Advanced Programming in C

Rex Jaeschke

Advanced Programming in C

Edition: 3.0

The training materials associated with this book are available for license. Interested parties should contact the author.

Please address comments, corrections, and questions to the author, Rex Jaeschke, at rex@RexJaeschke.com.

Much of the information in this book has been published previously, as follows:

1. A series of monthly columns called "Let's C Now" in *DEC Professional*, Professional Press, running from 1984–1995.
2. The book "Mastering Standard C", Rex Jaeschke, 2e, CBM Books 1996, ISBN 1-878956-55-8.

The glossary of terms from "Mastering Standard C", 1e, was spun-off and expanded into its own book, "The Dictionary of Standard C", Prentice Hall 2001, ISBN 0-13-090620-4. This dictionary can be viewed on-line, free of charge, at www.prenhall.com/jaeschke. An earlier version of this dictionary, now out of print, was published by Professional Press Books 1991, ISBN 0-07-707657-5.

Preface

This book covers a number of more advanced (or esoteric) language and library topics.

Reader Assumptions

To fully understand and exploit the material, you should be conversant with the information presented in the companion volume, “Programming in C”, and preferably have more than a little experience in actually writing C code. The main topics covered by that volume are:

- Formatted I/O using printf and scanf
- Built-in data types
- Literals and identifiers
- The type qualifier `const`
- Type synonyms
- Automatic and static storage durations
- The operators, their precedence, and order of evaluation
- Type conversion
- All the statements
- Arrays and strings
- Functions
- The preprocessor
- Basic data pointer use
- Dynamic memory allocation
- Structures, bit-Fields, and unions
- Implementation-defined behavior

and any associated library headers and functions.

Limitations

GUI, calling non-C routines, inter-process communications, and operating system-specific features are outside the scope of this text.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming language training for more than 20 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and focused. Many books use GUI and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than is any other; I simply know that my approach works well, and has formed the basis of my successful seminar business.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory, within which each program may well have its own subdirectory. For example, the source code for the program called in01 in the "Internationalization" chapter can be found in the following directory hierarchy: Source, Internationalization, in01. By convention, the names of C source files end in "`.c`".

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided electronically in a directory tree named Labs, where each chapter has its own subdirectory, within which each program has its own subdirectory.[1] For example, lab solution lbco02 in the "Comma Operator" chapter has the following fully qualified directory hierarchy: Labs, CommaOperator, lbco02.

Exercises that are not so marked have no general solution and may require experimentation or research in an implementation's documentation.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

The Status of Standard C

The history of the standardization of C is as follows:

- C89 – The first ANSI C standard, ANSI X3.159-1989, was produced in 1989 by the U.S. committee X3J11.
- C90 – The first ISO C standard, ISO/IEC 9899:1990, was produced in 1990 by committee ISO/IEC JTC 1/SC 22/WG 14 in conjunction with committee X3J11. C90 was technically equivalent to C89.
- C95 – An amendment to C90 was produced in 1995 by committee WG 14 in conjunction with the U.S. committee X3J11. The additions included digraphs, the header `iso646.h`, and many multibyte and wide-character functions via the headers `wchar.h` and `wctype.h`.
- C99 – The second edition of the ISO C standard, ISO/IEC 9899:1999, was produced by committee WG14 in conjunction with the U.S. committee INCITS/J11 (formerly X3J11). The additions included a few language features, a number of headers, and many library functions. Throughout its development, C99 was commonly referred to as C9x.
- C11 – The third edition of the ISO C standard, ISO/IEC 9899:2011, was produced by committee WG14 in conjunction with the U.S. committee INCITS/PL22.11 (formerly INCITS/J11). The additions included support for multiple threads of execution, processing Unicode characters and strings, and the querying and specification of the alignment of objects, among other things.
- C17 – The fourth edition of the ISO C standard, ISO/IEC 9899:2017, was produced by committee WG14 in conjunction with the U.S. committee INCITS/PL22.11. This was a maintenance release that included corrections to Defect Reports. No new functionality was added.

The C standards committee is currently working on various maintenance issues and informative Technical Reports.

Electronic copies of the latest C standard can be purchased from www.ansi.org or www.iso.ch.

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my C seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, June 2018

An example containing one of each is:

```
int i;          /* basic type,    int                    */
int a[10];      /* derived type, array of 10 int         */
int f(void);    /* derived type, function returning int */
int *pi;        /* derived type, pointer to int          */
```

The simple rule to remember here involves the position of the derived type punctuation in the declarator. In the case of arrays and functions, the [] and () are postfix punctuators; that is, they follow immediately after the identifier to which they apply. The pointer notation * is a prefix punctuator, so it comes immediately before the identifier to which it applies.

3.4 Deriving from a Derived Type

Since a derived type is derived from another type, it follows that we can derive a type from another derived type, which in turn was derived from another type, *ad infinitum*. Some simple examples follow:

```
char **ppc;             /* ppc is a ptr to a ptr to a char           */

char ***pppc;           /* pppc is a ptr to a ptr to a ptr to a char */

long table[10][5];      /* table is an array of 10 elements each of  */
                        /* which is an array of 5 elements each of   */
                        /* which is a long                           */

long counts[5][4][6];   /* counts is an array of 5 elements each of  */
                        /* which is an array of 4 elements each of   */
                        /* which is an array of 6 elements each of   */
                        /* which is a long                           */
```

All derived types ultimately come down to a basic type, as well they must, since the basic types are the only ones for which we have keywords.

In the case of a pointer declarator, * is a prefix punctuator so we simply add an extra * in front of a valid type declarator until we get the desired level of indirection. With arrays, the situation is very similar except that [] is a postfix punctuator and is added to the end of the type declarator we wish to modify. This gives rise to the notion of a multidimensional array being an array each of whose elements is an array, and so on, until the final dimension array contains either objects of a basic or derived type.

In the declarations above, we have derived a type using the same punctuator multiple times. We have omitted the () punctuator since it cannot be applied to itself; that is, a function cannot return another function. We can combine these three punctuators giving rise to other possibilities; however, not all such combinations are valid. Two valid combinations are:

```
char *keywords[10];     /* keywords is an array of 10 elements each of */
                        /* which is a ptr to a char                    */

double *test(void);     /* test is a function returning a ptr to double */
```

We can derive in three ways from a derived type, which was, itself, derived in one of three ways, giving nine possible combinations. They are:

Table 3-1: Valid Derived Type Combinations

Derived type/"Basic type"	Pointer	Array	Function
Pointer to	valid	valid	valid
Array of	valid	valid	invalid
Function returning	valid	invalid	invalid

Reproducing this table at will is easy: The three "invalids" should be obvious; a function cannot return an array or function, and we cannot have an array of functions. All other entries are "valid". The only unusual "valid" is pointer to array.[1]

3.5 Precedence of Punctuators

Once a declarator contains more than one occurrence of the three punctuators `[]`, `()`, and `*`, we need to be concerned about the order in which they apply to the identifier. For example, does `char *keywords[10]` declare `keywords` to be an array of 10 pointers to `char` or a pointer to an array of 10 `char`? According to the table above, both are possible, so which is it, and how do we write the other?

