C/Pointers

Contents

1. Memory and addresses

Memory in a typical modern computer is divided into two classes: a small number of **registers**, which live on the CPU chip and perform specialized functions like keeping track of the location of the next machine code instruction to execute or the current stack frame, and **main memory**, which (mostly) lives outside the CPU chip and which stores the code and data of a running program. When the CPU wants to fetch a value from a particular location in main memory, it must supply an address: a 32-bit or 64-bit unsigned integer on typical current architectures, referring to one of up to 2^{32} or 2^{64} distinct 8-bit locations in the memory. These integers can be manipulated like any other integer; in C, they appear as **pointers**, a family of types that can be passed as arguments, stored in variables, returned from functions, etc.

2. Pointer variables

2.1. Declaring a pointer variable

The convention is C is that the declaration of a complex type looks like its use. To declare a pointer-valued variable, write a declaration for the thing that it points to, but include a * before the variable name:

Toggle line numbers

```
1     int *pointerToInt;
2     double *pointerToDouble;
3     char *pointerToChar;
4     char **pointerToPointerToChar;
```

2.2. Assigning to pointer variables

Declaring a pointer-valued variable allocates space to hold the pointer but *not* to hold anything it points to. Like any other variable in C, a pointer-valued variable will initially contain garbage---in this case, the address of a location that might or might not contain something important. To initialize a pointer variable, you have to assign to it the address of something that already exists. Typically this is done using the & (**address-of**) operator:

Toggle line numbers

```
1     int n;              /* an int variable */
2     int *p;             /* a pointer to an int */
3
4     p = &n;             /* p now points to n */
```

2.3. Using a pointer

Pointer variables can be used in two ways: to get their value (a pointer), e.g. if you want to assign an address to more than one pointer variable:

Toggle line numbers

```
1     int n;              /* an int variable */
2     int *p;             /* a pointer to an int */
3     int *q;             /* another pointer to an int */
4
5     p = &n;             /* p now points to n */
6     q = p;              /* q now points to n as well */
```

But more often you will want to work on the value stored at the location pointed to. You can do this by using the * (**dereference**) operator, which acts as an inverse of the address-of operator:

Toggle line numbers

```
1     int n;              /* an int variable */
2     int *p;             /* a pointer to an int */
3
4     p = &n;             /* p now points to n */
5
```

```
    *p = 2;             /* sets n to 2 */
    *p = *p + *p;       /* sets n to 4 */
```

The * operator binds very tightly, so you can usually use `*p` anywhere you could use the variable it points to without worrying about parentheses. However, a few operators, such as `--`, `++`, and `.` (used in C/Structs) bind tighter, requiring parantheses if you want the * to take precedence.

Toggle line numbers

```
1     (*p)++;             /* increment the value pointed to by p */
2     *p++;               /* WARNING: increments p itself */
```

2.4. Printing pointers

You can print a pointer value using `printf` with the `%p` format specifier. To do so, you should convert the pointer to type `void *` first using a cast (see below for `void *` pointers), although on machines that don't have different representations for different pointer types, this may not be necessary.
Here is a short program that prints out some pointer values:

Toggle line numbers

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int G = 0;   /* a global variable, stored in BSS segment */
 5
 6 int
 7 main(int argc, char **argv)
 8 {
 9     static int s;  /* static local variable, stored in BSS segment */
10     int a;         /* automatic variable, stored on stack */
11     int *p;        /* pointer variable for malloc below */
12
13     /* obtain a block big enough for one int from the heap */
14     p = malloc(sizeof(int));
15
16     printf("&G   = %p\n", (void *) &G);
17     printf("&s   = %p\n", (void *) &s);
18     printf("&a   = %p\n", (void *) &a);
19     printf("&p   = %p\n", (void *) &p);
20     printf("p    = %p\n", (void *) p);
21     printf("main = %p\n", (void *) main);
22
23     free(p);
24
25     return 0;
26 }
```

looking_at_pointers.c

When I run this on a Mac OS X 10.6 machine after compiling with `gcc`, the output is:

