7

Functions

To iterate is human,
to recurse divine.
– L. Peter Deutsch

Function declarations and definitions — argument passing — return values — function overloading — ambiguity resolution — default arguments — ***stdargs*** — pointers to functions — macros — advice — exercises.

7.1 Function Declarations [fct.dcl]

The typical way of getting something done in a C++ program is to call a function to do it. Defining a function is the way you specify how an operation is to be done. A function cannot be called unless it has been previously declared.

A function declaration gives the name of the function, the type of the value returned (if any) by the function, and the number and types of the arguments that must be supplied in a call of the function. For example:

```
Elem* next_elem();
char* strcpy(char* to, const char* from);
void exit(int);
```

The semantics of argument passing are identical to the semantics of initialization. Argument types are checked and implicit argument type conversion takes place when necessary. For example:

```
double sqrt(double);

double sr2 = sqrt(2);          // call sqrt() with the argument double(2)
double sq3 = sqrt("three");    // error: sqrt() requires an argument of type double
```

The value of such checking and type conversion should not be underestimated.

A function declaration may contain argument names. This can be a help to the reader of a program, but the compiler simply ignores such names. As mentioned in §4.7, ***void*** as a return type means that the function does not return a value.

7.1.1 Function Definitions [fct.def]

Every function that is called in a program must be defined somewhere (once only). A function definition is a function declaration in which the body of the function is presented. For example:

```
extern void swap(int*, int*);  // a declaration

void swap(int* p, int* q)      // a definition
{
      int t = *p;
      *p = *q;
      *q = t;
}
```

The type of the definition and all declarations for a function must specify the same type. The argument names, however, are not part of the type and need not be identical.

It is not uncommon to have function definitions with unused arguments:

```
void search(table* t, const char* key, const char*)
{
      // no use of the third argument
}
```

As shown, the fact that an argument is unused can be indicated by not naming it. Typically, unnamed arguments arise from the simplification of code or from planning ahead for extensions. In both cases, leaving the argument in place, although unused, ensures that callers are not affected by the change.

A function can be defined to be ***inline***. For example:

```
inline int fac(int n)
{
      return (n<2) ? 1 : n*fac(n-1);
}
```

The ***inline*** specifier is a hint to the compiler that it should attempt to generate code for a call of ***fac()*** inline rather than laying down the code for the function once and then calling through the usual function call mechanism. A clever compiler can generate the constant ***720*** for a call ***fac(6)***. The possibility of mutually recursive inline functions, inline functions that recurse or not depending on input, etc., makes it impossible to guarantee that every call of an ***inline*** function is actually inlined. The degree of cleverness of a compiler cannot be legislated, so one compiler might generate ***720***, another ***6*fac(5)***, and yet another an un-inlined call ***fac(6)***.

To make inlining possible in the absence of unusually clever compilation and linking facilities, the definition – and not just the declaration – of an inline function must be in scope (§9.2). An ***inline*** specifier does not affect the semantics of a function. In particular, an inline function still has a unique address and so has ***static*** variables (§7.1.2) of an inline function.

7.1.2 Static Variables [fct.static]

A local variable is initialized when the thread of execution reaches its definition. By default, this happens in every call of the function and each invocation of the function has its own copy of the variable. If a local variable is declared ***static***, a single, statically allocated object will be used to represent that variable in all calls of the function. It will be initialized only the first time the thread of execution reaches its definition. For example:

```
void f(int a)
{
    while (a--) {
        static int n = 0;          // initialized once
        int x = 0;                 // initialized n times

        cout << "n == " << n++ << ", x == " << x++ << '\n';
    }
}

int main()
{
    f(3);
}
```

This prints:

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

A static variable provides a function with ''a memory'' without introducing a global variable that might be accessed and corrupted by other functions (see also §10.2.4).

7.2 Argument Passing [fct.arg]

When a function is called, store is set aside for its formal arguments and each formal argument is initialized by its corresponding actual argument. The semantics of argument passing are identical to the semantics of initialization. In particular, the type of an actual argument is checked against the type of the corresponding formal argument, and all standard and user-defined type conversions are performed. There are special rules for passing arrays (§7.2.1), a facility for passing unchecked arguments (§7.6), and a facility for specifying default arguments (§7.5). Consider:

```
void f(int val, int& ref)
{
    val++;
    ref++;
}
```

When `f()` is called, ***val***++ increments a local copy of the first actual argument, whereas ***ref***++ increments the second actual argument. For example,

```
void g()
{
    int i = 1;
    int j = 1;
    f(i,j);
}
```

will increment *j* but not *i*. The first argument, *i*, is passed *by value*, the second argument, *j*, is passed *by reference*. As mentioned in §5.5, functions that modify call-by-reference arguments can make programs hard to read and should most often be avoided (but see §21.2.1). It can, however, be noticeably more efficient to pass a large object by reference than to pass it by value. In that case, the argument might be declared ***const*** to indicate that the reference is used for efficiency reasons only and not to enable the called function to change the value of the object:

```
void f(const Large& arg)
{
    // the value of "arg" cannot be changed without explicit use of type conversion
}
```

The absence of ***const*** in the declaration of a reference argument is taken as a statement of intent to modify the variable:

```
void g(Large& arg); // assume that g() modifies arg
```

Similarly, declaring a pointer argument ***const*** tells readers that the value of an object pointed to by that argument is not changed by the function. For example:

```
int strlen(const char*);                    // number of characters in a C-style string
char* strcpy(char* to, const char* from);   // copy a C-style string
int strcmp(const char*, const char*);       // compare C-style strings
```

The importance of using ***const*** arguments increases with the size of a program.