It is no coincidence that these punctuators are also used as operators in an identical context. Consider the following example:

```
void f()
{
/*1*/   char *keywords[10];
        char c;

/*2*/   c = *keywords[0];
}
```

Given the declaration in case 1, we can see that regardless of whatever `keywords` itself is, an expression of the form `*keywords[i]` has type `char`. Therefore, we can assign such an expression to the `char` variable `c`, as shown. In case 2, the operator precedence table tells us that `[]` takes precedence over unary `*`. Therefore, `keywords` is subscripted giving a pointer to `char` which is then dereferenced to give the value of the `char` to which it points, and that `char`'s value is assigned to `c`.

What we see then is that the order of precedence of operators in the expression is *identical* to that of the same characters used as punctuators in a declarator. So, we can talk about precedence of evaluation of operators in expressions and precedence of binding of punctuators in declarations. In any event, we use the precedence table to resolve both.

[1] For more information on pointers to arrays, refer to §6.2.

Using this information let's reconsider the declarator `char *keywords[10]`. Since `[]` takes precedence over *, `keywords` is first and foremost an array of 10 elements, each of which is a pointer that points to a `char`.

Returning to our earlier example, once we apply the associativity suggested by the precedence table, we get the following declarators:

```
char **ppc;             /* * associates right-to-left  */
char ***pppc;           /*      "       "        "     */
long table[10][5];      /* [] associates left-to-right */
long counts[5][4][6];   /*      "       "        "     */
```

In the case of `ppc`, we have two punctuators with the same precedence. However, the precedence table indicates that multiple * operators (and punctuators) associate right to left. In the case of multiple `[]` punctuators, they associate left to right, as would a combination of `[]` and `()`.

3.6 Forcing Punctuator Precedence

If `char *keywords[10]` declares `keywords` to be an array of 10 pointers to `char`, how do we declare a pointer to an array of 10 `char`? Both declarators require a prefix * and a postfix `[]`.

The way we change the precedence of operators in an expression is to use grouping parentheses. We can also use them to change the precedence of binding of punctuators in a declarator, as follows:

```
char (*pa)[10]; /* pa is a ptr to an array of 10 elements of type char */
```

Here, the parentheses force the * to bind closer to `pa` than does `[]` causing `pa` to be first, and foremost, a pointer, which points to an array of 10 elements, each of which is a `char`.

Just as we can have redundant grouping parentheses in expressions, they can also exist in declarations. Therefore, some of our earlier declarations can be rewritten as follows:

```
char (*(*ppc));
long ((table[10])[5]);
char (*(keywords[10]));
double (*(test(void)));
```

In these cases, the grouping parentheses simply document the default binding. Once we have mastered the reading of such declarations, we generally would not use the extra parentheses, since they tend to clutter up the declarator.

3.7 Writing Declarations

From our 3×3 type derivation table, we know the limits of type derivation. Now, we can apply this information to write any arbitrarily complex declarator. Let's start with a relatively simple one:

Case 1: `pf` is a `static` pointer to a function that takes one argument of type "pointer to `const char`" and returns a pointer to an `int`.

It is most important that the English description be written left-to-right, which is the reverse order of the levels of derivation. Specifically, the name of the identifier being declared goes to the extreme left and the term on the extreme right must be a basic type keyword. The trick to converting these words into the corresponding declaration is to work from left to right.

The sentence describing a derived type must be made up of a combination of only three phrases: "an array of *n* elements each of which is", "a function returning", and "a pointer to". Only the six valid combinations shown in the type derivation table are permitted. Once we have written the description, if none of the three invalid combinations is present, the description can be turned into a valid declarator.

Before we start writing the corresponding declarator, it is useful to rewrite the description leaving out all function argument, storage class, and type qualifier information. We shall explain why later. The revised description then is:

Case 2: `pf` is a pointer to a function that returns a pointer to an `int`.

Let's start writing the declarator. First, `pf` is a pointer:

```
(*pf)
```

Since the pointer notation uses a prefix punctuator we write the * before the `pf`. We surround `*pf` with parentheses to ensure that the * binds tightest regardless of any other punctuators that might follow.

This pointer points to a function, so we add the postfix function-call punctuator and the binding parentheses:

```
((*pf)())
```

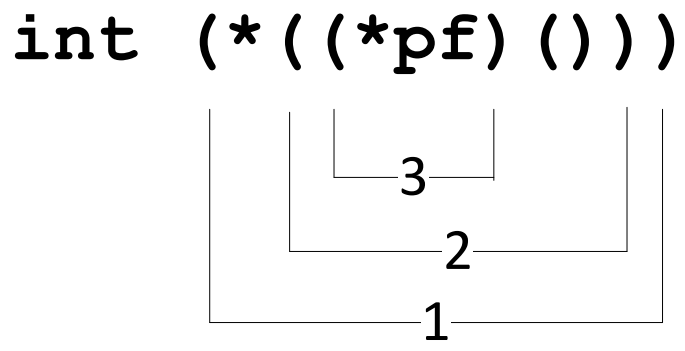
This function returns a pointer, so we add the prefix * and yet another pair of grouping parentheses:

```
(*((*pf)()))
```

And, finally, this pointer points to an `int`, so we add the basic type `int` as a prefix, giving the syntactically complete declarator:

```
int (*((*pf)()))
```

We have added grouping parentheses at every step just in case they were needed. In this example, as often happens, one or more of the pairs are redundant and can be removed. (Of course, leaving them in causes no harm unless we feel it hurts readability.) Let's determine which sets of parentheses are redundant. To make it easier, we'll number the sets of grouping parentheses (being careful to ignore the function-call parentheses), as follows:



Clearly, pair 1 is redundant, since no other punctuators are outside them. Pair 2 is also redundant, since `()` has higher precedence than *. If we erase those two pairs, we get:

```
int *(*pf)()
```


Now we have one pair left, pair 3. They ensure that * has precedence over `()` and, since this is not the case by default, the parentheses are needed. If they were removed (as in `int **pf();`), `pf` would become a function that returns a pointer to a pointer to an `int`, a quite different type.

Now we can add the function argument, storage class, and type-qualifier information we removed earlier. Each argument to a function can be described using an English sentence and turned into a declarator using the rules we have just learned. We can add the argument information in all function-call parentheses, and then add any type qualifiers. There can be one storage class keyword only in a declarator and that goes at the front. The complete declarator then is:

```
static int *(*pf)(const char *arg)
```

Exercise 3-1: Convince yourself that the declarators shown below match their English descriptions:

1. `f` is a function that takes no arguments and returns a pointer to a function that takes no arguments and returns a pointer to an array of 10 elements each of which is a structure of type `tag`.

```
struct tag (*(*f(void))(void))[10];
```

2. `p` is a pointer to a pointer to an array of 5 elements, each of which is a pointer to a `double`.

```
double *(**p)[5];
```

3. `a` as an array of 10 elements, each of which has 5 elements, each of which is a pointer to a function taking no arguments and returning a `long`.

```
long int (*a[10][5])(void);
```

3.8 Reading Declarations

Once we understand the rules for writing a declarator, reading a declaration is easy—we simply apply the rules in reverse. The only difference is that the declaration almost certainly does not contain redundant grouping parentheses, in which case, we will have to revert to the precedence table to determine the order of binding of the punctuators.

