### Conversions to real floating-point types

Not all integer values can be exactly represented in floating-point types. For example, although the value range of the type `float` includes the range of the types `long` and `long long`, `float` is precise to only six decimal digits. Thus, some `long` values cannot be stored exactly in a `float` object. The result of such a conversion is the next lower or next higher representable value, as the following example illustrates:

```
long  l_var = 123456789L;
float f_var = l_var;          // Implicitly converts long value
                              // to float.

printf("The rounding error (f_var - l_var) is %f\n",
                                          (double)f_var  - l_var);
```

Note that the subtraction in this example is performed with at least `double` precision. Typical output produced by this code is:

```
The rounding error (f_var - l_var;) is 3.000000
```

Any value in a floating-point type can be represented exactly in another floating-point type of greater precision. Thus, when a `double` value is converted to `long double`, or when a `float` value is converted to `double` or `long double`, the value is exactly preserved. In conversions from a more precise to a less precise type, however, the value being converted may be beyond the range of the new type. If the value exceeds the target type's range, the result of the conversion is undefined. If the value is within the target type's range, but not exactly representable in the target type's precision, then the result is the next smaller or next greater representable value. The program in Example 2-2 illustrates the rounding error produced by such a conversion to a less-precise floating-point type.

When a complex number is converted to a real floating-point type, the imaginary part is simply discarded, and the result is the complex number's real part, which may have to be further converted to the target type as described in this section.

### Conversions to complex floating-point types

When an integer or a real floating-point number is converted to a complex type, the real part of the result is obtained by converting the value to the corresponding real floating-point type as described in the previous section. The imaginary part is zero.

When a complex number is converted to a different complex type, the real and imaginary parts are converted separately according to the rules for real floating-point types:

```
#include <complex.h>          // Defines macros such as the imaginary
                              // constant I
double _Complex dz = 2;
float _Complex fz = dz + I;
```

In the first of these two initializations, the integer constant 2 is implicitly converted to `double _Complex` for assignment to `dz`. The resulting value of `dz` is 2.0 + 0.0 × `I`.

In the initialization of `fz`, the two parts of the `double _Complex` value of `dz` are converted (after the addition) to `float`, so that the real part of `fz` is equal to `2.0F`, and the imaginary part `1.0F`.

## Conversion of Nonarithmetic Types

Pointers and the names of arrays and functions are also subject to certain implicit and explicit type conversions. Structures and unions cannot be converted, although pointers to them can be converted to and from other pointer types.

### Array and Function Designators

An array or function designator is any expression that has an array or function type. In most cases, the compiler implicitly converts an expression with an array type, such as the name of an array, into a pointer to the array's first element. The array expression is *not* converted into a pointer only in the following cases:

- When the array is the operand of the `sizeof` operator
- When the array is the operand of the address operator `&`
- When a string literal is used to initialize an array of `char`, `wchar_t`, `char16_t`, or `char32_t`

The following examples demonstrate the implicit conversion of array designators into pointers, using the conversion specification `%p` to print pointer values:

```
#include <stdio.h>

int *iPtr = 0;                  // A pointer to int, initialized with 0.
int iArray[] = { 0, 10, 20 };   // An array of int, initialized.

int array_length = sizeof(iArray) / sizeof(int); // The number of
                                                 // elements:
                                                 // in this case, 3.

printf("The array starts at the address %p.\n", iArray);

*iArray = 5;                      // Equivalent to iArray[0] = 5;

iPtr = iArray + array_length - 1; // Point to the last element of
                                  // iArray: equivalent to
                                  // iPtr = &iArray[array_length-1];

printf("The last element of the array is %d.\n", *iPtr);
```

In the initialization of `array_length` in this example, the expression `sizeof(iArray)` yields the size of the whole array, not the size of a pointer. However, the same identifier `iArray` is implicitly converted to a pointer in the other three statements in which it appears:

- As an argument in the first `printf()` call
- As the operand of the dereferencing operator `*`
- In the pointer arithmetic operations and assignment to `iPtr` (see also "Modifying and Comparing Pointers" on page 147)