Note that the semantics of argument passing are different from the semantics of assignment. This is important for ***const*** arguments, reference arguments, and arguments of some user-defined types (§10.4.4.1).

A literal, a constant, and an argument that requires conversion can be passed as a ***const***& argument, but not as a non-***const*** argument. Allowing conversions for a ***const T***& argument ensures that such an argument can be given exactly the same set of values as a ***T*** argument by passing the value in a temporary, if necessary. For example:

```
float fsqrt(const float&);  // Fortran-style sqrt taking a reference argument

void g(double d)
{
    float r = fsqrt(2.0f);      // pass ref to temp holding 2.0f
    r = fsqrt(r);               // pass ref to r
    r = fsqrt(d);               // pass ref to temp holding float(d)
}
```

Disallowing conversions for non-***const*** reference arguments (§5.5) avoids the possibility of silly mistakes arising from the introduction of temporaries. For example:

```
void update(float& i);

void g(double d, float r)
{
    update(2.0f);   // error: const argument
    update(r);      // pass ref to r
    update(d);      // error: type conversion required
}
```

Had these calls been allowed, *update*() would quietly have updated temporaries that immediately were deleted. Usually, that would come as an unpleasant surprise to the programmer.

7.2.1 Array Arguments [fct.array]

If an array is used as a function argument, a pointer to its initial element is passed. For example:

```
int strlen(const char*);

void f()
{
    char v[] = "an array";
    int i = strlen(v);
    int j = strlen("Nicholas");
}
```

That is, an argument of type *T*[] will be converted to a *T** when passed as an argument. This implies that an assignment to an element of an array argument changes the value of an element of the argument array. In other words, arrays differ from other types in that an array is not (and cannot be) passed by value.

The size of an array is not available to the called function. This can be a nuisance, but there are several ways of circumventing this problem. C-style strings are zero-terminated, so their size can be computed easily. For other arrays, a second argument specifying the size can be passed. For example:

```
void compute1(int* vec_ptr, int vec_size);      // one way

struct Vec {
    int* ptr;
    int size;
};

void compute2(const Vec& v);        // another way
```

Alternatively, a type such as *vector* (§3.7.1, §16.3) can be used instead of an array.

Multidimensional arrays are trickier (see §C.7), but often arrays of pointers can be used instead, and they need no special treatment. For example:

```
char* day[] = {
    "mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

Again, *vector* and similar types are alternatives to the built-in, low-level arrays and pointers.

7.3 Value Return [fct.return]

A value must be returned from a function that is not declared ***void*** (however, ***main()*** is special; see §3.2). Conversely, a value cannot be returned from a ***void*** function. For example:

```
int f1() { }                // error: no value returned
void f2() { }               // ok

int f3() { return 1; }      // ok
void f4() { return 1; }     // error: return value in void function

int f5() { return; }        // error: return value missing
void f6() { return; }       // ok
```

A return value is specified by a return statement. For example:

```
int fac(int n) { return (n>1) ? n*fac(n-1) : 1; }
```

A function that calls itself is said to be *recursive*.

There can be more than one return statement in a function:

```
int fac2(int n)
{
    if (n > 1) return n*fac2(n-1);
    return 1;
}
```

Like the semantics of argument passing, the semantics of function value return are identical to the semantics of initialization. A return statement is considered to initialize an unnamed variable of the returned type. The type of a return expression is checked against the type of the returned type, and all standard and user-defined type conversions are performed. For example:

```
double f() { return 1; }  // 1 is implicitly converted to double(1)
```

Each time a function is called, a new copy of its arguments and local (automatic) variables is created. The store is reused after the function returns, so a pointer to a local variable should never be returned. The contents of the location pointed to will change unpredictably:

```
int* fp() { int local = 1; /* ... */ return &local; }    // bad
```

This error is less common than the equivalent error using references:

```
int& fr() { int local = 1; /* ... */ return local; }    // bad
```

Fortunately, a compiler can easily warn about returning references to local variables.

A ***void*** function cannot return a value. However, a call of a ***void*** function doesn't yield a value, so a ***void*** function can use a call of a ***void*** function as the expression in a ***return*** statement. For example:

```
void g(int* p);

void h(int* p) { /* ... */ return g(p); }  // ok: return of ''no value''
```

This form of return is important when writing template functions where the return type is a template parameter (see §18.4.4.2).

7.4 Overloaded Function Names [fct.over]

Most often, it is a good idea to give different functions different names, but when some functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name. Using the same name for operations on different types is called *overloading*. The technique is already used for the basic operations in C++. That is, there is only one name for addition, +, yet it can be used to add values of integer, floating-point, and pointer types. This idea is easily extended to functions defined by the programmer. For example:

```
void print(int);              // print an int
void print(const char*);      // print a C-style character string
```

As far as the compiler is concerned, the only thing functions of the same name have in common is that name. Presumably, the functions are in some sense similar, but the language does not constrain or aid the programmer. Thus overloaded function names are primarily a notational convenience. This convenience is significant for functions with conventional names such as ***sqrt***, ***print***, and ***open***. When a name is semantically significant, this convenience becomes essential. This happens, for example, with operators such as +, *, and <<, in the case of constructors (§11.7), and in generic programming (§2.7.2, Chapter 18). When a function ***f*** is called, the compiler must figure out which of the functions with the name ***f*** is to be invoked. This is done by comparing the types of the actual arguments with the types of the formal arguments of all functions called ***f***. The idea is to invoke the function that is the best match on the arguments and give a compile-time error if no function is the best match. For example:

```
void print(double);
void print(long);

void f()
{
    print(1L);      // print(long)
    print(1.0);     // print(double)
    print(1);       // error, ambiguous: print(long(1)) or print(double(1))?
}
```

Finding the right version to call from a set of overloaded functions is done by looking for a best match between the type of the argument expression and the parameters (formal arguments) of the functions. To approximate our notions of what is reasonable, a series of criteria are tried in order:

[1] Exact match; that is, match using no or only trivial conversions (for example, array name to pointer, function name to pointer to function, and ***T*** to ***const T***)

[2] Match using promotions; that is, integral promotions (***bool*** to ***int***, ***char*** to ***int***, ***short*** to ***int***, and their ***unsigned*** counterparts; §C.6.1), ***float*** to ***double***, and ***double*** to ***long double***

[3] Match using standard conversions (for example, ***int*** to ***double***, ***double*** to ***int***, ***Derived**** to ***Base**** (§12.2), T* to ***void**** (§5.6), ***int*** to ***unsigned int***; §C.6)

[4] Match using user-defined conversions (§11.4)

[5] Match using the ellipsis . . . in a function declaration (§7.6)

If two matches are found at the highest level where a match is found, the call is rejected as ambiguous. The resolution rules are this elaborate primarily to take into account the elaborate C and C++ rules for built-in numeric types (§C.6). For example:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
    print(c);       // exact match: invoke print(char)
    print(i);       // exact match: invoke print(int)
    print(s);       // integral promotion: invoke print(int)
    print(f);       // float to double promotion: print(double)

    print('a');     // exact match: invoke print(char)
    print(49);      // exact match: invoke print(int)
    print(0);       // exact match: invoke print(int)
    print("a");     // exact match: invoke print(const char*)
}
```

The call ***print(0)*** invokes ***print(int)*** because ***0*** is an ***int***. The call ***print('a')*** invokes ***print(char)*** because ***'a'*** is a ***char*** (§4.3.1). The reason to distinguish between conversions and promotions is that we want to prefer safe promotions, such as ***char*** to ***int***, over unsafe conversions, such as ***int*** to ***char***.

The overloading resolution is independent of the order of declaration of the functions considered.

Overloading relies on a relatively complicated set of rules, and occasionally a programmer will be surprised which function is called. So, why bother? Consider the alternative to overloading. Often, we need similar operations performed on objects of several types. Without overloading, we must define several functions with different names:

```
void print_int(int);
void print_char(char);
void print_string(const char*); // C-style string

void g(int i, char c, const char* p, double d)
{
    print_int(i);       // ok
    print_char(c);      // ok
    print_string(p);    // ok

    print_int(c);       // ok? calls print_int(int(c))
    print_char(i);      // ok? calls print_char(char(i))
    print_string(i);    // error
    print_int(d);       // ok? calls print_int(int(d))
}
```

Compared to the overloaded ***print()***, we have to remember several names and remember to use those correctly. This can be tedious, defeats attempts to do generic programming (§2.7.2), and generally encourages the programmer to focus on relatively low-level type issues. Because there is no overloading, all standard conversions apply to arguments to these functions. It can also lead to

errors. In the previous example, this implies that only one of the four calls with a ''wrong'' argument is caught by the compiler. Thus, overloading can increase the chances that an unsuitable argument will be rejected by the compiler.

7.4.1 Overloading and Return Type [fct.return]

Return types are not considered in overload resolution. The reason is to keep resolution for an individual operator (§11.2.1, §11.2.4) or function call context-independent. Consider:

```
float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
    float fl = sqrt(da);    // call sqrt(double)
    double d = sqrt(da);    // call sqrt(double)
    fl = sqrt(fla);         // call sqrt(float)
    d = sqrt(fla);          // call sqrt(float)
}
```

If the return type were taken into account, it would no longer be possible to look at a call of **sqrt( )** in isolation and determine which function was called.