If the declarator is complex, follow these steps in order:

1. Remove all function-argument, storage-class, and type-qualifier information. Using the example we just wrote, this results in `int *(*pf)()`.
2. Make sure there are grouping parentheses around every term/punctuator pair. We now have `int (*((*pf)()))`.
3. Read the declarator starting from the identifier being declared. (Since function arguments were removed in the first step, there will be only one identifier left in the declarator.) We now read, "`pf` is a pointer to a function that returns a pointer to an `int`".
4. Add text to the declarator that corresponds to all function-argument, storage-class, and type-qualifier information removed in step 1.

Let's use the information we have learned to determine what the declaration of `signal` means:

```
void (*signal(int signal_type, void (*action)(int)))(int);
```

`signal` is a function that takes two arguments, the first having type `int` and the second having type "pointer to function taking one argument of type `int` and returning `void`". `signal` returns a value of type "pointer to function taking one argument of type `int` and returning `void`". The type of the return value is the same as that of the second argument.

> **Exercise 3-2:** Convince yourself the following declarators mean what the narrative says:
>
> 1. `register long int (*(*x)(const double *pd))[5];`
> declares `x` as a `register` pointer to a function taking an argument of type "pointer to `const double`" and returning a pointer to an array of 5 `long int`s.
>
> 2. `enum color *(*(*(*a[5])[3])(void))[6];`
> declares `a` to be an array of 5 pointers to an array of 3 pointers to functions taking no arguments and returning a pointer to an array of 6 pointers to `enum`s of type `color`.

3.9 Using Type Information

Obviously, it is necessary to know how to construct a declaration if we are ever to have an identifier of that type. However, once we have mastered the reading and writing of declarations, our job is not completed. To exploit fully the language, we must learn how to apply that type information in other ways.

C uses type information in a number of contexts. Apart from identifier declarations, types can be used in casts, with `sizeof`, and in defining type synonyms with `typedef`. We will look at the rules in constructing each of these and will apply them to a series of types, which have an increasing degree of complexity.

The following rules describe how to apply type information. Rule 1 explains how to extract type information from a declaration while rules 2–4 explain how to build a cast, use `sizeof`, and use `typedef`, respectively.

Rule 1: To extract the type information from a declarator, omit the identifier. The following table shows some examples of declarators and their corresponding types:

Table 3-2: Some Declarations and their Corresponding Types

Declaration	Type
`double d`	`double`
`short int s[6]`	`short int [6]`
`struct tags *pstr`	`struct tags *`
`union tagu (*pun)[7]`	`union tagu (*)[7]`
`enum tage (*(*pf)(void))[3]`	`enum tage (*(*)(void))[3]`

Rule 2: To generate a cast for type *T*, use Rule 1 to extract the type information, enclose it in parentheses, and use it as a prefix operator.

Rule 3: To find the size of an object of type *T*, use Rule 1 to extract the type information, enclose it in parentheses, and use it as the postfix operand of the `sizeof` operator.

Rule 4: To create a synonym for a type, declare an identifier to have that type. Add the keyword `typedef` in front of the declaration; now that identifier is a synonym for that type.

Let's apply these rules to four declarators:

Table 3-3: A Pointer to `char`

Action	Syntax
declaration	`char *pc;`
cast	`(char *)` *expression*
`sizeof`	`sizeof(char *)`
type synonym	`typedef char *PTC;`

Table 3-4: An Array of 5 Pointers to `char`

Action	Syntax
declaration	`char *ap[5];`
cast	`(char *[5])` *expression*
`sizeof`	`sizeof(char *[5])`
type synonym	`typedef char *A5PTC[5];`

The cast generated by the rules in this example is invalid—we cannot cast to an array type.

Table 3-5: A Pointer to an Array of 5 `char`

Action	Syntax
declaration	`char (*pa)[5];`
cast	`(char (*)[5])` *expression*
`sizeof`	`sizeof(char (*)[5])`
type synonym	`typedef char (*PA5C)[5];`

The parentheses around * are not redundant—they serve to distinguish between an array of pointers and a pointer to an array.

Table 3-6: A Pointer to a Function Returning an `int`

Action	Syntax
declaration	`int (*pf)(void);`
cast	`(int (*)(void))` *expression*
`sizeof`	`sizeof(int (*)(void))`
type synonym	`typedef int (*PTFRI)(void);`

Again, the parentheses around * are not redundant.

Exercise 3-3: Look at the definitions of the macros `SIG_DFL`, `SIG_ERR`, and `SIG_IGN` in `signal.h`. Typically, they involve a complex cast expression.

6. The Comma Operator

6.1 The Basics

One of the most confusing (and consequently least used) operators is the comma. The confusing aspect is that this operator is represented by the comma, yet almost every comma token present in a C program is a comma punctuator, *not* an operator. Despite its unfortunate spelling, the comma operator is a very powerful tool, as we shall see.

Consider the following example:

```
i = 0;
j = 0;
for (k = 0; k > 0; --k)
{
        /* … */
        ++i;
        ++j;
}
```

By definition, the `for` construct contains three optional expressions separated by semicolons. In this example, each expression is simple. We can rewrite this example as follows:

```
for (i = 0, j = 0, k = 0; k > 0; ++i, ++j, --k)
{
        /* … */
}
```

To do more than one thing at initialization and at the end of each iteration of the loop, we can specify a set of expressions separated by comma operators. As shown above, each comma operator expression contains three subexpressions. The comma operator allows us to paste together an arbitrary number of expressions and have them treated syntactically as one large expression.

The comma is a binary operator, and its two operands can be expressions of any data type, including `void`. It has the lowest precedence of all the operators, and it associates left-to-right. A comma expression has the following general form:

exp1 , *exp2*

The left operand, *exp1*, is evaluated, and its value is discarded. Therefore, to be useful, *exp1* must contain a side-effect. Then the right operand, *exp2*, is evaluated, and its type and value, if any, become that of the whole comma expression. Consider the following example. It is not intended to produce a useful result; it merely demonstrates the syntax. In fact, each statement can be rewritten more clearly, as shown in the corresponding comment:

```
int f();
int g();
int i;
int j = 10;

void test()
{
/*1*/   i = (f(), g());         /* f(); i = g(); */
/*2*/   i = f(), g();           /* i = f(); g(); */
/*3*/   i = (j++, ++j);         /* j += 2; i = j; */
/*4*/   i = 5, j = i, f();      /* i = 5; j = i; f(); */
}
```

In case 1, function `f` is called and the `int` value returned is discarded. Then function `g` is called, and the `int` value it returns becomes the value of the whole comma expression. This value is then assigned to `i`. Note that the grouping parentheses force the comma to take precedence over the assignment.

