



QACADV\_v1.0



# Advanced C

## 05 Pointers



transforming performance  
through learning

## Pointers

- **Objectives**
  - Gain a sound understanding of advanced pointer techniques
- **Contents**
  - Declarations
  - The "Level of Indirection"
  - Pointer arithmetic
  - Untyped pointers
  - Function pointers
- **Practical**
  - Practice function pointers and pointer arithmetic
- **Summary**



Pointers are one of the most powerful yet potentially dangerous features of the C language. A sound appreciation of pointer techniques is necessary in order to get the most from the language.

In this chapter, we shall start by reviewing pointer notation. We will then discuss pointer arithmetic, highlighting the valid operations that can be applied to pointers as we do so.

There are occasions when it is desirable to declare a *generic* pointer, that is a pointer that can hold the address of any kind of data object. For example, the `fread()` function, which reads data from a file into memory, needs to be passed a generic pointer value that specifies the address of the destination buffer in memory.

Function pointers are also extremely useful, although the syntax can be quite off-putting to the uninitiated! We shall review the syntax and then look at a number of examples which show the true power of function pointers.

The chapter concludes with an advanced discussion on pointer internals.

## Pointer declarations

- In C, pointers are declared using the `*` type modifier, which denotes an address container

```
int i;    /* i is an integer */
int *p;   /* p is a pointer to an integer */
```

*modifies the type of p*

- This notation is unfortunate since `*` is also the "contents of" operator:

```
int i;
int *p = &i;
*p = &i; ❌
```

*\* here is not the same as \* here*

C's notation for declaring pointers is rather unfortunate, since the `*` operator means two different things in the two different places it is used.

In a declaration, `*` modifies the type of the objects being declared. `int p` declares `p` as an integer, whereas `int *p` declares `p` as a pointer to an integer. Any number of `*` operators may be specified, as in `int ****p` which declares `p` as a *pointer to a pointer to a pointer to a pointer to an integer*.

The declaration `int *p = &i` reads "declare `p` as a pointer to an integer and initialise with the address of the variable `i`". It is very important to realise this is two statements combined.

```
int i;
int *p = &i;
```

is equivalent to:

```
int i;
int *p; p = &i;
```

but quite definitely not:

```
int i;
int *p; *p = &i; (which would not compile)
```

The second (incorrect) version would not compile since `p` is a pointer to an integer, the compiler would attempt to place into an integer (the one `p` points to) the address of an integer. Since integers cannot contain addresses, a type mismatch occurs and the compiler produces a warning message.

## Pointer declaration problems

- To bypass this confusion, some programmers move the \* type-modifier towards the type:

```
int i;  
int* p = &i;  
  
p = &i;
```



- Although solving one problem, it can be rather misleading:

```
int* p, q;
```

- The \* neither belongs with the type nor with the variable name - some programmers reflect this as follows:

```
int * p;
```

Confusion arises because a declaration `int *p = &i` is valid, yet the code `*p = &i` is invalid. A possible solution is to make the declaration and the code look alike. This can be done by moving the \* type-modifier towards the type (since the compiler ignores all white space, the \* may be placed anywhere).

Doing this, however, is quite misleading. The declaration `int* p, q` suggests that *both* p and q are pointers to integers, whereas in reality q has been declared as an integer. One way to solve this problem is to declare only one variable per line.

Some programmers move the \* type-modifier so it is neither adjacent to the type, nor to the variable. This is to reflect the fact that the \* is neither part of the type nor part of the variable.

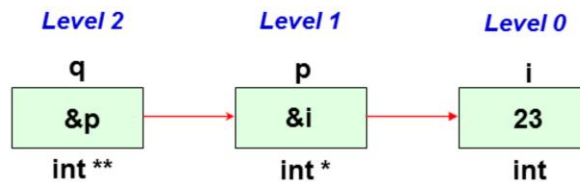
Which method is best? Kernighan and Ritchie in their famous (or infamous?) book "The C Programming Language" place the \* next to the variable. Generations of C programmers read the book and emulated the style.