7.4.2 Overloading and Scopes [fct.scope]

Functions declared in different non-namespace scopes do not overload. For example:

```
void f(int);

void g()
{
    void f(double);
    f(1);           // call f(double)
}
```

Clearly, **f(int)** would have been the best match for **f(1)**, but only **f(double)** is in scope. In such cases, local declarations can be added or subtracted to get the desired behavior. As always, intentional hiding can be a useful technique, but unintentional hiding is a source of surprises. When overloading across class scopes (§15.2.2) or namespace scopes (§8.2.9.2) is wanted, *using-declarations* or **using-directives** can be used (§8.2.2). See also §8.2.6 and §8.2.9.2.

7.4.3 Manual Ambiguity Resolution [fct.man.ambig]

Declaring too few (or too many) overloaded versions of a function can lead to ambiguities. For example:

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);
```

```
void k(int i)
{
    f1(i);      // ambiguous: f1(char) or f1(long)
    f2(0);      // ambiguous: f2(char*) or f2(int*)
}
```

Where possible, the thing to do in such cases is to consider the set of overloaded versions of a function as a whole and see if it makes sense according to the semantics of the function. Often the problem can be solved by adding a version that resolves ambiguities. For example, adding

```
inline void f1(int n) { f1(long(n)); }
```

would resolve all ambiguities similar to `f1(i)` in favor of the larger type ***long int***.

One can also add an explicit type conversion to resolve a specific call. For example:

```
f2(static_cast<int*>(0));
```

However, this is most often simply an ugly stopgap. Soon another similar call will be made and have to be dealt with.

Some C++ novices get irritated by the ambiguity errors reported by the compiler. More experienced programmers appreciate these error messages as useful indicators of design errors.

7.4.4 Resolution for Multiple Arguments [fct.fct.res]

Given the overload resolution rules, one can ensure that the simplest algorithm (function) will be used when the efficiency or precision of computations differs significantly for the types involved. For example:

```
int pow(int, int);
double pow(double, double);

complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);

void k(complex z)
{
    int i = pow(2,2);           // invoke pow(int,int)
    double d = pow(2.0,2.0);    // invoke pow(double,double)
    complex z2 = pow(2,z);      // invoke pow(double,complex)
    complex z3 = pow(z,2);      // invoke pow(complex,int)
    complex z4 = pow(z,z);      // invoke pow(complex,complex)
}
```

In the process of choosing among overloaded functions with two or more arguments, a best match is found for each argument using the rules from §7.4. A function that is the best match for one argument and a better than or equal match for all other arguments is called. If no such function exists, the call is rejected as ambiguous. For example:

```
void g()
{
    double d = pow(2.0,2);  // error: pow(int(2.0),2) or pow(2.0,double(2))?
}
```

The call is ambiguous because `2.0` is the best match for the first argument of `pow(double,double)` and `2` is the best match for the second argument of `pow(int,int)`.

7.5 Default Arguments [fct.defarg]

A general function often needs more arguments than are necessary to handle simple cases. In particular, functions that construct objects (§10.2.3) often provide several options for flexibility. Consider a function for printing an integer. Giving the user an option of what base to print it in seems reasonable, but in most programs integers will be printed as decimal integer values. For example:

```
void print(int value, int base =10); // default base is 10

void f()
{
    print(31);
    print(31,10);
    print(31,16);
    print(31,2);
}
```

might produce this output:

```
31 31 1f 11111
```

The effect of a default argument can alternatively be achieved by overloading:

```
void print(int value, int base);
inline void print(int value) { print(value,10); }
```

However, overloading makes it less obvious to the reader that the intent is to have a single print function plus a shorthand.

A default argument is type checked at the time of the function declaration and evaluated at the time of the call. Default arguments may be provided for trailing arguments only. For example:

```
int f(int, int =0, char* =0);    // ok
int g(int =0, int =0, char*);    // error
int h(int =0, int, char* =0);    // error
```

Note that the space between the `*` and the `=` is significant (`*=` is an assignment operator; §6.2):

```
int nasty(char*=0);              // syntax error
```

A default argument can be repeated in a subsequent declaration in the same scope but not changed. For example:

```
void f(int x = 7);
void f(int = 7);          // ok
void f(int = 8);          // error: different default arguments

void g()
{
    void f(int x = 9);    // ok: this declaration hides the outer one
    // ...
}
```

Declaring a name in a nested scope so that the name hides a declaration of the same name in an outer scope is error prone.

7.6 Unspecified Number of Arguments [fct.stdarg]

For some functions, it is not possible to specify the number and type of all arguments expected in a call. Such a function is declared by terminating the list of argument declarations with the ellipsis (`...`), which means ''and maybe some more arguments.'' For example:

```
int printf(const char* ...);
```

This specifies that a call of the C standard library function ***printf()*** (§21.8) must have at least one argument, a ***char****, but may or may not have others. For example:

```
printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d = %d\n",2,3,5);
```

Such a function must rely on information not available to the compiler when interpreting its argument list. In the case of ***printf()***, the first argument is a format string containing special character sequences that allow ***printf()*** to handle other arguments correctly; ***%s*** means ''expect a ***char****** argument'' and ***%d*** means ''expect an ***int*** argument.'' However, the compiler cannot in general know that, so it cannot ensure that the expected arguments are really there or that an argument is of the proper type. For example,

```
#include <stdio.h>

int main()
{
    printf("My name is %s %s\n",2);
}
```

will compile and (at best) cause some strange-looking output (try it!).

Clearly, if an argument has not been declared, the compiler does not have the information needed to perform the standard type checking and type conversion for it. In that case, a ***char*** or a ***short*** is passed as an ***int*** and a ***float*** is passed as a ***double***. This is not necessarily what the programmer expects.

A well-designed program needs at most a few functions for which the argument types are not completely specified. Overloaded functions and functions using default arguments can be used to

take care of type checking in most cases when one would otherwise consider leaving argument types unspecified. Only when both the number of arguments *and* the type of arguments vary is the ellipsis necessary. The most common use of the ellipsis is to specify an interface to C library functions that were defined before C++ provided alternatives:

```
int fprintf(FILE*, const char* ...);      // from <cstdio>
int execl(const char* ...);               // from UNIX header
```

A standard set of macros for accessing the unspecified arguments in such functions can be found in <*cstdarg*>. Consider writing an error function that takes one integer argument indicating the severity of the error followed by an arbitrary number of strings. The idea is to compose the error message by passing each word as a separate string argument. The list of string arguments should be terminated by a null pointer to *char*:

```
extern void error(int ...);
extern char* itoa(int, char[]);       // see §6.6[17]

const char* Null_cp = 0;

int main(int argc, char* argv[])
{
    switch (argc) {
    case 1:
        error(0,argv[0],Null_cp);
        break;

    case 2:
        error(0,argv[0],argv[1],Null_cp);
        break;

    default:
        char buffer[8];
        error(1,argv[0], "with",itoa(argc-1,buffer),"arguments", Null_cp);
    }
    // ...
}
```

The function *itoa*( ) returns the character string representing its integer argument.

Note that using the integer *0* as the terminator would not have been portable: on some implementations, the integer zero and the null pointer do not have the same representation. This illustrates the subtleties and extra work that face the programmer once type checking has been suppressed using the ellipsis.

The error function could be defined like this:

```
void error(int severity ...) // "severity" followed by a zero-terminated list of char*s
{
    va_list ap;
    va_start(ap,severity);      // arg startup
```

```
        for (;;) {
                char* p = va_arg(ap, char*);
                if (p == 0) break;
                cerr << p << ' ';
        }

        va_end(ap);                     // arg cleanup

        cerr << '\n';
        if (severity) exit(severity);
}
```

First, a ***va_list*** is defined and initialized by a call of ***va_start()***. The macro ***va_start*** takes the name of the ***va_list*** and the name of the last formal argument as arguments. The macro ***va_arg()*** is used to pick the unnamed arguments in order. In each call, the programmer must supply a type; ***va_arg()*** assumes that an actual argument of that type has been passed, but it typically has no way of ensuring that. Before returning from a function in which ***va_start()*** has been used, ***va_end()*** must be called. The reason is that ***va_start()*** may modify the stack in such a way that a return cannot successfully be done; ***va_end()*** undoes any such modifications.

7.7 Pointer to Function [fct.pf]

There are only two things one can do to a function: call it and take its address. The pointer obtained by taking the address of a function can then be used to call the function. For example:

```
void error(string s) { /* ... */ }

void (*efct)(string);           // pointer to function

void f()
{
        efct = &error;          // efct points to error
        efct("error");          // call error through efct
}
```

The compiler will discover that ***efct*** is a pointer and call the function pointed to. That is, dereferencing of a pointer to function using * is optional. Similarly, using & to get the address of a function is optional:

```
void (*f1)(string) = &error;    // ok
void (*f2)(string) = error;     // also ok; same meaning as &error

void g()
{
        f1("Vasa");             // ok
        (*f1)("Mary Rose");     // also ok
}
```

Pointers to functions have argument types declared just like the functions themselves. In pointer assignments, the complete function type must match exactly. For example:

```
void (*pf)(string);   // pointer to void(string)
void f1(string);      // void(string)
int f2(string);       // int(string)
void f3(int*);        // void(int*)

void f()
{
    pf = &f1;         // ok
    pf = &f2;         // error: bad return type
    pf = &f3;         // error: bad argument type

    pf("Hera");       // ok
    pf(1);            // error: bad argument type

    int i = pf("Zeus");   // error: void assigned to int
}
```

The rules for argument passing are the same for calls directly to a function and for calls to a function through a pointer.

It is often convenient to define a name for a pointer-to-function type to avoid using the somewhat nonobvious declaration syntax all the time. Here is an example from a UNIX system header:

```
typedef void (*SIG_TYP)(int);      // from <signal.h>
typedef void (*SIG_ARG_TYP)(int);
SIG_TYP signal(int, SIG_ARG_TYP);
```

An array of pointers to functions is often useful. For example, the menu system for my mouse-based editor is implemented using arrays of pointers to functions to represent operations. The system cannot be described in detail here, but this is the general idea:

```
typedef void (*PF)();

PF edit_ops[] = {          // edit operations
    &cut, &paste, &copy, &search
};

PF file_ops[] = {          // file management
    &open, &append, &close, &write
};
```

We can then define and initialize the pointers that control actions selected from a menu associated with the mouse buttons:

```
PF* button2 = edit_ops;
PF* button3 = file_ops;
```

In a complete implementation, more information is needed to define each menu item. For example, a string specifying the text to be displayed must be stored somewhere. As the system is used, the meaning of mouse buttons changes frequently with the context. Such changes are performed (partly) by changing the value of the button pointers. When a user selects a menu item, such as item 3 for button 2, the associated operation is executed:

```
button2[2]();  // call button2's 3rd function
```

One way to gain appreciation of the expressive power of pointers to functions is to try to write such code without them – and without using their better-behaved cousins, the virtual functions (§12.2.6). A menu can be modified at run-time by inserting new functions into the operator table. It is also easy to construct new menus at run-time.

Pointers to functions can be used to provide a simple form of polymorphic routines, that is, routines that can be applied to objects of many different types:

```
typedef int (*CFT)(const void*, const void*);

void ssort(void* base, size_t n, size_t sz, CFT cmp)
/*
    Sort the "n" elements of vector "base" into increasing order
    using the comparison function pointed to by "cmp".
    The elements are of size "sz".

    Shell sort (Knuth, Vol3, pg84)
*/

{
    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap) {
                char* b = static_cast<char*>(base);  // necessary cast
                char* pj = b+j*sz;                   // &base[j]
                char* pjg = b+(j+gap)*sz;            // &base[j+gap]

                if (cmp(pj,pjg)<0) {                 // swap base[j] and base[j+gap]:
                    for (int k=0; k<sz; k++) {
                        char temp = pj[k];
                        pj[k] = pjg[k];
                        pjg[k] = temp;
                    }
                }
            }
}
```

The *ssort*( ) routine does not know the type of the objects it sorts, only the number of elements (the array size), the size of each element, and the function to call to perform a comparison. The type of *ssort*( ) was chosen to be the same as the type of the standard C library sort routine, *qsort*( ). Real programs use *qsort*( ), the C++ standard library algorithm *sort* (§18.7.1), or a specialized sort routine. This style of code is common in C, but it is not the most elegant way of expressing this algorithm in C++ (see §13.3, §13.5.2).

Such a sort function could be used to sort a table such as this:

```
struct User {
    char* name;
    char* id;
    int dept;
};
```

```
User heads[] = {
    "Ritchie D.M",     "dmr",    11271,
    "Sethi R.",        "ravi",   11272,
    "Szymanski T.G.",  "tgs",    11273,
    "Schryer N.L.",    "nls",    11274,
    "Schryer N.L.",    "nls",    11275,
    "Kernighan B.W.", "bwk",     11276
};

void print_id(User* v, int n)
{
    for (int i=0; i<n; i++)
        cout << v[i].name << '\t' << v[i].id << '\t' << v[i].dept << '\n';
}
```

To be able to sort, we must first define appropriate comparison functions. A comparison function must return a negative value if its first argument is less than the second, zero if the arguments are equal, and a positive number otherwise:

```
int cmp1(const void* p, const void* q)   // Compare name strings
{
    return strcmp(static_cast<const User*>(p)->name, static_cast<const User*>(q)->name);
}

int cmp2(const void* p, const void* q)   // Compare dept numbers
{
    return static_cast<const User*>(p)->dept - static_cast<const User*>(q)->dept;
}
```

This program sorts and prints:

```
int main()
{
    cout << "Heads in alphabetical order:\n";
    ssort(heads, 6, sizeof(User), cmp1);
    print_id(heads, 6);
    cout << "\n";

    cout << "Heads in order of department number:\n";
    ssort(heads, 6, sizeof(User), cmp2);
    print_id(heads, 6);
}
```

You can take the address of an overloaded function by assigning to or initializing a pointer to function. In that case, the type of the target is used to select from the set of overloaded functions. For example:

```
void f(int);
int f(char);

void (*pf1)(int) = &f;      // void f(int)
int (*pf2)(char) = &f;      // int f(char)
void (*pf3)(char) = &f;     // error: no void f(char)
```

A function must be called through a pointer to function with exactly the right argument and return types. There is no implicit conversion of argument or return types when pointers to functions are assigned or initialized. This means that