```
&G   = 0x100001078
&s   = 0x10000107c
&a   = 0x7fff5fbff2bc
&p   = 0x7fff5fbff2b0
p    = 0x100100080
main = 0x100000e18
```

The interesting thing here is that we can see how the compiler chooses to allocate space for variables based on their storage classes. The global variable `G` and the static local variable `s` both persist between function calls, so they get placed in the BSS segment (see WikiPedia: .bss) that starts somewhere around `0x100000000`, typically after the code segment containing the actual code of the program. Local variables `a` and `p` are allocated on the stack, which grows down from somewhere near the top of the address space. The block that `malloc` returns that `p` points to is allocated off the heap, a region of memory that may also grow over time and starts after the BSS segment. Finally, `main` appears at 0x100000e18; this is in the code segment, which is a bit lower in memory than all the global variables.

3. The null pointer

The special value `0`, known as the **null pointer** may be assigned to a pointer of any type. It may or may not be represented by the actual address `0`, but it will act like `0` in all contexts (e.g., it has the value false in an `if` or `while` statement). Null pointers are often used to indicate missing data or failed functions. Attempting to dereference a null pointer can have catastrophic effects, so it's important to be aware of when you might be supplied with one.

4. Pointers and functions

A simple application of pointers is to get around C's limit on having only one return value from a function. Because C arguments are copied, assigning a value to an argument inside a function has no effect on the outside. So the `doubler` function below doesn't do much:

Toggle line numbers

```
 1 #include <stdio.h>
 2
 3 /* doesn't work */
 4 void
 5 doubler(int x)
 6 {
 7     x *= 2;
 8 }
 9
10 int
11 main(int argc, char **argv)
12 {
13     int y;
```

```
14
15      y = 1;
16
17      doubler(y);                 /* no effect on y */
18
19      printf("%d\n", y);          /* prints 1 */
20
21      return 0;
22  }
```

bad_doubler.c

However, if instead of passing the value of `y` into `doubler` we pass a pointer to `y`, then the `doubler` function can reach out of its own stack frame to manipulate `y` itself:

Toggle line numbers

```
1  #include <stdio.h>
2
3  /* doesn't work */
4  void
5  doubler(int *x)
6  {
7      *x *= 2;
8  }
9
10 int
11 main(int argc, char **argv)
12 {
13     int y;
14
15     y = 1;
16
17     doubler(&y);                /* sets y to 2 */
18
19     printf("%d\n", y);          /* prints 2 */
20
21     return 0;
22 }
```

good_doubler.c

Generally, if you pass the value of a variable into a function (with no &), you can be assured that the function can't modify your original variable. When you pass a pointer, you should assume that the function can and will change the variable's value. If you want to write a function that takes a pointer argument but promises not to modify the target of the pointer, use `const`, like this:

Toggle line numbers

```
1 void
2 printPointerTarget(const int *p)
3 {
4     printf("%d\n", *p);
5 }
```

The `const` qualifier tells the compiler that the target of the pointer shouldn't be modified. This will cause it to return an error if you try to assign to it anyway:

Toggle line numbers

```
1 void
2 printPointerTarget(const int *p)
3 {
4     *p = 5;  /* produces compile-time error */
5     printf("%d\n", *p);
6 }
```

Passing `const` pointers is mostly used when passing large structures to functions, where copying a 32-bit pointer is cheaper than copying the thing it points to.
If you really want to modify the target anyway, C lets you "cast away `const`":

Toggle line numbers

```
1 void
2 printPointerTarget(const int *p)
3 {
4     *((int *) p) = 5;  /* produces compile-time error */
5     printf("%d\n", *p);
6 }
```

There is usually no good reason to do this; the one exception might be if the target of the pointer represents an AbstractDataType, and you want to modify its representation during some operation to optimize things somehow in a way that will not be visible outside the abstraction barrier, making it appear to leave the target constant.
Note that while it is safe to pass pointers down into functions, it is very dangerous to pass pointers up. The reason is that the space used to hold any local variable of the function will be reclaimed when the function exits, but the pointer will still point to the same location, *even though something else may now be stored there*. So this function is very dangerous:

Toggle line numbers

```
1 int *
2 dangerous(void)
3 {
4     int n;
5
6     return &n;          /* NO! */
```

```
 7 }
 8
 9 ...
10
11     *dangerous() = 12;  /* writes 12 to some unknown location */
```

An exception is when you can guarantee that the location pointed to will survive even after the function exits, e.g. when the location is dynamically allocated using `malloc` (see below) or when the local variable is declared `static`:

Toggle line numbers

