More about pointers

Timo Karvi

September, 2013

More about Pointers

We study the following topics:

- generic pointers,
- pointer conversions,
- pointer subtraction,
- protecting return values and parameter values: const,
- pointers to functions,
- generic pointers,
- pointers to blocks containing pointers.

Generic Pointers

- A special pointer type void* can be safely converted to any other pointer type.
  Example:

```
void * p;
char c = 'c';
char *cp = &c;

p = cp;

/* illegal putchar(*p); */

putchar(*(char*)p);
```

Pointer Conversions I

- Storage alignment means that storage units must begin on certain addressing boundaries. For example, on a byte-oriented machine, a 16-bit word may have to start on a multiple of bytes, such as 4. In order to satisfy this requirement, compilers insert *pad bytes*. Storage alignment is one of the reasons for the lack of portability.
- Consider the following variable definitions:

  ```
  char * pc;
  int * pi;
  ```

  and assume that pc is initialized to 1001. Given this information, what is the value of (int*)pc?
- The conversion of pc to integer pointer type may require an adjustment to an address that is divisible by 4, accomplished either by scaling down to 1000 or scaling up to 1004. The C language does not specify whether the adjustment is backward or forward, so both possibilities could occur.

Pointer Conversions II

- Thus the value of (int*)pc can differ from the value of pc because of this scaling.
- The other possibility is that no address adjustment is performed when pointers are converted from one type another. In this case, an expression involving the derefencing of that converted pointer, for example *((int*)pc), may result in the operating system aborting the program because of illegal addressing (an integer at an address not divisible by 4).
- In general, pointers to a type S may be safely converted to pointers to a type T and back if S is more restrictive than T.

Pointer Subtraction I

- Given two pointers, p and q, which are of the same type, assuming that p is greater than q and that both point to objects in a single memory block, the expression

$$p - q$$

  yields the number of objects between p and q, including the object pointed to by q.
- The type of the result of pointer difference is *ptrdiff_t*, defined in *stddef.h*.
- The type is defined in the library because the result of pointer subtraction may have to be represented as a "small" value, if the so-called small memory model is used (for example , the model limited to 64K). It may also have to be represented as a large value for large memory models. Therefore, for the sake of portability, you can only assume that this type is signed, and avoid making of conversions.

Pointer Subtraction II

- Pointer subtraction can be used to find the first occurence of the value 0 in a block of doubles. Variable *position* will be initialized to the position in the block that the 0 occurs, or to -1 if 0 does not occur in the block.

```
int position;
for (q=p; q<p+SIZE; q++)
    if (*q == 0.0)
        break;
position = (q == p+SIZE) ? -1 : (q-p)+1;
```

- This code compares q with the pointer pointing to $p + SIZE$, which points beyond the block allocated for p. C allows to do this, as long as you do not try to derefence such a pointer.

Protecting Return Values I

- Sometimes a function can have unwanted side effects. Consider the following function.

```
#define SIZE 10
double *p;
if (MALLOC(p, double, 10))
    error;

double product( double *block, int size) {
   int i;
   for (i=1; i<size; i++)
      block[0] *= block[i];

   return block[0];
}
```

Protecting Return Values II

- The function calculates the product using the first element of the block. (Do not use this kind solutions in your programs!) A call to this function will confuse the caller, who most likely does not expect the function to modify his or her block of memory.
- This side effect can be prevented by adding the *const* keyword:

  ```
  double product ( const double * block, int size);
  ```

  Now *const* specifies that *block* is a pointer to constant data, and any attempt to modify this data would produce a compiler's warning.
- Consider a different example:

Protecting Return Values III

```
const int* f(int i) {
    int * p;
    if ((p = malloc(sizeof(int)) == NULL)
        return NULL;
        *p = i;
        return p;
}
```

f() return a pointer to constant data, and any attempt to modify it will fail. Specifically,

```
int *p = f(2);
```

will produce a compiler warning, while

```
const int * q = f(2);
*q = 3;
```

will produce an error.

Qualified Pointers

Especially, we have the following possibilities with constant pointers:

- `const int *p`; pointer to constant integer, the value of *p* may change, but the value of `*p` can not;
- `int *const p`; constant pointer to integer; the value of `*p` can change, but the value of *p* can not;
- `const int *const p`; constant pointer to constant integer;

There is an alternative syntax for a pointer to constant data:
`int const *p`.
Pointers can also be qualified as `volatile`, mainly to deal with problems encountered with real-time systems. Volatile variables can be modified asynchronously.

