```
readdir(dir)
char *dir;
{
    static struct direct dentry;
    register int j;
    register struct lbuf *ep;

    if ((dirf = fopen(dir, "r")) == NULL) {
        printf("%s unreadable\n", dir);
        return;
    }
    for(;;) {
        if (fread((char *)&dentry, sizeof(dentry), 1, dirf) != 1)
            break;
        if (dentry.d_ino==0
         || aflg==0 && dentry.d_name[0]=='.' &&  (dentry.d_name[1]=='\0'
            || dentry.d_name[1]=='.' && dentry.d_name[2]=='\0'))
            continue;
        ep = gstat(makename(dir, dentry.d_name), 0);
        [...]
        for (j=0; j<DIRSIZ; j++)
            ep->ln.lname[j] = dentry.d_name[j];
    }
    fclose(dirf);
}
```


Figure 7.35 Directly interpreting a directory's data in the Seventh Edition Unix

cause programs that directly interpreted directory data to display wrong results or fail in mysterious ways.

Exercise 7.55 What are the disadvantages of data abstraction in terms of efficiency and readability? Provide concrete examples. How can these problems be avoided?

7.4.3 Type Checking

If data abstraction is a policy promoting (among other things) a system's stability and maintainability, type-checking is the enforcement mechanism. An implementation that takes advantage of a language's type-checking features will catch erroneous modifications at compile time; an implementation based around loose types or one that circumvents the language's type system can result in difficult-to-locate runtime errors.

Designs, APIS, or implementations that fail to take advantage of a language's type system involve two symmetrical operations: one whereby information about an element's type is lost as this element is upcast into a more generic type (commonly `void *` or `Object`) and one whereby the assumed type information is plucked out of thin air when a generic type is downcast into a more specific type. The first operation is generally harmless, but the second one assumes that the element being downcast is indeed of the corresponding type; if this is not true, depending on the language, we may end up with a runtime error or a difficult-to-trace bug. You can see a concrete

```
public class ArgoEventPump {
    [...]
    private ArrayList _listeners = null;
    [...]
    protected void doAddListener(int event, ArgoEventListener listener) {
        if (_listeners == null) _listeners = new ArrayList();
    _listeners.add(new Pair(event, listener));
    }
    [...]
    protected void doFireEvent(ArgoEvent event) {
    [...]
        ListIterator iterator = _listeners.listIterator();
        while (iterator.hasNext()) {
         Pair pair = (Pair)iterator.next();
         [...]
        }

    }
}
```

A container of typeless (Object) elements

1 A Pair element is added to the Object container (an upcast from Pair to Object)

2 An (assumed to be Pair) element is extracted from the Object container (a downcast from Object to Pair)

Figure 7.36 Playing loose with types in pre–Java 1.5 code

example in Figure 7.36.[185] This (pre–Java 1.5) code uses an `ArrayList` as a container for storing `Pair` elements as plain Java `Objects`. When a `Pair` element is added to `_listeners` (Figure 7.36:1), it is implicitly cast into an `Object` for the purposes of compile-time type checks. Then, when an element is retrieved (Figure 7.36:2), the code *assumes* that the `Object` is indeed a `Pair` and downcasts it into that type. The compiler cannot verify this assumption; if we changed the code in the `doAddListenet` method to add a different element type into the container, we would find the problem only at runtime, as a `ClassCastException` when `doFireEvent` got executed. Worse, the runtime error would manifest itself only if our test coverage included both `doAddListenet` and `doFireEvent`, in that order. In Java 1.5, we can avoid this (quite common) coding style by using the generics language extension.