In case 2, function `f` is called and the value it returns is assigned to `i`, since assignment has higher precedence than comma. The assignment side-effect is completed before the right operand is evaluated. This is important, because `g` might access the global `i`. Then function `g` is called, and the value it returns is used as that of the whole comma expression. In this case, the result actually is discarded, but that is not a property of the comma operator—we simply failed to use it.

In case 3, `j++` is evaluated. Again, the side-effect is completed before the right operand is evaluated. Then the right operand is evaluated, and its value is assigned to `i`.

Since the order of evaluation is guaranteed by each comma in case 4, it can easily be rewritten as three separate statements as shown.

There are situations in which comma operators and punctuators can be used in the same context. For example:

```
/*1*/   f(i, j)
/*2*/   g((++i, j))
```

An argument list is a possibly empty list of expressions separated by comma punctuators. Therefore, function `f` is called with two arguments, `i` and `j`. Function `g`, on the other hand, is called with only one argument, since the expression in the argument list is the comma expression `(++i, j)`. The outer parentheses represent the function call operator, while the inner pair represent grouping.

The comma operator has two important properties. First, it allows an arbitrary number of expressions to be pasted together, yet syntactically they are all considered part of one big expression. Second, the type and value of the right-most expression percolates to the front, becoming the type and value of the result.

It should be obvious that the comma operator can detract from a program's readability and we should avoid using it as much as possible.

A common misuse of the comma operator is as follows:

```
for (exp1, exp2, exp3)
{
        /* … */
}
```


The two commas should be semicolons. However, the compiler treats them as comma operators and does not complain about missing semicolons until the right parenthesis is seen.

Exercise 6-1: What is the value of `sizeof(c++,c)` where `c` has type `char`? What is the significance of the ++?

Exercise 6-2: Consider the following program fragments. Can you find any use for such constructs?

```
while (f(), i)
        /* … */

for ( exp1 ; f(), i; exp3 )
        /* … */
```

Exercise 6-3: Programmers new to C, or those who switch between C and other languages, sometimes write multidimensional array subscripts as `a[i,j]` instead of `a[i][j]`. What kind of message, if any, does your compiler produce in such cases? If `a` is a two-dimensional array of `char` having size 4×6, what is the type of the expression `a[i,j]`?

6.2 A Function Trace Facility

In this section, we examine a relatively simple technique for producing an audit trail of function calls. The key to this technique lies in the inclusion of a tracing header in every source file, some clever use of various preprocessor capabilities, and the comma operator.

6.2.1 Defining the Problem

Consider the following functions `main`, `f`, and `g`, and the header they share, each of which is defined in its own source file. The program is not meant to serve any useful purpose other than to call function `f` from several different places (see files co01*.*):

```
#ifndef MSCO01_H
        extern void (f)(void);
        extern void (g)(void);

        #define MSCO01_H
#endif
/* ------------------------------ */
#include <stdio.h>
#include "co01.h"

int main()
{
        f();
        g();
        f();

        return 0;
}
```

```
/* ---------------------------- */
#include <stdio.h>
#include "co01.h"

void (f)()
{
        printf("inside f\n");
}
/* ---------------------------- */
#include "co01.h"

void (g)()
{
        f();
}
```

As the program has multiple, global functions, they are all declared in a header that is included in all the places those functions are called or defined, to ensure those functions are called correctly.

When run, the output produced is:

```
inside f
inside f
inside f
```

Now the redundant parentheses around the function names in their declarations and definitions are quire unusual, and something you are unlikely to come across in the real world. However, they are needed for the solution shown later to work. As such, they'll have to be added. However, once you know of the technique presented, you may wish to put them in all future code; doing so certainly doesn't hurt.

Now we wish to get a traceback of each place we call function `f`, so we can see just what call path we are going down. Perhaps this function sometimes behaves strangely, and we wish to see under what circumstances it is being called. Ideally, we would like to be able to achieve this by making as few changes as possible to the existing code. Why? Having overt debugging code inline almost always makes the code harder to read; it's distracting. Also, it means we must explicitly find all the places where `f` is called, so we can add the trace facility.

6.2.2 Tracing Function Calls

The approach we use is to write a function-like trace macro called `f` such that every call to the function `f` will actually be recognized as a call to our trace macro instead. Our macro will then do any extra work we require, as well as actually calling the user's function `f`.

To make sure we are able to substitute the macro for the function we must make that macro definition available to all source files we create in the project. And if we have no tracing in progress, we can provide a way to not define that macro. For this demonstration the header will be called `co01trace.h`. Because we never know at design time which modules we might wish to debug, and in theory all modules are candidates for debugging, this header should be included in each source file.

Here's the modified version of `co01.h`:

```
#ifndef MSCO01_H
        extern void (f)(void);
        extern void (g)(void);

        #ifdef TRACE
                #include "co01trace.h"
        #endif

        #define MSCO01_H
#endif
```

As such, we can toggle tracing on/off by defining/not defining the macro TRACE during compilation.

And here's the trace header, `co01trace.h`:

```
#ifndef MSCO01TRACE_H
    void traceA(const char *filename, unsigned int line);
    void traceB(const char *filename, unsigned int line);

    #define f() (traceA(__FILE__, __LINE__),    \
        f(),                                    \
        traceB(__FILE__, __LINE__))

    #define MSCO01TRACE_H
#endif
```

The macro `f` expands into calls to `traceA`, `f`, and `traceB`, in that order. Functions `traceA` and `traceB` are called immediately before and after `f`, respectively. We might wish to use only one of the trace functions depending on what it is we are trying to detect. Since the original user's code `f()` is an expression of type `void`, the definition of macro `f` must expand to an expression of the same type, so the replacement is transparent to the user program. Hence the use of the comma operator to paste all three function calls into one large expression. The type of the macro expression is the type of the right-most subexpression, `traceB()`, which is `void`, the same as function `f`. The outer grouping parentheses are present to preserve the very low precedence of the comma operator.

A problem with C preprocessors prior to C89 involves the definition of the macro `f`. Back then, some preprocessors recursively expanded a macro if its definition contained a direct or indirect reference to itself. In this example, macro `f` expands directly to a call to function `f`. According to Standard C, such references do not expand further, thus allowing the approach used here. If, however, our preprocessor recurses here we need to make a minor change, as in:

```
#define f() (traceA(__FILE__, __LINE__),    \
    (f)(),                                  \
    traceB(__FILE__, __LINE__))
```

By adding seemingly redundant parentheses around the `f` in the function call, we have stopped the preprocessor from seeing a call to the function-like macro `f`. Such a call is recognized by an occurrence of the macro name for which the next source token is a left parenthesis. In this case it is not; it is a right parenthesis.

Knowing this, we need to protect the declaration and definition of global function `f` from being seen as calls to our object-like macro `f`, which is why they have enclosing parentheses in the header `co01.h`.