```
 1 int *
 2 returnStatic(void)
 3 {
 4     static int n;
 5
 6     return &n;
 7 }
 8
 9 ...
10
11     *returnStatic() = 12;       /* writes 12 to the hidden static variable */
```

Usually returning a pointer to a `static` local variable is not good practice, since the point of making a variable local is to keep outsiders from getting at it. If you find yourself tempted to do this, a better approach is to allocate a new block using `malloc` (see below) and return a pointer to that. The downside of the `malloc` method is that the caller has to promise to call `free` on the block later, or you will get a storage leak.

5. Pointer arithmetic and arrays

Because pointers are just numerical values, one can do arithmetic on them. Specifically, it is permitted to

Add an integer to a pointer or subtract an integer from a pointer. The effect of `p+n` where `p` is a pointer and `n` is an integer is to compute the address equal to `p` plus `n` times the size of whatever `p` points to (this is why `int *` pointers and `char *` pointers aren't the same).

Subtract one pointer from another. The two pointers must have the same type (e.g. both `int *` or both `char *`). The result is an integer value, equal to the numerical difference between the addresses divided by the size of the objects pointed to.

Compare two pointers using `==`, `!=`, `<`, `>`, `<=`, or `>=`.

Increment or decrement a pointer using `++` or `--`.

The main application of pointer arithmetic in C is in **arrays**. An array is a block of memory that holds one or more objects of a given type. It is declared by giving the type of object the array holds followed by the array name and the size in square brackets:

Toggle line numbers

```
1     int a[50];          /* array of 50 ints */
2     char *cp[100];      /* array of 100 pointers to char */
```

Declaring an array allocates enough space to hold the specified number of objects (e.g. 200 bytes for `a` above and 400 for `cp`---note that a `char *` is an address, so it is much bigger than a `char`). The number inside the square brackets must be a constant whose value can be determined at compile time.
The array name acts like a constant pointer to the zeroth element of the array. It is thus possible to set or read the zeroth element using `*a`. But because the array name is constant, you can't assign to it:

Toggle line numbers

```
1     *a = 12;            /* sets zeroth element to 12 */
2
3     a = &n;             /* #### DOESN'T WORK #### */
```

More common is to use square brackets to refer to a particular element of the array. The expression `a[n]` is defined to be equivalent to `*(a+n)`; the **index** `n` (an integer) is added to the base of the array (a pointer), to get to the location of the `n`-th element of `a`. The implicit `*` then dereferences this location so that you can read its value (in a normal expression) or assign to it (on the left-hand side of an assignment operator). The effect is to allow you to use `a[n]` just as you would any other variable of type `int` (or whatever type `a` was declared as).
Note that C doesn't do any sort of bounds checking. Given the declaration `int a[50];`, only indices from `a[0]` to `a[49]` can be used safely. However, the compiler will not blink at `a[-12]` or `a[10000]`. If you read from such a location you will get garbage data; if you write to it, you will overwrite god-knows-what, possibly trashing some other variable somewhere else in your program or some critical part of the stack (like the location to jump to when you return from a function). It is up to you as a programmer to avoid such **buffer overruns**, which can lead to very mysterious (and in the case of code that gets input from a network, security-damaging) bugs. The `valgrind` program can help detect such overruns in some cases (see C/valgrind).
Another curious feature of the definition of `a[n]` as identical to `*(a+n)` is that it doesn't actually matter which of the array name or the index goes inside the braces. So all of `a[0]`, `*a`, and `0[a]` refer to the zeroth entry in `a`. Unless you are deliberately trying to obfuscate your code, it's best to write what you mean.

5.1. Arrays and functions

Because array names act like pointers, they can be passed into functions that expect pointers as their arguments. For example, here is a function that computes the sum of all the values in an array `a` of size `n`:

Toggle line numbers

```
1 /* return the sum of the values in a, an array of size n */
2 int
3 sumArray(int in, const int *a)
4 {
5     int i;
6     int sum;
7
8     sum = 0;
9     for(i = 0; i < n; i++) {
10        sum += a[i];
11    }
```

```
12
13      return sum;
14  }
```

Note the use of `const` to promise that `sumArray` won't modify the contents of `a`.
Another way to write the function header is to declare `a` as an array of unknown size:

Toggle line numbers

```
1  /* return the sum of the values in a, an array of size n */
2  int
3  sumArray(int n, const int a[])
4  {
5      ...
6  }
```

This has *exactly* the same meaning to the compiler as the previous definition. Even though normally the declarations `int a[10]` and `int *a` mean very different things (the first one allocates space to hold 10 `int`s, and prevents assigning a new value to `a`), in a function argument `int a[]` is just SyntacticSugar for `int *a`. You can even modify what `a` points to inside `sumArray` by assigning to it. This will allow you to do things that you usually don't want to do, like write this hideous routine:

Toggle line numbers