Bjarne Stroustrup in his book "The C++ Programming Language" favours the placement of the \* next to the type. Perhaps this will cause generations of C++ programmers to do the same.

## Levels of indirection

- A favourite compiler error message is:  
*different levels of indirection*
- The *level* of a pointer can be thought of as the number of hops that must be made before the *real* data is reached

```
int i = 23;  
int * p = &i;  
int ** q = &p;
```



The Microsoft compiler tends to spit out the error message "different levels of indirection" whenever something is wrong with code involving pointers. However, what *are* these levels of indirection?

The "level" of a pointer is the distance it is from the data it eventually refers to. Thus a pointer to an integer is one level removed from the integer it refers to. A pointer to a pointer to an integer is two levels removed from the data it refers to.

We can see from this that there is an exact correspondence between the number of "\*"s in a declaration and the "level" of the pointer.

## Predicting levels

- For this idea to be useful, we must be able to explain and understand everything going on in this code



```
int i, j;  
int * p;  
int ** q;  
int a[10];  
date d;  
date * r;
```

```
p = &i;  
j = *p;  
q = &p;  
j = **q;
```

```
p = a;  
i = a[3];
```

```
r = &d;  
j = r->day;
```

Clearly, this idea of the "level" of a pointer is going to be useless if we end up having to guess at the level of a particular pointer or piece of data.

Just as there is a correspondence between the number of \*'s used and the level of the pointer, so other operators, [ ], & and -> have their effects on the level of pointers.

## The rules

- **A variable starts at level 0**
- **In declarations:**
  - \* increases the level (by 1)
  - [] increases the level
- **In code:**
  - \* decreases the level
  - [] decreases the level
  - & increases the level
  - > decreases the level

```
int * pointer;  
int array[42];
```

```
*pointer = 42;  
array[24] = 42;  
pointer = &array[24];  
date_ptr->day = 24;
```

Examining the code (which compiles cleanly) on the previous foil helps us to deduce the rules above. It is necessary to draw a distinction between the declarations and the code itself. We have already seen this in:

```
int i;  
int *p = &i;  
  
*p = i;
```

In the declaration of the variable "p", the \* modifies the type from integer to pointer to integer. Thus the level rises from zero to one. In the code, however, the \* dereferences the pointer, "hopping" from the pointer to the data. Thus the level falls from one to zero.

### Exercise: what levels are these?

```
int * p;  
int ** p;  
int * a[10];  
int ** b[10];  
int * c[10][10];  
int (*pa)[10];
```



A quick rule of thumb when declaring variables is to remember that only `*` and `[ ]` may be used. Nothing else is valid (in C anyway). These two operators increase the level of the variable. All that needs to be done is to count up their numbers and this gives the level of the variable.



## Typed pointers

- **Pointers are bound to a particular type for three reasons:**

- To keep track of how many valid *bytes* of data are at the end of the pointer
- To keep track of the *format* of the data at the end of the pointer (i.e. IEEE format for floats, etc.)
- To *scale* the pointer by the correct amount when an arithmetic operation is performed on it



In C, pointer variables are declared as *pointers to certain types of data*. Pointers are said to be *bound* to a type.

If a pointer to an `int` is declared, it must have the address of an `int` assigned to it. If the address of a `short int`, `long int`, `float`, `double` or any other type is assigned to the pointer, the compiler issues an error message.

An integer pointer containing the address of a `double` would give the compiler some problems. Firstly, there would be `sizeof(double)` bytes of valid data at the end of the pointer, rather than the `sizeof(int)` bytes the compiler would expect.

Secondly, the data at the end of the pointer would probably be in IEEE format, which the compiler would not expect from the nature of the pointer.

Finally, if the pointer is incremented by 1, the compiler scales the addition by the size of an integer. The pointer moves into the second half of the `double` (depending on its size), not beyond the data as the programmer might expect.