Here are the trace functions, which simply write the calling source filename and line number to `stderr`, and keep track of the number of times they have each been called. They could be made to write the trace to a file or device, to display global variables, or perform other operations as deemed appropriate:

```
#include <stdio.h>

void traceA(const char *filename, unsigned int line)
{
        static unsigned int counter = 0;

        fprintf(stdout, "traceA> %3u: file: %s, line: %u\n",
            ++counter, filename, line);
}
/* ----------------------------- */
void traceB(const char *filename, unsigned int line)
{
        static unsigned int counter = 0;

        fprintf(stdout, "traceB> %3u: file: %s, line: %u\n\n",
            ++counter, filename, line);
}
```

When the program is run with the macro TRACE defined, the output produced is:

```
traceA>   1: file: co01.c, line: 17
inside f
traceB>   1: file: co01.c, line: 17

traceA>   2: file: co01g.c, line: 16
inside f
traceB>   2: file: co01g.c, line: 16

traceA>   3: file: co01.c, line: 19
inside f
traceB>   3: file: co01.c, line: 19
```

6.2.3 Functions with Arguments

The approach described above works for functions having no arguments; however, in reality, functions most often take one or more arguments. Let's add one `int` argument to `f`. Not only can we define the macro to handle that argument, we can trace the exact expression text used as the argument in each call. For example (see files co02*.*):

```
extern void (f)(int);
```



```
/* ---------------------------- */
int main()
{
        int i = 5;

        f(i + 4);
        g();
        f((2 * i * i) + (3 * i) + 4);

        return 0;
}
/* ---------------------------- */
void (f)(int i)
{
        printf("inside f with value %d\n", i);
}
/* ---------------------------- */
void (g)()
{
        f(sizeof(double));
}
```

The output produced is:

```
traceA>   1: file: co02.c, line: 19, argument-text: i + 4
inside f with value 9
traceB> ...

traceA>   2: file: co02g.c, line: 16, argument-text: sizeof(double)
inside f with value 8
traceB> ...

traceA>   3: file: co02.c, line: 21, argument-text: (2 * i * i) + (3 * i) + 4
inside f with value 69
traceB> ...
```

`traceA` displays the actual set of source tokens, after macro expansion, used in each call. The key to this approach lies in the definition of the macro `f`:

```
#ifndef MSCO02TRACE_H
    void traceA(const char *filename, unsigned int line, const char *argText);
    void traceB(const char *filename, unsigned int line, const char *argText);

    #define f(arg) (traceA(__FILE__, __LINE__, #arg),    \
            f(arg),                                      \
            traceB(__FILE__, __LINE__, #arg))

    #define MSCO02TRACE_H
```

```
#endif
```

The unary preprocessor operator # generates a string literal from the source tokens of the actual macro argument. Most importantly, this operator does *not* evaluate its operand! This string is then passed as the new third argument to both trace functions. (For the purposes of this example, `traceB` is unimportant.)

```
void traceA(const char *filename, unsigned int line, const char *argText)
{
    static unsigned int counter = 0;

    fprintf(stdout, "traceA> %3u: file: %s, line: %u, argument-text: %s\n",
        ++counter, filename, line, argText);
}

void traceB(const char *filename, unsigned int line, const char *argText)
{
    static unsigned int counter = 0;

    fprintf(stdout, "traceB> ...\n\n");
}
```

The technique can be extended to handle functions with a any fixed number of arguments. For example:

```
#define f(arg1, arg2)                                   \
    (trace1(__FILE__, __LINE__, #arg1 ", " #arg2),    \
    f(arg1, arg2),                                      \
    trace2(__FILE__, __LINE__, #arg1 ", " #arg2))
```

The trick here is to concatenate the text of all the arguments into one large string literal. This technique requires compiler support for concatenation of adjacent string literals, which C89 introduced.

Exercise 6-4*: While the approach shown above displays the text of the arguments, it does *not* show the value of those arguments, something you'd likely want to know as well. So, how to do that as well? The obvious solution is to extend the trace functions and macro, as follows:

```
void traceA(const char *filename, unsigned int line,
    const char *argText, int argVal);
void traceB(const char *filename, unsigned int line,
    const char *argText, int argVal);

#define f(arg) (traceA(__FILE__, __LINE__, #arg, arg),    \
    f(arg),                                               \
    traceB(__FILE__, __LINE__, #arg, arg))
```

However, this has a major weakness. Consider the following calls to `f`, each of which contains side effects:

```
f(i++ + 2);
f(10 - --i);
```

As defined, the macro would result in the evaluation of each argument three times when it should only be evaluated once. The trace macro must *not* change the meaning of the program. To resolve

this, we need to evaluate the argument once, store that value somewhere, and then use that stored value in the other places. [Hint: How about using a global variable?] (See labs directory lbco01.)

Exercise 6-5*: Thus far, we've ignored the possibility of the function being traced returning a value. Enhance the macro `f` from the exercise above to support this as well. (See labs directory lbco02.)

Exercise 6-6*: C99 added the ability of declaring a function inline. If your compiler supports this, implement a solution to the trace problem that uses an inline function (and which no longer needs the comma operator). (See labs directory lbco05.)

Exercise 6-7*: Can this approach to tracing be applying to a function taking a variable number of arguments, such as `printf`? Certainly, a limited subset of the functionality can be achieved by defining `f` to be an object-like macro. (See labs directory lbco03.) However, if your compiler is C99-compliant, you can define an object-like macro to have a variable number of arguments using the `...` punctuator and the special name `__VA_ARGS__`.. (See labs directory lbco04.)

The first argument is a count of the arguments that follow. In such an approach, the type of this argument determines the maximum number of arguments a function can have. For example, an `unsigned char`, where `char`s have eight bits, limits functions to 255 arguments, which seems like a reasonable number. In this model, the size of each argument is usually required to be the same. This gives rise to the idea of wide and narrow types. For example, why have arguments of types `char`, `short`, and `float` historically been widened to `int`, `int`, and `double`, respectively? Well on some systems, this is necessary to keep objects of certain types aligned.

This calling-argument format works well when all arguments are passed by address, or small-size arguments are passed by value. However, it breaks down when large structures are passed by value. It can also be a problem when a `long int`, a `long long int`, a `double`, or a `long double` is passed by value since the size of these may exceed the size of an argument in this model. As a result, the argument count is no longer a count of logical arguments but, rather, of entries in the argument list. For example, if each argument in the list is four bytes and we pass a 100-byte structure by value, the argument count would be 25, not 1. So, the upper limit of 255 could be reached by passing just one large structure by value.

The model for argument passing usually employed by a C implementation is close to that shown above. However, most often, an argument count is not automatically supplied; if the user wants one they must provide it themselves. The implication of this is that calls such as the following cannot be dealt with in a portable manner:

```
maximum(10, 5, 6, 7)
maximum(2, 65, 876)
maximum(7654, 234, 2374, 3421, 6487)
```

Without a preceding argument count, how can `maximum` determine the number of arguments passed to it? We could reserve some special-valued terminator but that really isn't workable. For example, all `int` values are valid—how would we terminate a variable-length list of `int`s in a call to `maximum`? For pointer argument lists, a null pointer would certainly work, however; but what about lists of arguments having mixed types?

