ADVANCED C: POINTERS

Welcome to this course on C programming with pointers. This document provides some additional information to assist you in using the course materials. There is also a short section (the FAQ) containing answers to questions that have been asked by students. Any errors and corrections are noted in the Errata.

Information on this course

COURSE LEVEL

This course is suitable for programmers with existing experience of programming in C. It is *not* appropriate for beginners. I already have another course which is aimed at beginner C programmers. Accordingly, I provide no explanations of the basics of C programming and I assume that you already know how to edit, compile and run C programs. You are free to use an editor or IDE on any desktop operating system. However, I cannot provide any support related to specific IDEs or compilers.

HOW TO USE THIS COURSE

The core learning materials in this course are the video lectures. If you are already a fairly experienced programmer, you might want to skip the early lessons and go straight to some of the more advanced topics. However, bear in mind that even the early lessons help to explain some essential basics of pointers.

You should also download the source code archive. In addition, there are some online quizzes to help test your understanding of the topics covered. The quizzes provide information on the lessons that explain the details of specific questions. If you have problems answering a question, I'd suggest that you re-watch the specified video lessons.

THE SOURCE CODE

The source code of all the examples shown in this course is supplied in a Zip archive which you should download and extract. I've provided the code as a series of projects that can be loaded into the free CodeLite editor. If you want the simplest,

fastest way to load and run the code, you may want to install CodeLite. You can download a copy from www.codelite.org.

However, as this is an advanced level course, I am assuming that you are already programming C and you'll probably have your own favourite C editor. That's no problem: just create an empty C project in whatever editor you like, then load my code into a text editor and copy and paste it into your C project.

The programs contain a lot of comments. So again, if you need some clarification of exactly how the code works, reading the comments will probably be helpful.

GOOD LUCK!

Pointers can be quite tough to understand and use safely in the initial stages. I hope this course will help ease your way into understanding pointers. This really is an important subject and a mastery of pointers will really set you apart from other programmers. I hope you enjoy this course!

Best wishes (and good programming!)
Huw Collingbourne
(*Bitwise Courses instructor*)

FAQ

These are answers to some questions that have been asked by students of this course.

STRNCPY() ERROR

Question: My compiler treats `strnpy()` as an error. What should I do?

Answer: Microsoft's C compiler and other C compilers provide alternatives to traditional functions such as `strncpy()`. I have not used these are they are not compatible across all compilers (the MS functions may not work on Linux compilers and so on). Your compiler may warn about certain functions for which alternatives exist. You can ignore the warnings for the purposes of this course or disable them if your compiler or IDE provides that as an option. For example, the Microsoft compiler warns that you need to define this constant when using `strncpy()`.

```
#define _CRT_SECURE_NO_WARNINGS
```

You can define this in your code or at a project-level. Right click the project in the Solution Explorer, select *Properties*, *C/C++*, *Preprocessor*. Then add this (after a semicolon) to the preprocessor definitions: `_CRT_SECURE_NO_WARNINGS`

Or, of course, you can just use the suggested alternative function (the MS compiler suggests `strncpy_s()`). If you are not bothered about compatibility with other compilers that is probably the simplest solution.

Note that functions that are superseded should produce *warnings* rather than errors. If they produce errors you may have the '*Treat warnings as errors*' option enabled. Turn that off. In Visual Studio, you would do that by selecting Project Properties, General, then make sure *Treat Warnings as errors* is set to *No*.

> Also, be sure to watch the lesson called '*Functions that cause errors and warnings*' in Section 2 of the course.

%P OR %D IN FORMAT STRINGS?

Question: To output the address of a pointer in a format string, shouldn't the `%p` specifier be used instead of `%d`?

Answer: You are right. It is usual to use the `%p` format specifier when displaying pointer values. In most cases, that is what you should do. But `%p` prints the address in hexadecimal and, for the purposes of some examples in this course, it is easier to understand the values in decimal - for example, to see when pointer arithmetic increments an address by a certain number of bytes. This is self-evident when shown in decimal format but it can look pretty inscrutable in hexadecimal – at any rate, for those of us who find it tricky to count in hexadecimal!

I explain my reasons for (sometimes) using `%d` instead of `%p` in the first video lesson on *Pointer Arithmetic*. As I mention in that lesson, some compilers may complain about using `%d` for a pointer but (unless you have a "*treat warnings as errors*" option enabled) they should allow it anyway. So even though `%p` is the normal format specifier, the greater clarity `%d` provides in certain of my examples seemed to me worth tolerating any compiler warnings.

Incidentally, `%p` was not a valid format specifier in older implementations of the C language (prior to the definition of the ANSI standard) and it was once quite standard to print pointer values using one of the integer format specifiers. If you are lucky enough to have access to the first edition of Kernighan and Ritchie's 'The C Programing Language' (1978) you will see (page 146) that no `%p` specifier exists. The advantage of using `%p` is that this allows the compiler do some type checking.

> This is discussed more fully in the document "How To Display Pointer Values" which you can download from Section 1 of this course.