## Examples

```
int    i;  
short s = 27,    * sp = &s;  
long   l = 0L,   * lp = &l;  
float  f = 4.3F, * fp = &f;
```

```
l = *sp;
```

**Compiler knows `sizeof(short)` bytes at the end of the pointer must be scaled to fit into the `sizeof(long)` bytes of the variable `l`**

```
l = *lp;
```

**Compiler knows `sizeof(long)` bytes at end of the pointer need neither promotion nor truncation**

```
f = *sp / 5;
```

**Compiler knows integer division is required**

```
i = *fp / 5;
```

**Compiler knows floating-point division is required**

As pointers are typed, the compiler can ensure that all the above assignments are handled properly.

The last two assignments show how the compiler uses the right hand side of the expression to determine the type of arithmetic to be done. In the second to last assignment, the compiler knows that `*sp` (ie what is at the end of the `sp` pointer) is a short integer. Since the type of `5` is integer, the compiler performs integer division.  $27 / 5 = 5$ . The integer `5` is promoted on assignment to `5.000000`.

With the last assignment, the compiler knows that `*fp` is a float. The type of `5` is integer. The `5` is promoted and floating point division is done. This float is then truncated (with a warning) to an integer in order to be assigned to `i`.

Having typed pointers is a distinct benefit of the C language, since the compiler is able to check for mismatches of type and report such errors to the programmer.

Unfortunately, some compilers, such as the GNU gcc compiler, give no warnings or errors for this code. Others, such as the Microsoft compiler, generate only a warning message when a mismatch in pointer types is detected. This alerts the programmer to a potential problem in the code, but does not halt the compilation. So the rule must be...

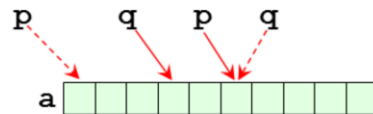
*Never ignore warnings from the compiler, especially warnings such as "Indirection to different types"!*

## Pointer arithmetic

- Whenever an arithmetic operation is performed on a pointer, the compiler scales by the size of the object pointed to:



```
int    a[10];  
int *  p = a;  
int *  q;  
  
p += 5;  
  
q = p;  
  
q -= 2;
```



If a pointer to an integer is incremented, the compiler adds the size of an integer to the stored address. If a pointer to a `float` is incremented, the compiler adds the size of a `float` to the address.

*There is no addition or subtraction that can be performed on a pointer which will not result in scaling.*

The exception to this rule is the `char` pointer. If a `char` pointer is incremented, the compiler scales by one byte (the size of a `char`). This can be thought of as not being scaled at all.

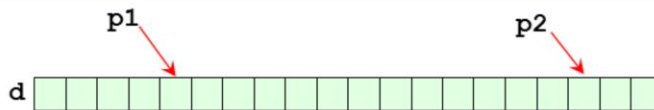
## Pointer subtraction

- Subtraction of pointers (of the same type) is also scaled by the compiler, giving an integer result
- This is only meaningful *when the pointers reference the same array*

```
long double d[20];  
long double * p1, * p2;
```

```
p1 = &d[4];  
p2 = &d[17];
```

```
printf("%d elements separate the two pointers\n", p2 - p1);
```



When an integer is added to a pointer, the addition is scaled by the compiler. In a similar way, when two pointers are subtracted, the number of elements (independent of their type) separating them is returned.

In the example above, the types of `d`, `p1` and `p2` could be substituted for any other type; the result ( $17 - 4 = 13$ ) would be the same.

For the answer to be meaningful, both pointers must point into the same array. If the pointers point into different arrays (or to scalar variables) the answer will only indicate where in memory the compiler has placed the variables/arrays in question.

### restrict keyword (C99)

- **Used for Optimisation**
- **Restricts access to an object through a single pointer**
  - Two (or more) restricted pointers do not refer to objects that overlap
  - New prototype for memcpy( ):
- memmove( ) can be used if the areas do overlap (slower)

```
void *memcpy (void * restrict dest,  
              const void * restrict src, size_t n);
```

**C99 specific**

---

C99 has introduced a new qualifier for pointers – restrict. Its use is purely to allow optimisation, the C99 standard says "Anyone for whom this is not a concern can safely ignore this feature". However, high performance is a major factor in choosing C as a language, so this probably is of concern!

The restrict keyword can be used anywhere a pointer is declared, in a function prototype (as shown), in a formal declaration, or in a struct or union definition.