Pointers to Functions I

- Consider a type definition

  ```
  int (*fp) (void)
  ```

  Here *fp* is a pointer to an integer function that has no parameters. The brackets around **fp* are necessary. By C's precedence rules,

  ```
  int *fp( )
  ```

  is a function returning a pointer to *int*.
- Another example:

  ```
  double* (*gp)(int);
  ```

  gp is a pointer to a function that returns a pointer to a double and has one integer parameter.
- A pointer to a function determines the prototype of this function, but it does not specify its implementation. You can assign an existing function to the pointer as long as both have identical parameter lists and return types, using this assignment

Pointers to Functions II

```
ptrName = funcName;
```

For example:

```
int (*fp) (void);
double* (*gp) (int);
int f(void);
double* g(int);

fp = f;
gp = g;
```

You can call the function *f()* through the pointer *fp*:

```
int i = fp();
```

- One application of this techniques is to write generic sort functions that make the assumption the user provides the required function, such as the comparison function.

Pointers to Functions III

- Pointers to functions may be used to pass function as parameters to other functions. Suppose we want to write a function *tabulate()* which has a function, say *f()*, as one of its parameters:

```
void tabulate (double low, double high, double step,
               double (*f)(double)) {
double x;

for (x = low; x <= high; x += step)
printf("%13.5f %20.10f\n", x, f(x));
}

tabulate(-1.0, 1.0, 0.01, pol1);
tabulate(-2.0, 2.0, 0.02, pol2);
```

Generic Pointers I

- It may be very useful to have functions with typeless parameters. Consider first a task where there is a block of double values and a function must search a particular value in this block. In this case the function is simple:

```
int search(const double *block, size_t size,
           double value) {
double * p;

if (block == NULL)
return 0;
for (p=block; p < block+size; p++)
if (*p == value)
return 1;
return 0;}
```

Generic Pointers II

- Next we try to generalize this function so that the function can be used with any parameter type: double, int, etc. The result is as follows:

```
int searchGen(const void *block, size_t size,
                   void *value, size_t elsize,
                   int (*compare) (const void *,
                   const void *))
{   void *p;

if (block == NULL)
return 0;
for (p = block; p < block + size*elsize; p += elsize)
if (compare(p, value))
return 1;
return 0; }
```

Pointers to Blocks Containing Pointers I

- In the excises we have defined an arrays whose elements are pointers to some elements. These kind of definitions can also be made by the following way:

  ```
  double **block;
  ```

  Now *block* points to an area in the memory, where every element points to a double value. We can reserve, for example, space for three block element:

  ```
  #define SIZE 3;
  if ((block = calloc(SIZE, sizeof(double*))) == NULL )
   ...
  ```

The next step allocates memory for each element of the block:

Pointers to Blocks Containing Pointers II

```
for (i = 0; i < SIZE; i++)
if ((block[i] = calloc(1, sizeof(double))) == NULL)
   ...
```

If we want to save a double value to the element pointed by *block[0]*, we can use different notations:

```
*(*block) = 2.1; or
*(block[0]) = 2.1; or
block[0][0] = 2.1;

block[1,0] = 3.1; or
*(block[1]) = 3.1; or
*(*(block + 1)) = 3.1;
```

Macros

Any line of C code that begins with # (pound symbol) contains a preprocessing command. These commands can be used to define macros. We start with

- parameterless macros and continue with
- predefined macros,
- macros with parameters.

Parameterless Macros I

Examples:

```
#define PI  3.14
#define SCREEN_W  80
#define GUI  Graphical User \
             Interface
```

The difference between a macro and a constant is the scope: the macro can be used, starting with its definition, until the end of the file (unless it is later undefined, see later). The declaration of a constant is subject to the same scope rules as any other declared variable.
More examples:

```
#define ABORT return EXIT_FAILURE
#define PROMPT  printf("Enter real value: ")
#define SKIP  while (getchar() != '\n');
```

Parameterless Macros II

Macro can be used in the definition of another macro:

```
#define EMPTY    (maxUsed ==  0)

#define ASSERT    if (!(EMPTY ? current == 0 :\
                  0 < current && current <= maxUsed)) { \
                  fprintf(stderr, "invariant failed;\
                    current = %d\t; maxUsed = %d\n",
                    current, maxUsed); \
                   exit(1); }
```

Using macros may be prone to errors. For example

```
#define A 2+4
#define B A*3
```

produces the expression $2 + 4 * 3$. If you want $(2 + 4) * 3$, then define A by

```
define A  (2+4)
```

Predefined Macros I

There are four predefined macros:

`_LINE_`	current line number of the source file
`_FILE_`	name of the current source file
`_TIME_`	time of translation
`_STDC_`	1 if the compiler conforms to ANSI C

Example:

```
if (y = 0) {
fprintf(stderr, "divide by zero error on line
        %d in the file %s\n",
        _LINE_, _FILE_);
 return EXIT_FAILURE;
} else x /= y;
```

Macros with Parameters I

Examples of macros with parameters:

```
#define RANGE(i)        (1 <= (i) && (i) <= maxUsed)
#define R(x)            scanf("%d", &x);
#define READ(c, fileHandle)   (c = fgetc(fileHandle))
```

Take care, if you use macros. Often extra parentheses must be added. If the macro READ is defined without parentheses and the code

```
if (READ(c, fileHandle) == 'x')
```

then the expanded text would result in

```
if (c = fgetc(fileHandle) == 'x')
```

and 0 or 1 would get assigned to c - probably not what was intended.