TYPEDEFS

Question: what is the difference between:

```
typedef struct ListItem {
    struct ListItem *prev;
    struct ListItem *next;
    int data;
} ListItem;
```

and…

```
typedef struct listitem {
    struct listitem *prev;
    struct listitem *next;
    int data;
} ListItem;
```

Answer: In the first example the pointer variables use your type definition, with a capital **L**, in the second example they use the struct name with a lowercase **l**. Since you have typedefed this struct with the name `ListItem` , they both refer to the same struct in these cases. Typedefs provide a mechanism for using the normal variable declaration syntax with user-defined structures. This is a short explanation from the book that I supply with my course *C programming for Beginners*:

> In the last example, I created a new data-type defined by a struct which I called `cd`. But, unlike standard data types such as `char` and `int` I was not able to create new `cd` variables just by proceeding the variable name with the type name. So this was *not* allowed:
>
> ```
> cd thiscd;
> ```
>
> Instead I had to precede the variable declaration with the keyword `struct` like this:
>
> ```
> struct cd thiscd;
> ```
>
> In fact, there is a way of creating new types that allow the declaration of variables using the same syntax as you would use for standard types. To do this you must explicitly define a type using the keyword `typedef` . By tradition it is normal to name types with an initial capital such as `Mytype` or

`Yourtype` . This is how I declare a type named `CD` (in uppercase) which identifies my struct named `cd` (in lowercase):

```
typedef struct cd CD;

struct cd {
 char name[50];
 char artist[50];
 int trackcount;
 int rating;
};
```

Alternatively, you can combine the `typedef` with the `struct` declaration like this (where the `typedef` name is placed after the closing curly bracket):

```
typedef struct cd {
 Str50 name;
 Str50 artist;
 int trackcount;
 int rating;
} CD;
```

Having done this I can now create `CD` variables like this:

```
CD thiscd;
```

INCONSTANT CONSTANTS?

Question: I assume that constants are non-assignable. But the compiler has let me cast a `const void*` to `int*`. Is it practically possible to avoid reassigning a `const void*`?

Answer: The `const` keyword is an instruction to the compiler to tell it that a certain value should not be reassigned (that is, it should not be treated as a variable). A `const` works as you would expect when used with a simple type such as an `int`. The compiler won't allow this:

```
const int i = 3;
i = 4;
```

But when you have a generic pointer, you will probably at some stage need to cast it to a pointer to a specific type. A cast in effect overrides type-checking. That is true whether or not the original pointer was generic (a pointer to `void`). When you cast something you are telling the compiler that that you know what type the pointer is actually pointing to, so you deliberately don't want it to report any errors. The level of checking on casting or assigning is likely to be different with different compilers on different platforms. You may need to see if there are any flags or options that give you useful warnings with whichever compiler you are using.

You may want to try this code example to see how this works:

```
void const_problem() {
	int i;
	int *pi;
	const int ci = 100;
	const int *cpi = &ci;

	i = 5;		// ok
	pi = &i;	// ok (normal)
	cpi = &i;	// re-assign but no warning
	cpi = &ci;	// ok - but no warning
	pi = cpi;	// warning: "'=': different 'const' qualifiers"
	*pi = 30;	// ok but ci is now 30!
	pi = (int*)cpi; // no warning: the cast tells the compiler to shut up
	//ci = 200;	// error - compiler does not allow
	printf("ci = %d", ci);
}

int main() {
	const_problem();
	return 0;
}
```

Notice that I *can* modify `const int ci`. I *can't* modify it by a simple assignment (`ci = 200` is *not* allowed). But by using the int *pointer* `pi` and then assigning `*pi = 30`, I *can* assign 30 to `ci`.

This really goes to the core of the C philosophy – C assumes that you know what you are doing and will not stop you from doing stupid things - no matter how catastrophic the end results may be. Other languages (such as C# and Java) take the opposite view. This is why C is used in low level programming in microcontrollers, real time etc., where you should know what you are doing and don't want the compiler to get too much in the way.

Errata

The source code archive contains the most up-to-date versions of the sample code used in this course. Occasionally some changes have been made to the source code since the video lectures were recorded. The code in the archive may contain changes to source code, strings and comments. If you have any questions about code shown in the videos, be sure to verify that changes have not been made to the equivalent code in the archive. I will note any significant code changes here in the errata.