There is no compilation time checking that the pointer is actually the only access to an object, it assumes the programmer is being truthful.

## Untyped pointers

- Standard C introduces an untyped generic pointer

```
void * v;
```

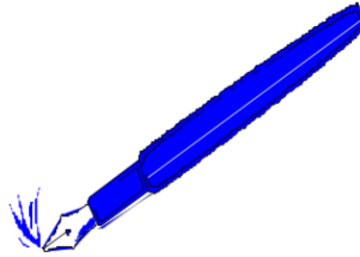
```
int      i;  
float    * p;  
double   d;  
struct termio t;
```

```
✓ v = &i;  
  v = &d;  
  v = &t;  
  v = p;
```

*A void pointer may hold the address of any data item*

```
✗ v = main;
```

*A void pointer may not hold the address of a function*



Having said much regarding pointers being typed in C, Standard C introduces an untyped pointer. It can be thought of as an address container.

A `void*` may hold the address of any item of data, regardless of its type. Its value may be assigned to (or from) any other pointer variable.

A `void` pointer may not contain the address of a function, as will be discussed later in the course.

The compiler does not keep track of what a `void*` is currently pointing to, in much the same way as the compiler keeps no information about the contents of a union.

## void pointers

```
int i = 15, j;
void * v = &i;
int * p;
```

**✗** `v++;`  
`v += 10;` → **A void pointer may not take part in any arithmetic**

**✗** `j = *v;` → **A void pointer may not be dereferenced**

**✓** `p = v;`  
`j = *p;` → **A void pointer must be assigned to a typed pointer to access the data**

Since the compiler maintains no information regarding the data that a `void*` pointer is currently pointing to, the `void` pointer cannot be dereferenced (the compiler does not know how many valid bytes of data are at the end of the pointer).

Similarly, a `void*` cannot be incremented, decremented or take part in arithmetic (the compiler does not know how many bytes of data to step over).

In order to gain access to the data that is currently being pointed to by a `void*`, it is necessary to assign the `void` pointer to a *typed pointer* (this typed pointer can then be dereferenced in the usual manner).

## Using void pointers

- **void pointers can be used wherever addresses need to be manipulated (regardless of type)**

```
void blk_mem_copy(void *to, const void *from, size_t count)
{
    char * cto = to;
    const char * cfrom = from;

    while (count-- > 0)
        *cto++ = *cfrom++;
}
```

```
int      a[10], b[10];
long double d1[50], d2[50];

block_mem_copy(b, a, sizeof(a));
block_mem_copy(d2, d1, sizeof(d1));
```

void pointers are useful when raw addresses need to be manipulated. As discussed on the previous page, these void pointers need to be converted to some tangible pointer type before use.

The compiler cannot perform any type checking where void pointers are used. A call to `blk_mem_copy()` could just as easily be made as follows:

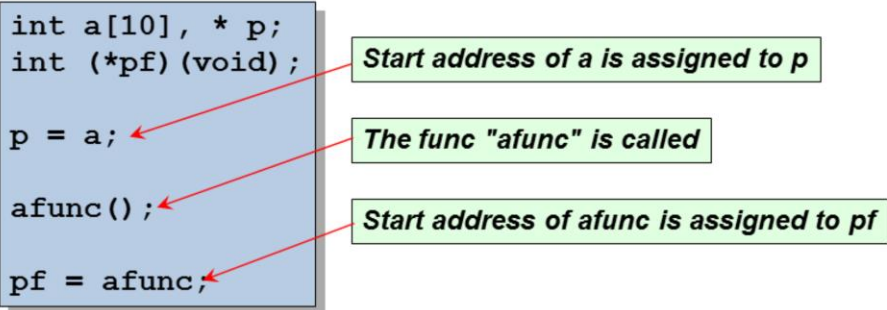
```
blk_mem_copy(a, d1, sizeof(d1));
// Compiler doesn't see any problem with this!
```

No errors would result from the compiler, because it simply converts the two incoming pointers to void pointers without question. At runtime, however, disaster would almost certainly ensue.