Languages that support calls similar to those of `maximum` typically do so because they have intrinsic functions built-in to the language; they are not really calls to externally compiled routines. As such, the compiler can generate special code behind the scenes to deal with such lists.

Essentially, the C model requires that the programmer pass the argument count as part of the list, either explicitly as a number, or implicitly by something like the conversion specifiers used by `printf` and `scanf`. For example:

```
/*1*/   maximum(4, 10, 5, 6, 7)

/*2*/   printf("%d %f", i, d)
```

In case 1, the first argument, 4, indicates that four more arguments follow. By definition, `maximum` expects all arguments to have the same type; in this case, `int`. In case 2, there are two conversion specifiers indicating that two more arguments follow. Their types are encoded in the specifiers allowing arguments of mixed types to be used.

> **Undefined Behavior:** In a call to a function having a variable number of arguments, if the number and/or type of arguments following the explicit or implicit argument count do not match what was promised.

Therefore, passing a variable number of arguments requires the programmer to check all such calls for correctness since the compiler cannot.

8.2 Implementing a Maximum Function

The following example shows how to use and define a function `maxi`, which returns the maximum of a set of `int` arguments (excluding the count) passed to it (see directory va01):

```
#include <stdio.h>
#include <stdarg.h>
#include <limits.h>

/*1*/   int maxi(int, ...);

int main()
{
/*2*/   printf("-> %d\n", maxi(4, 10, 20, -5, 17));
/*3*/   printf("-> %d\n", maxi(3, -10, -5, -17));

/* is type of second argument int or long? */

/*4*/   printf("-> %d\n", maxi(3, 65432, -5, -17));

/* call maxi with count too small or too large */

/*5*/   printf("-> %d\n", maxi(2, 10, -5, 17));
/*6*/   printf("-> %d\n", maxi(4, 10, -5, 17));

        return 0;
}
```

The output produced on one system in which `int`s are 16 bits and `long`s are 32-bits was:

```
4: 10, 20, -5, 17, -> 20
 3: -10, -5, -17, -> -5
 3: -104, 0, -5, -> 0           /* argument 65432 misinterpreted */
 2: 10, -5, -> 10
 4: 10, -5, 17, 250, -> 250     /* undefined behavior            */
```

The output produced on one system in which `int`s and `long`s are both 32-bits was:

```
4: 10, 20, -5, 17, -> 20
 3: -10, -5, -17, -> -5
 3: 65432, -5, -17, -> 65432    /* argument 65432 handled correctly */
 2: 10, -5, -> 10
 4: 10, -5, 17, 4391048, -> 4391048     /* undefined behavior       */
```

In case 1, in the prototype for `maxi`, the ellipsis punctuator indicates that `maxi` takes a variable number of arguments.

Undefined Behavior: If a function taking a variable number of arguments is called without a prototype containing an ellipsis in scope.

For example, without an ellipsis in scope at the call, the compiler might pass arguments in a way unexpected by the function itself; for example, in registers.

In cases 2 and 3, `maxi` is called correctly as shown by the output produced.

The function call in case 4 is subject to the type of the integer constant 65432. On some systems, it will be `int`, on others, `long int`. In the case of `long int`, the value passed likely will be interpreted in parts, as two or more `int`s, resulting in incorrect output.

Case 5 is quite predictable; since we promised two values, `maxi` takes only the next two arguments resulting in a maximum value of 10. Any excess arguments in the call are evaluated and their values passed in, however; those values are simply not used.

Case 6 results in undefined behavior since we have promised more arguments that we delivered. `maxi` marches to where the fourth argument should have been, typically grabbing some set of bits that it interprets as an `int`. The program could even fail! The worst than can happen is that the pseudo-`int` found has a value less than the others passed explicitly, misleading the programmer into thinking everything is fine when, in fact, there is a serious error laying dormant should that value ever be greater than all the others passed.

Here then is the source for `maxi`:

```
/*7*/ int maxi(int parmn, ...)
{
/*8*/   va_list ap;
        int value, j;
/*9*/   int max = INT_MIN;

/*10*/  va_start(ap, parmn);
        printf("%2d: ", parmn);

/*11*/  for (j = 1; j <= parmn; ++j)
        {
/*12*/          value = va_arg(ap, int);
                printf("%d, ", value);
                if (max < value)
                {
                        max = value;
                }
        }

/*13*/  va_end(ap);
        return max;
}
```

Like the prototype, the definition for `maxi` must also contain the ellipsis, as shown in case 7.

In case 8, we define an object of type `va_list`. This is used internally by a series of macros that we shall use to traverse the argument list passed to `maxi`. Since argument-passing details can vary considerably from one implementation to the next, the format is hidden behind several macros and this type. All Standard C says about `va_list` is it is a type suitable for holding information needed by the macros `va_start`, `va_arg`, `va_end`, and `va_copy`. The name `ap` is often chosen for this variable since it stands for "argument pointer".

In case 9, we seed the function's return value with the smallest possible value an `int` can hold, `INT_MIN`, a macro defined in `limits.h`.

A function taking a variable number of arguments must take at least one. That is, the argument list is made up of a leading fixed part followed by a variable trailing part. In the case of `maxi`, the fixed part consists of one argument, `parmn`. Regardless of how many arguments are in the fixed part, the right-most fixed argument, in this case `parmn`, is most important since this must be given to the macro `va_start`, as shown in case 10. `va_start` gets the program ready to start accessing the variable part of the argument list, whatever that entails. We don't need to know what `ap` is or how it is used, and we shouldn't even care; it works and its use is portable.

Since `maxi` expects each of the arguments in the variable part of the list to have the same type, we can deal with them inside a loop, which we start in case 11. (When writing a function with an interface like `printf`, we would need a set of calls to `va_arg`, one for each expected argument type. For example, `va_arg(ap, double)`, `va_arg(ap, char *)`, and the like.)

In case 12, we pick off the next argument from the variable part of the list using the macro `va_arg`. Clearly, this must be a macro since its second argument is required to be a type, something not possible with a function. (This type typically is used in a rather interesting cast.) The macro expands to a value of the type specified in the second argument and, if the function was called correctly, this value is that of the next argument. This value is displayed simply to monitor the program's progress.

In case 13, once the whole of the variable part of the argument list has been processed, we call `va_end`, which cleans things up internally. Finally, we return the maximum value computed.

The disadvantage of this whole approach is that we must count the number of arguments ourself and pass that count in explicitly. Unfortunately, there is no other solution if we wish to remain portable.

In certain cases, we can avoid using a variable argument list completely even though it appears we need one. For example, we could implement a maximum function by always passing it two arguments, a count and an array of values. However, for this to work the values must be organized directly in an array, and this is not always possible to arrange. Also, this approach is less overt from a notational point of view.

