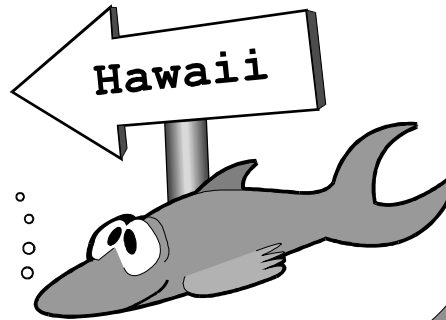


Pointers

- Declaring pointers
- The “&” operator
- The “*” operator
- Initialising pointers
- Type mismatches
- Call by reference
- Pointers to pointers



© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 1

Pointers

This chapter deals with the concepts and some of the many uses of pointers in the C language.

Pointers - Why?

- **Using pointers allows us to:**
 - Achieve call by reference (i.e. write functions which change their parameters)
 - Handle arrays efficiently
 - Handle structures (records) efficiently
 - Create linked lists, trees, graphs etc.
 - Put data onto the heap
 - Create tables of functions for handling Windows events, signals etc.
- **Already been using pointers with `scanf`**
- **Care must be taken when using pointers since there are no safety features**

Pointers - Why?

As C is such a low level language it is difficult to do anything without pointers. We have already seen that it is impossible to write a function which alters any of its parameters.

The next two chapters, dealing with arrays and dealing with structures, would be very difficult indeed without pointers.

Pointers can also enable the writing of linked lists and other such data structures (we look into linked lists at the end of the structures chapter).

Writing into the heap, which we will do towards the end of the course, would be impossible without pointers.

The Standard Library, together with the Windows, Windows 95 and NT programming environments use pointers to functions quite extensively.

One problem is that pointers have a bad reputation. They are supposed to be difficult to use and difficult to understand. This is, however, not the case, pointers are quite straightforward.

Declaring Pointers

- Pointers are declared by using “*”
- Declare an integer:

```
int i;
```

- Declare a pointer to an integer:

```
int *p;
```

- There is some debate as to the best position of the “*”

```
int* p;
```

Declaring Pointers

The first step is to know how to declare a pointer. This is done by using C's multiply character “*” (which obviously doesn't perform a multiplication). The “*” is placed at some point between the keyword **int** and the variable name. Instead of creating an integer, a *pointer to an integer* is created.

There has been, and continues to be, a long running debate amongst C programmers regarding the best position for the “*”. Should it be placed next to the type or next to the variable?

Example Pointer Declarations

```
int      *pi;          /* pi is a pointer to an int */
long int *p;           /* p is a pointer to a long int */
float*   pf;           /* pf is a pointer to a float */
char     c, d, *pc;    /* c and d are a char
                       pc is a pointer to char */
double*  pd, e, f;     /* pd is pointer to a double
                       e and f are double */
char*    start;        /* start is a pointer to a char */
char*    end;          /* end is a pointer to a char */
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 4

Example Pointer Declarations

Pointers Have Different Types

The first thing to notice about the examples above is that C has different *kinds* of pointer. It has pointers which point to **ints** and pointers which point to **long ints**. There are also pointers which point at **floats** and pointers to **chars**.

This concept is rather strange to programmers with assembler backgrounds. In assembler there are just pointers. In C this is not possible, only pointers to certain types exist. This is so the compiler can keep track of how much valid data exists on the end of a pointer. For instance, when looking down the pointer "start" only 1 byte would be valid, but looking down the pointer "pd" 8 bytes would be valid and the data would be expected to be in IEEE format.

Positioning the "*"

Notice that in: **char c, d, *pc;**

it seems reasonable that "c" and "d" are of type **char**, and "pc" is of type pointer to **char**. However it may seem less reasonable that in:

double* pd, e, f;

the type of "e" and "f" is **double** and NOT pointer to **double**. This illustrates the case for placing the "*" next to the variable and not next to the type.

The last two examples show how supporters of the "place the * next to the type" school of thought would declare two pointers. One is declared on each line.

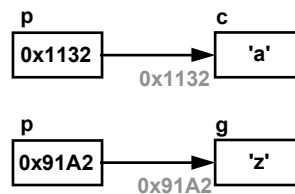
The “&” Operator

- The “&”, “address of” operator, generates the address of a variable
- All variables have addresses except register variables