The names of character arrays are used as pointers in string operations, as in this example:

```
#include <stdio.h>
#include <string.h>          // Declares size_t strlen( const char *s )

char msg[80] = "I'm a string literal."; // Initialize an array of char.
printf("The string is %d characters long.\n", strlen(msg));
                                        // Answer: 21.
printf("The array named msg is %d bytes long.\n", sizeof(msg));
                                        // Answer: 80.
```

In the function call `strlen(msg)` in this example, the array identifier `msg` is implicitly converted to a pointer to the array's first element with the function parameter's type, `const char *`. Internally, `strlen()` merely counts the characters beginning at that address until the first null character, the string terminator.

Similarly, any expression that designates a function, such as a function name, can also be implicitly converted into a pointer to the function. Again, this conversion does not apply when the expression is the operand of the address operator `&`. The `sizeof` operator cannot be used with an operand of function type. The following example illustrates the implicit conversion of function names to pointers (the program initializes an array of pointers to functions, then calls the functions in a loop):

```
#include <stdio.h>
void func0() { puts("This is the function func0(). "); }
void func1() { puts("This is the function func1(). "); }
/* ... */
void (*funcTable[2])(void) = { func0, func1 }; // Array of two pointers
                                               // to functions
                                               // returning void.
for ( int i = 0; i < 2; ++i )   // Use the loop counter as the array
   funcTable[i]();              // index.
```

## Explicit Pointer Conversions

To convert a pointer from one pointer type to another, you must usually use an explicit cast. In some cases, the compiler provides an implicit conversion, as

described in "Implicit Pointer Conversions" on page 61. Pointers can also be explicitly converted into integers, and vice versa.

### Object pointers

You can explicitly convert an object pointer—that is, a pointer to a complete or incomplete object type—to any other object pointer type. In your program, you must ensure that your use of the converted pointer makes sense. Here is an example:

```
float  f_var = 1.5F;
long *l_ptr = (long *)&f_var;     // Initialize a pointer to long with
                                  // the address of f_var.
double *d_ptr = (double *)l_ptr;  // Initialize a pointer to double
                                  // with the same address.

// On a system where sizeof(float) equals sizeof(long):

printf( "The %zu bytes that represent %f, in hexadecimal: 0x%lX\n",
        sizeof(f_var), f_var, *l_ptr );

// Using a converted pointer in an assignment can cause trouble:

/*  *d_ptr = 2.5;  */   // Don't try this! f_var's location doesn't
                        // have space for a double value!
*(float *)d_ptr = 2.5;  // OK: stores a float value in that location.
```

If the object pointer after conversion does not have the alignment required by the new type, the results of using the pointer are undefined. In all other cases, converting the pointer value back into the original pointer type is guaranteed to yield an equivalent to the original pointer.

If you convert any type of object pointer into a pointer to any `char` type (`char`, `signed char`, or `unsigned char`), the result is a pointer to the first byte of the object. The first byte is considered here to be the byte with the lowest address, regardless of the system's byte order structure. The following example uses this feature to print a hexadecimal dump of a structure variable:

```
#include <stdio.h>
struct Data {
              short id;
              double val;
            };

struct Data myData = { 0x123, 77.7 };          // Initialize a
                                               // structure.

unsigned char *cp = (unsigned char *)&myData;  // Pointer to the
                                               // first byte of
                                               // the structure.

printf( "%p: ", cp );                          // Print the starting
                                               // address.
```

```
for ( int i = 0; i < sizeof(myData); ++i )      // Print each byte
  printf( "%02X ", *(cp + i) );                 // of the structure,
putchar( '\n' );                                // in hexadecimal.
```

This example produces output like the following:

```
0xbffffd70: 23 01 00 00 00 00 00 00 CD CC CC CC CC 6C 53 40
```

The output of the first two bytes, `23 01`, shows that the code was executed on a little-endian system: the byte with the lowest address in the structure `myData` was the least significant byte of the `short` member `id`.