```
1  /* return the sum of the values in a, an array of size n */
2  int
3  sumArray(int n, const int a[])
4  {
5      const int *an;      /* pointer to first element not in a */
6      int sum;
7
8      sum = 0;
9      an = a+n;
10
11     while(a < an) {
12         sum += *a++;
13     }
14
15     return sum;
16 }
```

5.2. Multidimensional arrays

Arrays can themselves be members of arrays. The result is a multidimensional array, where a value in row `i` and column `j` is accessed by `a[i][j]`.
Declaration is similar to one-dimensional arrays:

Toggle line numbers

```
1 int a[6][4];    /* declares an array of 6 rows of 4 ints each */
```

This declaration produces an array of 24 `int` values, packed contiguously in memory. The interpretation is that `a` is an array of 6 objects, each of which is an array of 4 `int`s.
If we imagine the array to contain increasing values like this:

```
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17
```

the actual positions in memory will look like this:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
 ^                 ^                 ^
a[0]              a[1]              a[2]
```

To look up a value, we do the usual array-indexing magic. Suppose we want to find `a[1][4]`. The name `a` acts as a pointer to the base of the array.The name `a[1]` says to skip ahead 1 times the size of the things pointed to by `a`, which are arrays of 6 `int`s each, for a total size of 24 bytes assuming 4-byte `int`s. For `a[1][4]`, we start at `a[1]` and move forward 4 times the size of the thing pointed to by `a[1]`, which is an `int`; this puts us 24+16 bytes from `a`, the position of 10 in the picture above.
Like other array declarations, the size must be specified at compile time in pre-C99 C. If this is not desirable, a similar effect can be obtained by allocating each row separately using `malloc` and building a master list of pointers to rows, of type `int **`. The downside of this approach is that the array is no longer contiguous (which may affect cache performance) and it requires reading a pointer to find the location of a particular value, instead of just doing address arithmetic starting from the base address of the array. But elements can still be accessed using the `a[i][j]` syntax. An example of this approach is given in malloc2d.c.

5.3. Variable-length arrays

C99 adds the feature of **variable-length arrays**, where the size of the array is determined at run-time. These can only appear as local variables in procedures (*automatic variables*) or in argument lists. In the case of variable-length arrays in argument lists, it is also necessary that the length of the array be computable from previous arguments.
For example, we could make the length of the array explicit in our `sumArray` function:

Toggle line numbers

```
1 /* return the sum of the values in a, an array of size n */
2 int
3 sumArray(int n, const int a[n])
4 {
5     int i;
6     int sum;
7
8     sum = 0;
9     for(i = 0; i < n; i++) {
```

```
10          sum += a[i];
11      }
12
13      return sum;
14  }
```

This doesn't accompish much, because the length of the array is not used. However, it does become useful if we have a two-dimensional array, as otherwise there is no way to compute the length of each row:

Toggle line numbers

```
 1  int
 2  sumMatrix(int rows, int cols, const int m[rows][cols])
 3  {
 4      int i;
 5      int j;
 6      int sum;
 7
 8      sum = 0;
 9      for(i = 0; i < rows; i++) {
10          for(j = 0; j < cols; j++) {
11              sum += a[i][j];
12          }
13      }
14
15      return sum;
16  }
```

Here the fact that each row of `m` is known to be an array of `cols` many `int`s makes the implicit pointer computation in `a[i][j]` actually work. It is considerably more difficult to to this in ANSI C; the simplest approach is to pack `m` into a one-dimensional array and do the address computation explicitly:

Toggle line numbers

```
 1  int
 2  sumMatrix(int rows, int cols, const int a[])
 3  {
 4      int i;
 5      int j;
 6      int sum;
 7
 8      sum = 0;
 9      for(i = 0; i < rows; i++) {
10          for(j = 0; j < cols; j++) {
11              sum += a[i*cols + j];
12          }
13      }
14
```

```
15      return sum;
16  }
```

Variable-length arrays can sometimes be used for run-time storage allocation, as an alternative to `malloc` and `free` (see below). A variable-length array allocated as a local variable will be deallocated when the containing scope (usually a function body, but maybe just a compound statement marked off by braces) exits. One consequence of this is that you can't return a variable-length array from a function.
Here is an example of code using this feature:

Toggle line numbers

```
 1 /* reverse an array in place */
 2 void
 3 reverseArray(int n, int a[n])
 4 {
 5     /* algorithm: copy to a new array in reverse order */
 6     /* then copy back */
 7
 8     int i;
 9     int copy[n];
10
11     for(i = 0; i < n; i++) {
12         /* the -1 is needed to that a[0] goes to a[n-1] etc. */
13         copy[n-i-1] = a[i];
14     }
15
16     for(i = 0; i < n; i++) {
17         a[i] = copy[i];
18     }
19 }
```

While using variable-length arrays for this purpose can simplify code in some cases, as a general programming practice it is **extremely dangerous**. The reason is that, unlike allocations through `malloc`, variable-length array allocations are typically allocated on the stack (which is often more constrainted than the heap) and have no way of reporting failure. So if there isn't enough room for your variable-length array, odds are you won't find out until a segmentation fault occurs somewhere later in your code when you try to use it.
(As an additional annoyance, `gdb` is confused by two-dimensional variable-length arrays.)
Here's a safer version of the above routine, using `malloc` and `free`.

Toggle line numbers

```
1 /* reverse an array in place */
2 void
3 reverseArray(int n, int a[n])
4 {
5     /* algorithm: copy to a new array in reverse order */
6     /* then copy back */
7
8     int i;
```

```
 9     int *copy;
10
11     copy = (int *) malloc(n * sizeof(int));
12     assert(copy);  /* or some other error check */
13
14     for(i = 0; i < n; i++) {
15         /* the -1 is needed to that a[0] goes to a[n-1] etc. */
16         copy[n-i-1] = a[i];
17     }
18
19     for(i = 0; i < n; i++) {
20         a[i] = copy[i];
21     }
22
23     free(copy);
24 }
```

6. Void pointers

A special pointer type is `void *`, a "pointer to `void`". Such pointers are declared in the usual way:

Toggle line numbers

```
1     void *nothing;      /* pointer to nothing */
```

Unlike ordinary pointers, you can't derefence a `void *` pointer or do arithmetic on it, because the compiler doesn't know what type it points to. However, you are allowed to use a `void *` as a kind of "raw address" pointer value that you can store arbitrary pointers in. It is permitted to assign to a `void *` variable from an expression of any pointer type; conversely, a `void *` pointer value can be assigned to a pointer variable of any type.
If you need to use a `void *` pointer as a pointer of a particular type in an expression, you can **cast** it to the appropriate type by prefixing it with a type name in parentheses, like this:

Toggle line numbers

```
1     int a[50];          /* typical array of ints */
2     void *p;            /* dangerous void pointer */
3
4     a[12] = 17;         /* save that valuable 17 */
5     p = a;              /* p now holds base address of a */
6
7     printf("%d\n", ((int *) p)[12]);  /* get 17 back */
```

Usually if you have to start writing casts, it's a sign that you are doing something wrong, and you run the danger of **violating the type system**---say, by tricking the compiler into treating a block of bits that are supposed to be an `int` as four `char`s. But violating the type system like this will be necessary for

some applications, because even the weak type system in C turns out to be too restrictive for writing certain kinds of "generic" code that work on values of arbitrary types.

7. Run-time storage allocation

C does not permit arrays to be declared with variable sizes. C also doesn't let local variables outlive the function they are declared in. Both features can be awkward if you want to build data structures at run time that have unpredictable (perhaps even changing) sizes and that are intended to persist longer than the functions that create them. To build such structures, the standard C library provides the `malloc` routine, which asks the operating system for a block of space of a given size (in bytes). With a bit of pushing and shoving, this can be used to obtain a block of space that for all practical purposes acts just like an array.
To use `malloc`, you must include `stdlib.h` at the top of your program. The declaration for `malloc` is

Toggle line numbers

```
1 void *malloc(size_t);
```

where `size_t` is an integer type (often `unsigned long`). Calling `malloc` with an argument of `n` allocates and returns a pointer to the start of a block of `n` bytes if possible. If the system can't give you the space you asked for (maybe you asked for more space than it has), `malloc` returns a null pointer. It is good practice to test the return value of `malloc` whenever you call it.
Because the return type of `malloc` is `void *`, its return value can be assigned to any variable with a pointer type. Computing the size of the block you need is your responsibility---and you will be punished for any mistakes with difficult-to-diagnose buffer overrun errors---but this task is made slightly easier by the built-in `sizeof` operator that allows you to compute the size in bytes of any particular data type. A typical call to `malloc` might thus look something like this:

Toggle line numbers

```
 1 #include <stdlib.h>
 2
 3 /* allocate and return a new integer array with n elements */
 4 /* calls abort() if there isn't enough space */
 5 int *
 6 makeIntArray(int n)
 7 {
 8     int *a;
 9
10     a = malloc(sizeof(int) * n);
11
12     if(a == 0) abort();                /* die on failure */
13
14     return a;
15 }
```

When you are done with a `malloc`'d region, you should return the space to the system using the `free` routine, also defined in `stdlib.h`. If you don't do this, your program will quickly run out of space. The `free` routine takes a `void *` as its argument and returns nothing. It is good practice to write a matching **destructor** that de-allocates an object for each **constructor** (like `makeIntArray`) that makes one.

Toggle line numbers

```
1 void
2 destroyIntArray(int *a)
3 {
4     free(a);
5 }
```

It is a serious error to do anything at all with a block after it has been `free`d.
It is also possible to grow or shrink a previously allocated block. This is done using the `realloc` function, which is declared as

Toggle line numbers

```
1 void *realloc(void *oldBlock, size_t newSize);
```

The `realloc` function returns a pointer to the resized block. It may or may not allocate a new block; if there is room, it may leave the old block in place and return its argument. But it may allocate a new block and copy the contents of the old block, so you should assume that the old pointer has been `free`d. Here's a typical use of `realloc` to build an array that grows as large as it needs to be:

Toggle line numbers

```
 1 /* read numbers from stdin until there aren't any more */
 2 /* returns an array of all numbers read, or null on error */
 3 /* returns the count of numbers read in *count */
 4 int *
 5 readNumbers(int *count /* RETVAL */)
 6 {
 7     int mycount;        /* number of numbers read */
 8     int size;           /* size of block allocated so far */
 9     int *a;             /* block */
10     int n;              /* number read */
11
12     mycount = 0;
13     size = 1;
14
15     a = malloc(sizeof(int) * size);     /* allocating zero bytes is tricky */
16     if(a == 0) return 0;
17
18     while(scanf("%d", &n) == 1) {
19         /* is there room? */
20         while(mycount >= size) {
21             /* double the size to avoid calling realloc for every number read */
22             size *= 2;
23             a = realloc(a, sizeof(int) * size);
```

```
24              if(a == 0) return 0;
25          }
26
27          /* put the new number in */
28          a[mycount++] = n;
29      }
30
31      /* now trim off any excess space */
32      a = realloc(a, sizeof(int) * mycount);
33      /* note: if a == 0 at this point we'll just return it anyway */
34
35      /* save out mycount */
36      *count = mycount;
37
38      return a;
39  }
```

Because errors involving `malloc` and its friends can be very difficult to spot, it is recommended to test any program that uses `malloc` using `valgrind` if possible. (See C/valgrind).
(See also C/DynamicStorageAllocation for some old notes on this subject.)

8. The restrict keyword

In C99, it is possible to declare that a pointer variable is the only way to reach its target as long as it is in scope. This is not enforced by the compiler; instead, it is a promise from the programmer *to* the compiler that any data reached through this point will not be changed by other parts of the code, which allows the compiler to optimize code in ways that are not possible if pointers might point to the same place (a phenomenon called **pointer aliasing**). For example, in the short routine:

Toggle line numbers

```
1 // write 1 + *src to *dst and return *src
2 int
3 copyPlusOne(int * restrict dst, int * restrict src)
4 {
5     *dst = *src + 1;
6     return *src;
7 }
```

the output of `gcc -std=c99 -O3 -S` includes one more instruction if the `restrict` qualifiers are removed. The reason is that if `dst` and `src` may point to the same location, `src` needs to be re-read for the `return` statement, in case it changed, but if they don't, the compiler can re-use the previous value it already has in one of the CPU registers.
For most code, this feature is useless, and potentially dangerous if someone calls your routine with aliased pointers. However, it may sometimes be possible to increase performance of time-critical code by adding a `restrict` keyword. The cost is that the code might no longer work if called with aliased pointers.

CategoryProgrammingNotes

2014-06-17 11:57