9-byte allocation, diagrammed as follows:



Then it would be impossible to satisfy the 4-byte alignment requirement for both fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap (shown here as shaded in blue) between fields `c` and `j`:



As a result, `j` has offset 8, and the overall structure size is 12 bytes. Furthermore, the compiler must ensure that any pointer `p` of type `struct S1*` satisfies a 4-byte alignment. Using our earlier notation, let pointer `p` have value x_p. Then x_p must be a multiple of 4. This guarantees that both `p->i` (address x_p) and `p->j` (address $x_p + 8$) will satisfy their 4-byte alignment requirements.

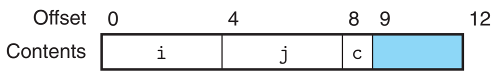
In addition, the compiler may need to add padding to the end of the structure so that each element in an array of structures will satisfy its alignment requirement. For example, consider the following structure declaration:

```
struct S2 {
    int  i;
    int  j;
    char c;
};
```

If we pack this structure into 9 bytes, we can still satisfy the alignment requirements for fields `i` and `j` by making sure that the starting address of the structure satisfies a 4-byte alignment requirement. Consider, however, the following declaration:

```
struct S2 d[4];
```

With the 9-byte allocation, it is not possible to satisfy the alignment requirement for each element of `d`, because these elements will have addresses $x_{\mathtt{d}}$, $x_{\mathtt{d}}+9$, $x_{\mathtt{d}}+18$, and $x_{\mathtt{d}}+27$. Instead, the compiler allocates 12 bytes for structure `S2`, with the final 3 bytes being wasted space:



That way, the elements of `d` will have addresses $x_{\mathtt{d}}$, $x_{\mathtt{d}}+12$, $x_{\mathtt{d}}+24$, and $x_{\mathtt{d}}+36$. As long as $x_{\mathtt{d}}$ is a multiple of 4, all of the alignment restrictions will be satisfied.

Practice Problem 3.44 (solution page 381)

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement for x86-64:

A. `struct P1 { short i; int c; int *j; short *d; };`

B. `struct P2 { int i[2]; char c[8]; short s[4]; long *j; };`

C. `struct P3 { long w[2]; int *c[2] };`

D. `struct P4 { char w[16]; char *c[2] };`

E. `struct P5 { struct P4 a[2]; struct P1 t };`

Practice Problem 3.45 (solution page 381)

Answer the following for the structure declaration

```
struct {
  int    *a;
  float  b;
  char   c;
  short  d;
  long   e;
  double f;
```

Aside A case of mandatory alignment

For most x86-64 instructions, keeping data aligned improves efficiency, but it does not affect program behavior. On the other hand, some models of Intel and AMD processors will not work correctly with unaligned data for some of the SSE instructions implementing multimedia operations. These instructions operate on 16-byte blocks of data, and the instructions that transfer data between the SSE unit and memory require the memory addresses to be multiples of 16. Any attempt to access memory with an address that does not satisfy this alignment will lead to an *exception* (see Section 8.1), with the default behavior for the program to terminate.

As a result, any compiler and run-time system for an x86-64 processor must ensure that any memory allocated to hold a data structure that may be read from or stored into an SSE register must satisfy a 16-byte alignment. This requirement has the following two consequences:

- The starting address for any block generated by a memory allocation function (`alloca`, `malloc`, `calloc`, or `realloc`) must be a multiple of 16.
- The stack frame for most functions must be aligned on a 16-byte boundary. (This requirement has a number of exceptions.)

More recent versions of x86-64 processors implement the AVX multimedia instructions. In addition to providing a superset of the SSE instructions, processors supporting AVX also do not have a mandatory alignment requirement.

```
    int     g;
    char    *h;
} rec;
```

A. What are the byte offsets of all the fields in the structure?

B. What is the total size of the structure?

C. Rearrange the fields of the structure to minimize wasted space, and then show the byte offsets and total size for the rearranged structure.

3.10 Combining Control and Data in Machine-Level Programs

So far, we have looked separately at how machine-level code implements the control aspects of a program and how it implements different data structures. In this section, we look at ways in which data and control interact with each other. We start by taking a deep look into pointers, one of the most important concepts in the C programming language, but one for which many programmers only have a shallow understanding. We review the use of the symbolic debugger GDB for examining the detailed operation of machine-level programs. Next, we see how understanding machine-level programs enables us to study buffer overflow, an important security vulnerability in many real-world systems. Finally, we examine