Undefining Macros

If you must redefine a macro, it is best to undefine it first, using the undef command:

```
#undef PI
```

Building Software with GNU make I

Consider a C-project consisting of several files. First, there is a project directory. It contains a subdirectory src which has files app.c, app.h, main.c, lib.c, bar.c. Executing the following command in the top-level project directory will generate the application named appexp:

```
gcc -o appexp src/main.c src/app.c src/bar.c src/lib.c
```

This command can also be broken into the incremental steps:

```
gcc -c -o main.o src/main.c
gcc -c -o app.o src/app.c
gcc -c -o bar.o src/bar.c
gcc -c -o lib.o src/lib.c
gcc -o appexp main.o app.o bar.o lib.o
```

It is also possible to arrange these commands in a script file. For example, consider buildit script:

Building Software with GNU make II

```
#!/bin/sh
# Build the example project

gcc -c -o main.o src/main.c
gcc -c -o app.o src/app.c
gcc -c -o bar.o src/bar.c
gcc -c -o lib.o src/lib.c
gcc -o appexp main.o app.o bar.o lib.o
```

Then a simple command ./buildit in the command line will build the application. One of the disadvantages of the build script is that it rebuilds the entire project every time it is invoked. One of the major enhancements of the *make* utility over a shell script solution is its capability to understand the dependencies of a project. This allows the *make* utility to rebuild only parts of the project that need updating due to source file changes. A simple Makefile would be in this case:

Building Software with GNU make III

```
appexp: main.o app.o bar.o lib.o
gcc -o appexp main.o app.o bar.o lib.o

main.o : src/main.c src/lib.c src/app.h
gcc -c -o main.o src/main.c

app.o : src/app.c src/lib.h src/app.h
gcc -c -o app.o src/app.c

bar.o : src/bar.c src/lib.h
gcc -c -o bar.o src/bar.c

lib.o : src/lib.c src/lib.h
gcc -c -o lib.o src/lib.c
```

Makefile variables I

GNU make provides support for variables. The basic syntax is

```
MY_VAR = A text string
```

For example, a simple variable Makefile:

```
1:
2:   MY_VAR = file1.c file2.c
3:
4:   all:
5:   echo ${MY_VAR}
6:
```

when the make utility is invoked, it will attempt to build the *all* target and the command on line 5 will be executed. The output is as follows:

Makefile variables II

```

   $ make
   echo file1.c file2.c
   file1.c file2.c

```

It is possible to catenate strings

```
MY_VAR = file.c
MY_VAR += file2.c
```

Using string variable may shorten Makefile, but it can also make it more complicated. Consider the former Makefile with main, app, lib, and bar and construct it now with the help of variables:

Makefile variables III

```

SRC_FILES=main.c app.c bar.c lib.c
OBJ_FILES=$(patsubst %.c, %.o, ${SRC_FILES})

VPATH = src

CFLAGS = -c -g
LDFLAGS = -g

appexp: ${OBJ_FILES}
gcc ${LDFLAGS} -o appexp ${OBJ_FILES}

%.o:%.c
gcc ${CFLAGS} -o $@ $15:
clean:
rm *.o.appexp
```

Makefile variables IV

```
18:
19: MAIN_HDRS=lib.h app.h
20: LIB_HDRS=lib.h
21:
22: main.o : $(addprefix src/, ${MAIN_HDRS})
23: app.o : $(addprefix src/, ${MAIN_HDRS})
24: bar.o : $(addprefix src/, ${LIB_HDRS})
25: lib.o : $(addprefix src/, ${LIB_HDRS})
```

Line 13 introduces a pattern rule to indicate to the make utility how to transform an arbitrary file ending with a.c extension into corresponding file ending in a.o extension.The transformation is accomplished by executing the commands associated with the pattern rule, in this case the command on line 14. Notice that the command on line 14 uses some special variable references to indicate the files that GCC should operate on.

Makefile variables V

The $ variable contains the filename matched for the left side of the rule, and the $ variable contains the filename matched for the right side of the variable.