## Function pointers

- **When a program runs, the code is in memory**
  - A function will therefore have some address
- **The address of a function can be found by giving the name of the function without parentheses**
  - The mechanism is similar to that of arrays:



The address of a *variable* may be taken because all variables live in memory while the program executes. However, not only variables but also *code* is resident in memory. C has a mechanism for finding the starting address of a function, just as it has a mechanism for finding the address of a data object.

The procedure is similar to that of finding the address of an array. The name of an array is the address at which it starts. Similarly, the name of a function is also the address at which it starts.

C syntax makes calling a function appear indivisible. This is not so, for example:

```
printf("hello world\n");
```

Firstly, `printf` yields the starting address of the function in memory. Secondly, the function-call operator `( )` takes this address and invokes the function at the end of it.

## Declaring pointers to functions

- Function pointers are bound only to functions of the return type and signature specified

```
int  func1(char *);  
int  func2(char, char);  
int  func3(char);  
long func4(char);  
  
int (*pf)(char);  
  
✗ pf = func1;  
✗ pf = func2;  
✓ pf = func3;  
✗ pf = func3();  
✗ pf = func4;
```

Just as a data pointer is bound to a type, so a code (i.e. function) pointer is bound only to functions of the specified signature/return type.

Once declared, a function pointer must point at the "right" kind of function, unless you want a seriously nasty cast to be involved!

## Calling functions via pointers

- The function referenced by the pointer can be called via the function call operator '()

```
#include <math.h>
double d;
double (*p)(double);

p = cos;

d = p(0.125);
d = (*p)(0.125);
```

Standard C call mechanism

K&R call mechanism

- The K&R calling mechanism is still allowed

Having assigned to a function pointer the start address of some suitable function, it only remains to discuss how to invoke the function via the pointer. There are two methods for doing this:

K&R mechanism - The pointer is dereferenced, then the () operator is applied.

Standard C mechanism - The () operator is applied directly to the pointer variable.

Although the Standard C mechanism is much easier to type and remember, it has the disadvantage that it makes the pointer "p" appear as if it were the name of a function. The poor programmer then spends several days searching the code for this function and never finds it!

The K&R mechanism makes it explicit that a function is being called via a pointer.

## Exercise: what does this print?

```
#include <stdio.h>

void q(void)
{
    printf("Goodbye world\n");
}

void p(void)
{
    printf("Hello world\n");
}

int main(void)
{
    void (*p) (void) = q;
    p();
    return 0;
}
```



The program above indicates some of the problems with the Standard C mechanism for calling functions via pointers.

## Function pointer uses

- An alternative to multi-way decision making

```
scanf("%d", &n);
switch (n)
{
    case 0: func0(param);
            break;
    case 1: func1(param);
            break;
    case 2: func2(param);
            break;
    case 3: func3(param);
            break;
    case 4: func4(param);
            break;
}
```

```
void (*pa[5])(int) =
{
    func0,
    func1,
    func2,
    func3,
    func4
};
scanf("%d", &n);

(*pa[n])(param);
pa[n](param);
```

K&amp;R C

Standard C

One typical use of function pointers is to replace `switch` or similar multi-way decision constructs. Some criteria is used to select an element of an array of function pointers. The function at the end of the pointer is then invoked.

The code given above shows the Standard C and K&R calling mechanisms. Although confusing in its previous form, the Standard C mechanism makes it more apparent that a function is being invoked via an element of the array.

When using arrays of pointers to functions, it is *very* important to remember bounds checking. The two examples above are not quite equivalent since the `switch` statement will safely do nothing if the user supplies "strange" values like 800 or -1. The function pointer code would index beyond the end or start of the array and choose a random value from memory, which would be treated as the address of some code in memory. Control of the program would be transferred to that address and the code that isn't there would be executed. This almost guarantees the death of the program.