```
int cmp3(const mytype*,const mytype*);
```

is not a suitable argument for *ssort()*. The reason is that accepting *cmp3* as an argument to *ssort()* would violate the guarantee that *cmp3* will be called with arguments of type *mytype** (see also §9.2.5).

7.8 Macros [fct.macro]

Macros are very important in C but have far fewer uses in C++. The first rule about macros is: Don't use them unless you have to. Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer. Because they rearrange the program text before the compiler proper sees it, macros are also a major problem for many programming tools. So when you use macros, you should expect inferior service from tools such as debuggers, cross-reference tools, and profilers. If you must use macros, please read the reference manual for your own implementation of the C++ preprocessor carefully and try not to be too clever. Also to warn readers, follow the convention to name macros using lots of capital letters. The syntax of macros is presented in §A.11.

A simple macro is defined like this:

```
#define NAME rest of line
```

Where *NAME* is encountered as a token, it is replaced by *rest of line*. For example,

```
named = NAME
```

will expand into

```
named = rest of line
```

A macro can also be defined to take arguments. For example:

```
#define MAC(x,y) argument1: x argument2: y
```

When *MAC* is used, two argument strings must be presented. They will replace *x* and *y* when *MAC()* is expanded. For example,

```
expanded = MAC(foo bar, yuk yuk)
```

will be expanded into

```
expanded = argument1: foo bar argument2: yuk yuk
```

Macro names cannot be overloaded, and the macro preprocessor cannot handle recursive calls:

```
#define PRINT(a,b) cout<<(a)<<(b)
#define PRINT(a,b,c) cout<<(a)<<(b)<<(c) /* trouble?: redefines, does not overload */
#define FAC(n) (n>1)?n*FAC(n-1):1        /* trouble: recursive macro */
```

Macros manipulate character strings and know little about C++ syntax and nothing about C++ types or scope rules. Only the expanded form of a macro is seen by the compiler, so an error in a macro will be reported when the macro is expanded, not when it is defined. This leads to very obscure error messages.

Here are some plausible macros:

```
#define CASE break;case
#define FOREVER for(;;)
```

Here are some completely unnecessary macros:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

Here are some dangerous macros:

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
```

To see why they are dangerous, try expanding this:

```
int xx = 0;       // global counter

void f()
{
      int xx = 0;                 // local variable
      int y = SQUARE(xx+2);       // y=xx+2*xx+2; that is y=xx+(2*xx)+2
      INCR_xx;                    // increments local xx
}
```

If you must use a macro, use the scope resolution operator `::` when referring to global names (§4.9.4) and enclose occurrences of a macro argument name in parentheses whenever possible. For example:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

If you must write macros complicated enough to require comments, it is wise to use `/* */` comments because C preprocessors that do not know about `//` comments are sometimes used as part of C++ tools. For example:

```
#define M2(a) something(a)    /* thoughtful comment */
```

Using macros, you can design your own private language. Even if you prefer this ''enhanced language'' to plain C++, it will be incomprehensible to most C++ programmers. Furthermore, the C preprocessor is a very simple macro processor. When you try to do something nontrivial, you are likely to find it either impossible or unnecessarily hard to do. The ***const***, ***inline***, ***template***, and ***namespace*** mechanisms are intended as alternatives to many traditional uses of preprocessor constructs. For example:

```
const int answer = 42;
template<class T> inline T min(T a, T b) { return (a<b)?a:b; }
```

When writing a macro, it is not unusual to need a new name for something. A string can be created by concatenating two strings using the ## macro operator. For example,

```
#define NAME2(a,b) a##b

int NAME2(hack,cah)();
```

will produce

```
int hackcah();
```

for the compiler to read.

The directive

```
#undef X
```

ensures that no macro called ***X*** is defined – whether or not one was before the directive. This affords some protection against undesired macros. However, it is not always easy to know what the effects of ***X*** on a piece of code were supposed to be.

7.8.1 Conditional Compilation [fct.cond]

One use of macros is almost impossible to avoid. The directive ***#ifdef identifier*** conditionally causes all input to be ignored until a ***#endif*** directive is seen. For example,

```
int f(int a
#ifdef arg_two
, int b
#endif
);
```

produces

```
int f(int a
);
```

for the compiler to see unless a macro called ***arg_two*** has been ***#define***d. This example confuses tools that assume sane behavior from the programmer.

Most uses of ***#ifdef*** are less bizarre, and when used with restraint, ***#ifdef*** does little harm. See also §9.3.3.

Names of the macros used to control ***#ifdef*** should be chosen carefully so that they don't clash with ordinary identifiers. For example:

```
struct Call_info {
        Node* arg_one;
        Node* arg_two;
        // ...
};
```

This innocent-looking source text will cause some confusion should someone write:

```
#define arg_two x
```

Unfortunately, common and unavoidable headers contain many dangerous and unnecessary macros.