C99 added the `va_copy` macro to all a variable-argument list processing environment to be cloned.

Exercise 8-1*: Write a function called `concatStr` that takes a variable number of arguments: a string count followed by that many strings. It returns a new string that is a concatention of all the string arguments, in order left-to-right. Here are several examples of calling that function:

```
newStr = concatStr(3, "abc", "", "vwxyz");
newStr = concatStr(5, "qq", "123", "ww", "444", "xxxxx");
```

(See labs directory lbva01.)

8.3 The Use of `restrict`

The standard library function `printf` is declared as follows:

```
int printf(const char * restrict format, ...);
```

As we discussed in §2.4, there is generally no point is using `restrict` in a function declaration unless that function has multiple arguments of the same pointer type. In the case of `printf` (and `scanf`), we can make no promises about whether the arguments that might be passed as part of the variable-argument list are aliases format or each other. The best we can do is to promise that `format` is not an alias to anything else.

12. Non-Local Jumps

The C library provides a facility to transfer control directly from one function to a parent function bypassing the usual function-return machinery. In this section, we will see how this very specialized facility can be used to solve real-world problems.

12.1 Introduction

A typical program involves a number of functions. Control is transferred from `main` downward through a hierarchy of functions and back up again. If a problem is encountered inside a function and it cannot be recovered from in that function, we typically return some error status value to its caller. And if that function can't recover, it too returns an error status to its caller, and so on.

While this approach can be made to work, it has several drawbacks. First, we have to write code in each affected function to handle such failures. That makes the program bigger and more complex. Second, that code is executed each time we return from a function call whether an error has occurred or not. That is, we have to check "just in case".

A more efficient approach would be to always assume the called function did its job properly, and in the (hopefully) few cases where it didn't, handle that somehow outside the normal function calling machinery. This makes the code logic simpler and we only pay the price of error handling when an error actually occurs.

The machinery for achieving this is provided in the standard header `setjmp.h` and involves the functions `setjmp` and `longjmp`, and the type `jmp_buf`. Essentially, a program's *context* can be saved into an object of type `jmp_buf` by `setjmp`, and the program can be restored to that context by a subsequent call to `longjmp`. What this amounts to is having a `goto` that can transfer back up the call tree into the middle of another function.[1]

Initially, `setjmp` is called to save the current context of the program. A corresponding call to `longjmp` causes control to return to the function in which `setjmp` was first called. It does this by returning as though `setjmp` had been called a second time. That is, `longjmp` causes an unconditional jump into the routine `setjmp`, which, in turn, returns to its original caller with a return value corresponding to the second argument passed to `longjmp`. When the programmer explicitly calls `setjmp`, it returns a zero value. When `setjmp` returns via an unconditional jump from `longjmp`, it returns a user-defined nonzero value. Note that `longjmp` does not return to its caller. Rather, it returns to the caller of `setjmp`.

12.2 Some Examples

> **Implementation-Defined Behavior:** The actual information that constitutes a current program context might be as little as a few machine registers.

[1] If you despise `goto` you likely won't be very excited about `setjmp` and `longjmp` either. However, each has legitimate uses.

It is imperative that we understand just what is and is not preserved across a `longjmp`. All external and `static` objects are preserved but `auto` and `register` variables are not. Also, even if all machine registers are saved, there is no way of knowing which `auto`s were placed in registers or which `register` storage class objects were not in registers.

> **Undefined Behavior:** The state of all `auto` and `register` objects defined in the function that calls `setjmp` after `longjmp` has returned to that function.

There is one important exception. Any `volatile` automatic object is guaranteed to be intact. Of course, any changes we made to `static`, external, and `volatile` automatic objects between the `setjmp` and `longjmp` calls remain in effect as do any things written to disk, the screen, and other such changes to the runtime environment. Consider the following example (see directory sj01):

```
#include <stdio.h>
#include <setjmp.h>

void test(jmp_buf buffer);

int main()
{
        jmp_buf buffer;
        int i;
        int j = 10;
        register int k = 100;

        i = setjmp(buffer);
        printf("setjmp => %d, j = %d, k = %d\n", i, j, k);

        j += 10;
        k += 20;
        if (i == 0)
        {
                test(buffer);
        }

        return 0;
}

void test(jmp_buf buffer)
{
        longjmp(buffer, 1);
}
```

`buffer` is used to save the program's context. `jmp_buf` is defined in the header in a manner appropriate for the host system. `j` and k are non-`volatile` local variables whose values change in `main`.

`setjmp` is called to save the current program context in the buffer allocated for it. The function returns a zero value, which is displayed. We then call `test`, which in turn, calls `longjmp` to restore the program to the

previously saved state. The value 1 given to `longjmp` will be returned by `setjmp` to `main` and the program will continue execution back in `main`.

When run on four different implementations, the following (different) results were produced. All are permitted:

```
setjmp => 0, j = 10, k = 100
setjmp => 1, j = 20, k = 100

setjmp => 0, j = 10, k = 100
setjmp => 1, j = 20, k = 120

setjmp => 0, j = 10, k = 100
setjmp => 1, j = 10, k = 100

setjmp => 0, j = 10, k = 100
setjmp => 1, j = 79, k = 3440
```

The variables `j` and k are automatic, and most implementations store automatics on a stack. They may also store some of them in machine registers. Remember, using the `register` keyword is a hint that the compiler is free to ignore. Specifically, the presence of `register` does not guarantee a register will be used and the absence of `register` does not prohibit a register from being used; that's the compiler's business. If the variable goes on the stack, its updated value will likely be preserved across the `longjmp`. If it was stored in a register, its new value almost certainly will not be preserved.

In the first set of outputs, it appears that `j` went on the stack and k went in a register. In the second set both went on the stack and in the third set both went into registers. In the fourth case, the "restored" values are weird. Standard C, however, says that the values of such variables are undefined after the restore and that's what we see. Since we generally have no control on how a compiler does such optimization, the program is unreliable if we want/need to count on the restored values being predictable.