## "How To" parameters

- A "how to" parameter may be passed into a function

```
#include <math.h>

typedef double (*how_t) (double);

void plot(how_t pf, double start, double end, double step);

double sin_squared(double);

int main(void)
{
    const double pi = 3.1415926535897932385;
    plot(sin, 0, 2 * pi, pi / 20);
    plot(cos, 0, 2 * pi, pi / 20);
    plot(sin_squared, 0, 2 * pi, pi / 20);
    return 0;
}
```

Function pointers are also used to pass functions into other functions. There are many examples of this, several of which are in the standard library and will be covered later in the course.

Here, the `plot()` function will (presumably) plot points on a screen between the values specified. The function itself does not know what points to plot, but every step between start and end invokes the function whose address has been passed as the first parameter.

All the `plot()` function needs to do is to invoke repeatedly the function whose address has been passed. In the slide shown above, `plot()` plots sine and cosine curves by providing the address of the two functions in the standard library. Any curve may be plotted, and to illustrate this, the address of a user-written `sin_squared` function is supplied.

One possible implementation of `plot()` is shown below. The details of displaying a single point have been delegated to some low-level function `draw_point`. This function would presumably be written somewhere else in the program.

```
void draw_point(double x, double y);          /* Draws a single point */
void plot(double(*func)(double), double start, double end, double step)
{
    double x;
    for (x = start; x <= end; x += step)
    {
        double y = func(x); /* Alternative syntax: y = (*func)(x); */
        draw_point(x, y);   /* Call function to draw the point */
    }
}
```

## Dynamic keybinding

- **Function pointers can be used to provide the ability to re-bind keys to different functions**

```
struct keymap
{
    void (*keyfunc) (void) ;
    int keystroke;
};

struct keymap map[] =
{
    { print_file,    K_F1 },
    { next_line,    K_F2 },
    { prev_line,    K_F3 },
    ...
};
```

*Editors are a good example*

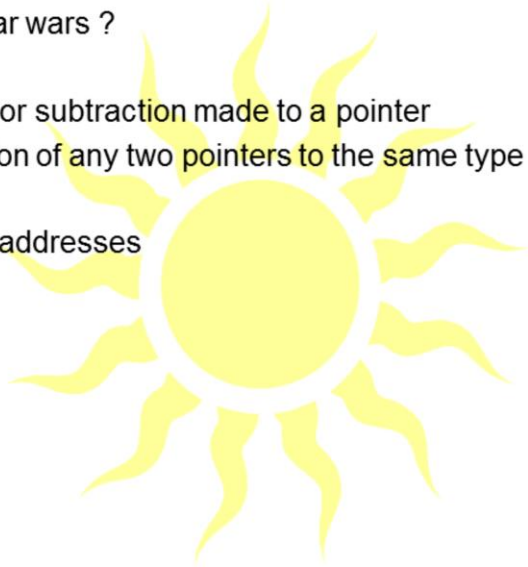
Function pointers are a useful tool when it is desirable to allow users to re-map their keys in some way. The code above maps key values to functions. The user would press F2 and generate the character/s whose value is K\_F2, predefined in some terminal handling header file (on Windows the constant would be VK\_F2). The program would scan the map array until the entry containing this value in the keystroke field was found. The address contained in the keyfunc field would be dereferenced and the function invoked, i.e. `next_line()`.

In these days of customisable editors, the user might wish to change the meaning of F3 so that it prints the file. Using hard-coded switch statements, this would be impossible to achieve.

However, using function pointers, all we have to do is to search the map array for the element containing K\_F3 in the keystroke member. Once found, the address stored in the keyfunc member would be altered to the address of the `print_file` function. There is no need to change the entry containing K\_F1, i.e. both F1 and F3 could access the same function.

## Summary

- **Pointer declaration**
  - Where to put the asterisk - star wars ?
- **Pointer arithmetic**
  - Compiler scales any addition or subtraction made to a pointer
  - Compiler scales the subtraction of any two pointers to the same type
- **Data pointers**
  - `void` pointers contain generic addresses
- **Function pointers**
  - The K&R mechanism
  - The Standard C mechanism



---

In this chapter, we hope to have cleared up any misconceptions you might have had regarding pointer declarations and usage. We have also covered some advanced pointer issues such as the use and misuse of `void` pointers and function pointers. As the course progresses, we will be making use of the techniques covered in this chapter.