MULTIPLEINDIRECTION

Line 33 of the **MultipleIndirection** project is shown incorrectly in a video lesson because the string says "the value of pi" rather than "the value of ppi":

```
printf("The address of pi is %p and the value of pi (what it
points to) is %p\n\n", &pi, ppi);
```

The correct version (in the source code archive) is:

```
printf("The address of pi is %p and the value of ppi (what it
points to) is %p\n\n", &pi, ppi);
```

ADDAFTER

In the demo program, **AddAfter**, the code that relinks the `prev` pointers in the `add_after()` function is shown incorrectly in a video lesson. This is the correct version (supplied in the code archive):

```
item->next = temp->next;
temp->next = item;
item->prev = temp;
item->next->prev = item;
```

QUEUE

One video shows this code in the `dequeue()` function of the **Queue** program:

```
head.last->prev = (LISTITEM*)&head;
```

The correct version (in the code archive is):

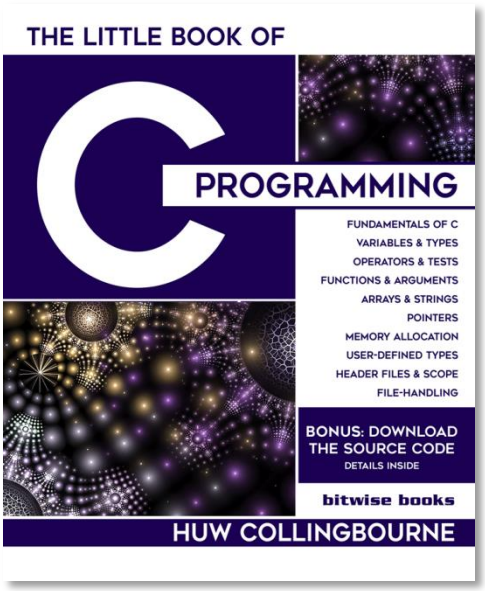
```
head.first->prev = (LISTITEM*)&head;
```

REALLOC()

The `realloc()` function does not change the contents of previously allocated memory. Therefore the second `strncpy()` following the call to `realloc()` in the *Realloc* sample project is not needed.

Books For Programmers...

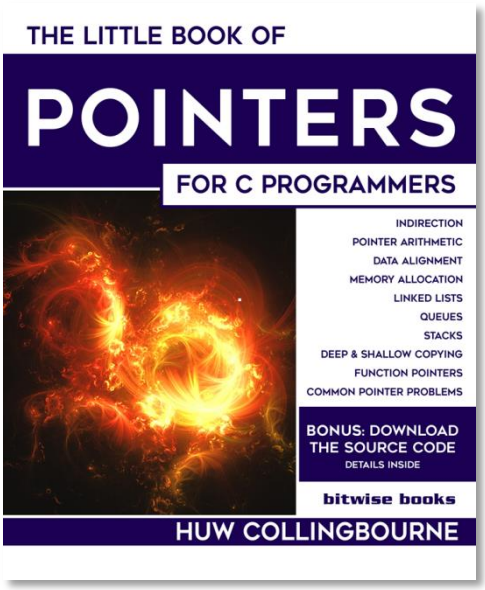
Bitwise Books publishes a range of paperback and Kindle programming books by Huw Collingbourne, the author of this course, including **The Little Book Of Pointers** (for C programmers) which covers the same range of topics covered by this course. See the Bitwise Books web site for more information: www.bitwisebooks.com.



The Little Book Of C

- Fundamentals of C
- Variables, Types, Constants
- Operators and Tests
- Loops and breaks
- Functions and Arguments
- Arrays and Strings
- Pointers
- Memory Allocation
- User-defined Types
- Header Files
- Scope
- File-handling

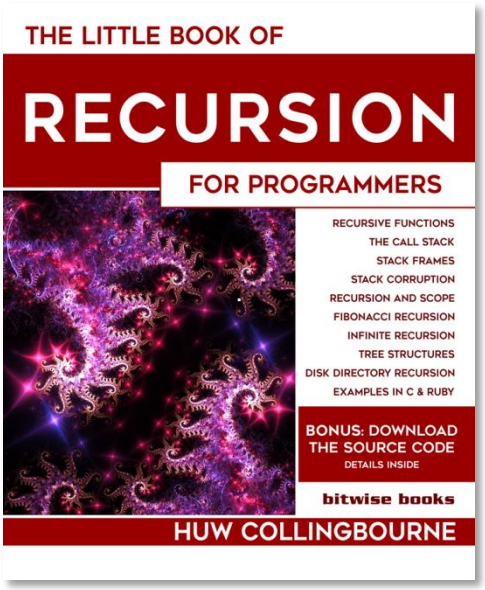
Bonus: Download the source code



The Little Book Of Pointers

- Multiple Indirection
- Pointer arithmetic
- Pointers to structs
- Data Alignment
- Arrays, Strings & Addresses
- Memory Allocation
- Linked Lists (single/double)
- List insertion/deletion
- Stacks
- Queues
- Function Pointers
- Deep & Shallow Copies
- Common Pointer Problems

Bonus: Download all the source code



The Little Book Of Recursion

- Recursive Functions
- The Call Stack
- Stack Frames
- Stack Corruption
- Recursion and Scope
- Fibonacci Recursion
- Infinite Recursion
- Navigating Tree Structures
- Recursing class hierarchies
- Disk Directory Recursion
- Examples in C, Ruby, C#

Bonus: Download all the source code

Free Programming Downloads…

Bitwise Books also provides free programming guides for download. Be sure to sign up to our mailing list to receive these delivered straight to your inbox: www.bitwisebooks.com.