### Function pointers

The type of a function always includes its return type, and may also include its parameter types. You can explicitly convert a pointer to a given function into a pointer to a function of a different type. In the following example, the `typedef` statement defines a name for the type "function that has one `double` parameter and returns a `double` value":

```
#include <math.h>                   // Declares sqrt() and pow().
typedef double (func_t)(double);    // Define a type named func_t.

func_t *pFunc = sqrt;               // A pointer to func_t, initialized
                                    // with the address of sqrt().

double y = pFunc( 2.0 );          // A correct function call by pointer.
printf( "The square root of 2 is %f.\n", y );

pFunc = (func_t *)pow;              // Change the pointer's value to
                                    // the address of pow().
/*  y = pFunc( 2.0 );  */          // Don't try this: pow() takes two
                                    // arguments.
```

In this example, the function pointer `pFunc` is assigned the addresses of functions that have different types. However, if the program uses the pointer to call a function with a definition that does not match the exact function pointer type, the program's behavior is undefined.

## Implicit Pointer Conversions

The compiler converts certain types of pointers implicitly. Assignments, conditional expressions using the equality operators `==` and `!=`, and function calls involve implicit pointer conversion in three kinds of cases, which are described individually in the sections that follow. The three kinds of implicit pointer conversion are:

- Any object pointer type can be implicitly converted to a pointer to `void`, and vice versa.

- Any pointer to a given type can be implicitly converted into a pointer to a more qualified version of that type—that is, a type with one or more additional type qualifiers.
- A *null pointer constant* can be implicitly converted into any pointer type.

### Pointers to void

Pointers to `void`—that is, pointers of the type `void *`—are used as "multipurpose" pointers to represent the address of any object, without regard for its type. For example, the `malloc()` function returns a pointer to `void` (see Example 2-3). Before you can access the memory block, the `void` pointer must always be converted into a pointer to an object.

Example 4-1 demonstrates more uses of pointers to `void`. The program sorts an array using the standard function `qsort()`, which is declared in the header file *stdlib.h* with the following prototype:

```
void qsort( void *array, size_t n, size_t element_size,
            int (*compare)(const void *, const void *) );
```

The `qsort()` function sorts the array in ascending order, beginning at the address *`array`*, using the quick-sort algorithm. The array is assumed to have *`n`* elements whose size is *`element_size`*.

The fourth parameter, *`compare`*, is a pointer to a function that `qsort()` calls to compare any two array elements. The addresses of the two elements to be compared are passed to this function in its pointer parameters. Usually this comparison function must be defined by the programmer. It must return a value that is less than, equal to, or greater than 0 to indicate whether the first element is less than, equal to, or greater than the second.

*Example 4-1. A comparison function for qsort()*

```
#include <stdlib.h>
#define ARR_LEN 20

/*
 * A function to compare any two float elements,
 * for use as a call-back function by qsort().
 * Arguments are passed by pointer.
 *
 * Returns: -1 if the first is less than the second;
 *           0 if the elements are equal;
 *           1 if the first is greater than the second.
 */
int  floatcmp( const void* p1, const void* p2 )
{
  float x = *(float *)p1,
        y = *(float *)p2;
```

```
  return (x < y) ? -1 : ((x == y) ? 0 : 1);
}

/*
 * The main() function sorts an array of float.
 */
int main()
{
  /* Allocate space for the array dynamically:  */
  float *pNumbers = malloc( ARR_LEN * sizeof(float) );

  /* ... Handle errors, initialize array elements ... */

  /* Sort the array: */
  qsort( pNumbers, ARR_LEN, sizeof(float), floatcmp );

  /* ... Work with the sorted array ... */

   return 0;
}
```

In Example 4-1, the `malloc()` function returns a `void *`, which is implicitly converted to `float *` in the assignment to `pNumbers`. In the call to `qsort()`, the first argument `pNumbers` is implicitly converted from `float *` to `void *`, and the function name `floatcmp` is implicitly interpreted as a function pointer. Finally, when the `floatcmp()` function is called by `qsort()`, it receives arguments of the type `void *`, the "universal" pointer type, and must convert them explicitly to `float *` before dereferencing them to initialize its `float` variables.