Legacy APIs often force us to abandon strict type checking. Two prime culprits in this category are the pre-Win32 swaths of the Windows platform API and the Unix `ioctl` interface. Both interfaces use "integer" arguments that can variously hold many other different and incompatible types. The type information is communicated through an out-of-band mechanism that the compiler can neither check nor enforce. Have a look at the following (fairly typical) Windows code, implementing a *callback function*: a user-level function that the Windows system calls when a specific event class occurs:[186]

[185] argouml/org/argouml/application/events/ArgoEventPump.java:29–34, 58–61, 140–175
[186] apache/src/os/win32/Win9xConHook.c:413–461

```
static LRESULT CALLBACK
ttyConsoleCtrlWndProc(HWND hwnd, UINT msg, WPARAM wParam,
                      LPARAM lParam)
{
    if (msg == WM_CREATE) {
        tty_info *tty =
         (tty_info*) (((LPCREATESTRUCT)lParam)->lpCreateParams);
        [...]
    } else if ((msg == WM_QUERYENDSESSION) ||
               (msg == WM_ENDSESSION)) {
        if (lParam & ENDSESSION_LOGOFF)
```

In that code, the same `lParam` argument is used as a pointer to a `CREATESTRUCT` if the `msg` argument has a value of `WM_CREATE` and as a bitfield if the `msg` argument has a value of `WM_QUERYENDSESSION` or `WM_ENDSESSION`.

The Unix `ioctl` and `fcntl` system calls suffer from a similar problem. The type of their third (and following) arguments depends on the value passed as the second argument. For example, in the following code extracts from the Unix *mt* magnetic tape control command,[187] an `ioctl` operation is used to

- Perform a tape operation (`MTIOCTOP`), passing as the third argument a `struct mtop` pointer
- Get the tape's status (`MTIOCGET`), passing as the third argument a `struct mt_status` pointer
- Get the tape's logical or hardware block address (`MTIOCRDSPOS`, `MTIOCRDHPOS`), passing as the third argument a pointer to an integer

```
int
main(int argc, char *argv[])
{
    struct mtget mt_status;
    struct mtop mt_com;
    int ch, len, mtfd, flags;
    int count;

    switch (comp->c_spcl) {
    case MTIOCTOP:
        if (ioctl(mtfd, MTIOCTOP, &mt_com) < 0)
            err(2, "%s", tape);
        break;
```

[187] netbsdsrc/bin/mt/mt.c:111–211

```
case MTIOCGET:
    if (ioctl(mtfd, MTIOCGET, &mt_status) < 0)
        err(2, "%s: %s", tape, comp->c_name);
    break;
case MTIOCRDSPOS:
case MTIOCRDHPOS:
    if (ioctl(mtfd, comp->c_spcl, (caddr_t) &count) < 0)
        err(2, "%s", tape);
    break;
```

The type of the third `ioctl` argument is not checked at compile time. Therefore, small changes to the `ioctl` interface are unthinkable; the affected programs would still compile without problems but fail in mysterious ways when executed. Although this situation of a difficult-to-change API may appear as stable (exactly what we are looking for in this section), the stability we have is that of a house of cards: We dare not make any changes to it lest it collapse.

Exercise 7.56 Comment on the type-safety of the C `printf` function. Some compilers can check the types of the data elements, based on the string passed as the format specification. Is this approach watertight? Does it overcome the problem?

7.4.4 Compile-Time Assertions

There are cases in which implementation choices cannot be abstracted in a way that will cleanly solve the problem at hand in an acceptable fashion. The underlying reasons can be traced back to efficiency concerns or language limitations. In such cases, the C/C++ language preprocessor allows us to use compile-time assertions to verify that the implementation assumptions we made remain valid in the face of maintenance changes. These compile-time assertions ensure that the software is always built within the context of the operational envelope it was designed for.

As an example of a compile-time assertion used to verify the compilation environment, the following code forms a table for converting ASCII characters into lowercase. This task can be efficiently implemented through a simple lookup table mapping character codes to their lowercase values. For historical reasons, the conversion should also be able to handle the `EOF` value. As this value is typically -1, it can be conveniently put at the table's first element, with the lookup function adjusted to add 1 to the value being examined.[188]

[188] netbsdsrc/lib/libc/gen/tolower_.c