7.9 Advice [dcl.advice]

[1] Be suspicious of non-***const*** reference arguments; if you want the function to modify its arguments, use pointers and value return instead; §5.5.
[2] Use ***const*** reference arguments when you need to minimize copying of arguments; §5.5.
[3] Use ***const*** extensively and consistently; §7.2.
[4] Avoid macros; §7.8.
[5] Avoid unspecified numbers of arguments; §7.6.
[6] Don't return pointers or references to local variables; §7.3.
[7] Use overloading when functions perform conceptually the same task on different types; §7.4.
[8] When overloading on integers, provide functions to eliminate common ambiguities; §7.4.3.
[9] When considering the use of a pointer to function, consider whether a virtual function (§2.5.5) or a template (§2.7.2) would be a better alternative; §7.7.
[10] If you must use macros, use ugly names with lots of capital letters; §7.8.

7.10 Exercises [fct.exercises]

1. (∗1) Write declarations for the following: a function taking arguments of type pointer to character and reference to integer and returning no value; a pointer to such a function; a function taking such a pointer as an argument; and a function returning such a pointer. Write the definition of a function that takes such a pointer as an argument and returns its argument as the return value. Hint: Use ***typedef***.
2. (∗1) What does the following mean? What would it be good for?

```
typedef int (&rifii) (int, int);
```

3. (∗1.5) Write a program like ''Hello, world!'' that takes a name as a command-line argument and writes ''Hello, *name* !''. Modify this program to take any number of names as arguments and to say hello to each.
4. (∗1.5) Write a program that reads an arbitrary number of files whose names are given as command-line arguments and writes them one after another on ***cout***. Because this program concatenates its arguments to produce its output, you might call it ***cat***.
5. (∗2) Convert a small C program to C++. Modify the header files to declare all functions called and to declare the type of every argument. Where possible, replace #***define***s with ***enum***, ***const***, or ***inline***. Remove ***extern*** declarations from `.c` files and if necessary convert all function definitions to C++ function definition syntax. Replace calls of ***malloc***() and ***free***() with ***new*** and ***delete***. Remove unnecessary casts.
6. (∗2) Implement ***ssort***() (§7.7) using a more efficient sorting algorithm. Hint: ***qsort***().
7. (∗2.5) Consider:

```
struct Tnode {
    string word;
    int count;
    Tnode* left;
    Tnode* right;
};
```

Write a function for entering new words into a tree of ***Tnode***s. Write a function to write out a tree of ***Tnode***s. Write a function to write out a tree of ***Tnode***s with the words in alphabetical order. Modify ***Tnode*** so that it stores (only) a pointer to an arbitrarily long word stored as an array of characters on free store using ***new***. Modify the functions to use the new definition of ***Tnode***.

8. (∗2.5) Write a function to invert a two-dimensional array. Hint: §C.7.
9. (∗2) Write an encryption program that reads from ***cin*** and writes the encoded characters to ***cout***. You might use this simple encryption scheme: the encrypted form of a character ***c*** is ***c^key[i]***, where ***key*** is a string passed as a command-line argument. The program uses the characters in ***key*** in a cyclic manner until all the input has been read. Re-encrypting encoded text with the same key produces the original text. If no key (or a null string) is passed, then no encryption is done.
10. (∗3.5) Write a program to help decipher messages encrypted with the method described in §7.10[9] without knowing the key. Hint: See David Kahn: *The Codebreakers*, Macmillan, 1967, New York, pp. 207-213.
11. (∗3) Write an ***error*** function that takes a ***printf***-style format string containing ***%s***, ***%c***, and ***%d*** directives and an arbitrary number of arguments. Don't use ***printf()***. Look at §21.8 if you don't know the meaning of ***%s***, ***%c***, and ***%d***. Use ***<cstdarg>***.
12. (∗1) How would you choose names for pointer to function types defined using ***typedef***?
13. (∗2) Look at some programs to get an idea of the diversity of styles of names actually used. How are uppercase letters used? How is the underscore used? When are short names such as ***i*** and ***x*** used?
14. (∗1) What is wrong with these macro definitions?

```
#define PI = 3.141593;
#define MAX(a,b) a>b?a:b
#define fac(a) (a)*fac((a)-1)
```

15. (∗3) Write a macro processor that defines and expands simple macros (like the C preprocessor does). Read from ***cin*** and write to ***cout***. At first, don't try to handle macros with arguments. Hint: The desk calculator (§6.1) contains a symbol table and a lexical analyzer that you could modify.
16. (∗2) Implement ***print()*** from §7.5.
17. (∗2) Add functions such as ***sqrt()***, ***log()***, and ***sin()*** to the desk calculator from §6.1. Hint: Predefine the names and call the functions through an array of pointers to functions. Don't forget to check the arguments in a function call.
18. (∗1) Write a factorial function that does not use recursion. See also §11.14[6].
19. (∗2) Write functions to add one day, one month, and one year to a ***Date*** as defined in §5.9[13]. Write a function that gives the day of the week for a given ***Date***. Write a function that gives the ***Date*** of the first Monday following a given ***Date***.