### Pointers to qualified object types

The type qualifiers in C are `const`, `volatile`, and `restrict` (see Chapter 11 for details on these qualifiers). For example, the compiler implicitly converts any pointer to `int` into a pointer to `const int` where necessary. If you want to remove a qualification rather than adding one, however, you must use an explicit type conversion, as the following example illustrates:

```
int n = 77;
const int *ciPtr = 0;   // A pointer to const int.
                        // The pointer itself is not constant!

ciPtr = &n;          // Implicitly converts the address to the type
                     // const int *.

n = *ciPtr + 3;      // OK: this has the same effect as n = n + 3;

*ciPtr *= 2;         // Error: you can't change an object referenced by
                     // a pointer to const int.
```

```
*(int *)ciPtr *= 2;  // OK: Explicitly converts the pointer into a
                     // pointer to a nonconstant int.
```

The second to last statement in this example illustrates why pointers to `const`-qualified types are sometimes called *read-only pointers*: although you can modify the pointers' values, you can't use them to modify objects they point to.

### Null pointer constants

A null pointer constant is an integer constant with the value 0, or a constant integer value of 0 cast as a pointer to `void`. The macro `NULL` is defined in the header files *stdlib.h*, *stdio.h*, and others as a null pointer constant. The following example illustrates the use of the macro `NULL` as a pointer constant to initialize pointers rather than an integer zero or a null character:

```
#include <stdlib.h>
long *lPtr = NULL;      // Initialize to NULL: pointer is not ready
                        // for use.

/* ... operations here may assign lPtr an object address ... */

if ( lPtr != NULL )
{
  /* ... use lPtr only if it has been changed from NULL ... */
}
```

When you convert a null pointer constant to another pointer type, the result is called a *null pointer*. The bit pattern of a null pointer is not necessarily zero. However, when you compare a null pointer to zero, to `NULL`, or to another null pointer, the result is always `true`. Conversely, comparing a null pointer to any valid pointer to an object or function always yields `false`.

## Conversions Between Pointer and Integer Types

You can explicitly convert a pointer to an integer type, and vice versa. The result of such conversions depends on the compiler, and should be consistent with the addressing structure of the system on which the compiled executable runs. Conversions between pointer and integer types can be useful in system programming, and necessary when programs need to access specific physical addresses, such as ROM or memory-mapped I/O registers.

When you convert a pointer to an integer type whose range is not large enough to represent the pointer's value, the result is undefined. Conversely, converting an integer into a pointer type does not necessarily yield a valid pointer. The header file *stdint.h* may optionally define the integer types `intptr_t` (signed) and `uintptr_t` (unsigned). Any valid pointer can be converted to either of these types, and a subsequent conversion back into a pointer is guaranteed to yield the original pointer. You should therefore use one of these types, if *stdint.h* defines them, any time you need to perform conversions between pointers and integers.

Here are a few examples:

```
float x = 1.5F, *fPtr = &x;       // A float, and a pointer to it.

// Save the pointer's value as an integer:
unsigned long long adr_val = (unsigned long long)fPtr;

// Or, if stdint.h has been included and uintptr_t is defined:
uintptr_t adr_val = (uintptr_t)fPtr;

/*
 * On an Intel x86 PC in DOS, the BIOS data block begins at the
 * address 0x0040:0000. The first two-byte word at that address
 * contains the I/O address of the serial port COM1.
 * (Compile using DOS's "large" memory model.)
 */
unsigned short *biosPtr = (unsigned short *)0x400000L;
unsigned short com1_io = *biosPtr;  // The first word contains the
                                    // I/O address of COM1.
printf( "COM1 has the I/O base address %Xh.\n", com1_io );
```

The last three statements obtain information about the hardware configuration from the system data table, assuming the operating environment allows the program to access that memory area. In a DOS program compiled with the large memory model, pointers are 32 bits wide and consist of a segment address in the higher 16 bits and an offset in the lower 16 bits (often written in the form *segment*:*offset*). Thus, the pointer `biosPtr` in the prior example can be initialized with a long integer constant.