By using the `volatile` qualifier, we can force the restored value to be its most recent. That is, to be preserved across the restore. In the case of non-`volatile` automatics, we can remove the initializer and replace it with an assignment statement after the call to `setjmp`. This way we guarantee it will be reinitialized after every save and restore thereby producing a predictable value. So, we can force a variable to have its original or its final value as we need, as follows (see directory sj02):

```
…
int main()
{
        …
        volatile int j = 10;    /* force j to stay changed after restore */
        int k;
        …

        i = setjmp(buffer);
        k = 100;                /* reset k after each restore */
        printf("setjmp => %d, j = %d, k = %d\n", i, j, k);

        …
}
```

The output produced is:

```
setjmp => 0, j = 10, k = 100
setjmp => 1, j = 20, k = 100
```

We can save as many program contexts as we like, and we can restore to any one of them provided we are restoring back up the function call tree that we came down. For example, the following program (see sj03) saves contexts in functions `main` and `testa` and allows `testa` to restore to `main` or, `testb` to restore to either `main` or `testa`.

```
#include <stdio.h>
#include <setjmp.h>

void testa(jmp_buf context);
void testb(jmp_buf context);

static jmp_buf context1;

int main()
{
        jmp_buf context2;
        int i;

        i = setjmp(context2);
        printf("main:  setjmp => %d\n", i);

        testa(context2);

        return 0;
}
```

```
void testa(jmp_buf context)
{
        int i;
        char input[2];

        i = setjmp(context1);
        printf("testa: setjmp => %d\n", i);

        printf("testa: Enter 1 to restore to main: ");
        scanf("%1s", input);

        if (input[0] == '1')
        {
                longjmp(context, 1);
        }

        testb(context);
}

void testb(jmp_buf context)
{
        char input[2];

        printf("testb: Enter 1 to restore to main:\n");
        printf("       Enter 2 to restore to testa: ");
        scanf("%1s", input);

        if (input[0] == '1')
        {
                longjmp(context, 2);
        }
        else if (input[0] == '2')
        {
                longjmp(context1, 2);
        }
}
```

The output produced is:

```
main:  setjmp => 0
testa: setjmp => 0
testa: Enter 1 to restore to main: 1
main:  setjmp => 1
testa: setjmp => 0
testa: Enter 1 to restore to main: <return>
testb: Enter 1 to restore to main:
       Enter 2 to restore to testa: 1
```

```
main:  setjmp => 2
testa: setjmp => 0
testa: Enter 1 to restore to main: <return>
testb: Enter 1 to restore to main:
       Enter 2 to restore to testa: 2
testa: setjmp => 2
testa: Enter 1 to restore to main: <return>
testb: Enter 1 to restore to main:
       Enter 2 to restore to testa: <return>
```

A common use for this recovery machinery is in handling certain kinds of interrupts. For example, assume we have begun computing a set of results on some data and the user discovers the data they entered is incorrect. Rather than wait for the computation to end, which could take many minutes or even hours, the user interrupts the program using something like a CTRL/C (or, on some systems, CTRL/D). The program traps that interrupt and restores the program to the point at which it can prompt for a new set of inputs. That is, it stops processing the old data and throws away the work it has done so far. (It could choose to keep it if that were useful.) In this example (see directory sj04), the "work" we interrupt simply involves displaying a set of integer values:

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>

jmp_buf buffer;

void my_handler(int);

int main()
{
        int j;

        setjmp(buffer);                 /* save context/return from restore */
        signal(SIGINT, my_handler);     /* register interrupt handler */

        printf("\nStarting loop from the beginning\n");

        for (j = 1; j <= 50; ++j)       /* start doing work */
        {
                printf("%d ", j);
        }

        printf("\n");
        return 0;
}
```

```
void my_handler(int arg)                /* Interrupt service function */
{
        signal(SIGINT, SIG_IGN);        /* ignore interrupts of this kind */

        longjmp(buffer, 1);             /* restore context */
}
```

The output produced from several executions of this program were:

```
Starting loop from the beginning
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ^C

Starting loop from the beginning
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 ^C

Starting loop from the beginning
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50
```

As the restore operation bypasses normal function cleanup and return, we must be particularly careful to free up resources we may have allocated. This could involve memory or even devices we may have assigned for our exclusive use. For example, the following program (see directory sj05) allocates memory but that memory isn't freed when a restore occurs. Eventually, we run out of memory:

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

void test(jmp_buf buffer);
```

```
int main()
{
        jmp_buf buffer;
        char *pc;

        setjmp(buffer);

        pc = malloc(30000);
        if (pc == NULL)
        {
                printf("Can't allocate memory\n");
                exit(1);
        }
        else
        {
                printf("Memory allocated\n");
        }

        test(buffer);

        return 0;
}

void test(jmp_buf buffer)
{
        longjmp(buffer, 1);
}
```

The output produced is:

```
Memory allocated
…
Memory allocated
Can't allocate memory
```

The following version (see directory sj06) frees the memory, if necessary, after each restore. Note the use of `volatile` to guarantee that `pc`'s value will be kept intact across the restore. Be careful to apply the type qualifier to the pointer, not to the object to which it points.


```
…
int main()
{
        jmp_buf buffer;
        char *volatile pc = NULL;

        setjmp(buffer);

        if (pc != NULL)
        {
                printf("Freeing old memory\n");
                free(pc);
        }

        pc = malloc(30000);
        …
}
```

The output produced is:

```
Memory allocated
Freeing old memory
…
Memory allocated
Freeing old memory
```

12.3 Program Context

`jmp_buf` is defined as an array of some suitable size to store the current program context, whatever that may be. Since an object of type `jmp_buf` is used only as a storage place for `setjmp` and `longjmp`, programmers should not be concerned about its actual representation. In this respect, it is much like a `FILE` object.

Implementation-Defined Behavior: The mapping of the type type `jmp_buf`.

We must always include `setjmp.h` rather than declaring `jmp_buf` explicitly in our code. For our interest, the following examples show how `jmp_buf` is defined in a number of diverse implementations:

```
typedef int jmp_buf[15];/* VAX running VAX/VMS */
typedef int jmp_buf[9]; /* Intel 8088 running DOS */
typedef int jmp_buf[15];/* Intel 80386 running DOS */
typedef int jmp_buf[32];/* IBM 370 running AIX */
typedef int jmp_buf[16];/* IBM RT */
typedef int jmp_buf[58];/* MC68K */
typedef int jmp_buf[29];/* Intel I860 */
```

```
typedef struct {        /* Intel 80x86 */
        unsigned j_sp;
        unsigned j_ss;
        unsigned j_flag;
        unsigned j_cs;
        unsigned j_ip;
        unsigned j_bp;
        unsigned j_di;
        unsigned j_es;
        unsigned j_si;
        unsigned j_ds;
        unsigned char st1[10], st2[10];
} jmp_buf[1];
```

These definitions indicate a context is simply some or all of the general-purpose register set, including the program counter, stack pointer, and processor status flags. Some also store information about the floating-point processor's state. Note the third case, which involves an array of only one element. C permits such a construct although it is rarely useful.

> **Exercise 12-1:** Inspect your implementation's definition of the type `jmp_buf`. Using the documentation set and any information provided in the `setjmp.h` header, determine just what is saved when `setjmp` is called.

12.4 Miscellaneous Issues

1. If the value of the second argument to `longjmp` is 0, it is ignored and 1 is used instead so that a direct call to `setjmp`, which returns 0, cannot be confused with `longjmp`'s returning through `setjmp` with a value of 0.
2. We should perform a `longjmp` only back up the program call tree, not across its branches. Also, the `longjmp` call must always follow on the same level as, or be below in level to, the `setjmp` call.

> **Undefined Behavior:** If `longjmp` attempts to restore to a context that was never saved by `setjmp`.
>
> **Undefined Behavior:** If `longjmp` attempts to restore to a context and the parent function, which called `setjmp` to save that context initially, has terminated.
>
> **Undefined Behavior:** If `longjmp` is invoked from a nested signal handler.