```
char g = 'z';
int main(void)
{
    char c = 'a';
    char *p;

    p = &c;
    p = &g;

    return 0;
}
```



© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 5

The “&” Operator

The “&” operator, which we have been using all along with **scanf**, generates the address of a variable. You can take the address of any variable which is stack based or data segment based. In the example above the variable “c” is stack based. Because the variable “g” is global, it is placed in the data segment. It is not possible to take the address of any **register** variable, because CPU registers do not have addresses. Even if the request was ignored by the compiler, and the variable is stack based anyway, its address still cannot be taken.

Pointers Are Really Just Numbers

You see from the program above that pointers are really just numbers, although we cannot say or rely upon the number of bits required to hold the number (there will be as many bits as required by the hardware). The variable “p” contains not a character, but the address of a character. Firstly it contains the address of “c”, then it contains the address of “g”. The pointer “p” may only point to one variable at a time and when pointing to “c” it is not pointing anywhere else.

By “tradition” addresses are written in hexadecimal notation. This helps to distinguish them from “ordinary” values.

Printing Pointers


The value of a pointer may be seen by calling **printf** with the **%p** format specifier.

Rules

- **Pointers may only point to variables of the same type as the pointer has been declared to point to**
- **A pointer to an `int` may only point to an `int`**
 - not to `char`, `short int` or `long int`, certainly not to `float`, `double` or `long double`
- **A pointer to a `double` may only point to a `double`**
 - not to `float` or `long double`, certainly not to `char` or any of the integers
- **Etc.....**

```
int    *p;           /* p is a pointer to an int */
long   large = 27L;  /* large is a long int,
                      initialised with 27 */

p = &large;          /* ERROR */
```



© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 6

Rules

Assigning Addresses

The compiler is very firm with regard to the rule that a pointer can only point at the type it is declared to point to.

Let us imagine a machine where an **int** and a **short int** are the same size, (presumably 2 bytes). It would seem safe to assume that if we declared a pointer to an **int** the compiler would allow us to point it at an **int** and a **short int** with impunity. This is definitely not the case. The compiler disallows such behavior because of the possibility that the next machine the code is ported to has a 2 byte **short int** and a 4 byte **int**.

How about the case where we are guaranteed two things will be the same size? Can a pointer to an **int** be used to point to an **unsigned int**? Again the answer is no. Here the compiler would disallow the behavior because using the **unsigned int** directly and in an expression versus the value at the end of the pointer (which would be expected to be **int**) could give very different results!

The “*” Operator

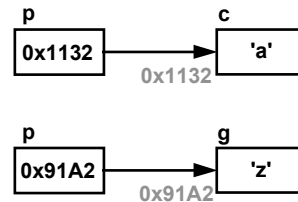
- The “*”, “points to” operator, finds the value at the end of a pointer

```
#include <stdio.h>
char g = 'z';
int main(void)
{
    char c = 'a';
    char *p;

    p = &c;
    printf("%c\n", *p);

    p = &g;
    printf("%c\n", *p);

    return 0;
}
```



print “what p points to”

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 7

The “*” Operator

The “*” operator is in a sense the opposite of the “&” operator. “&” generates the address of a variable, the “*” uses the address that is stored in a variable and finds what is at that location in memory.

Thus, in the example above, the pointer “p” is set to point to the variable “c”. The variable “p” contains the number 0x1132 (that’s 4402 in case you’re interested). “*p” causes the program to find what is stored at location 0x1132 in memory. Sure enough stored in location 0x1132 is the value 97. This 97 is converted by “%c” format specifier and ‘a’ is printed.

When the pointer is set to point to “g”, the pointer contains 0x91A2 (that is 37282 in decimal). Now the pointer points to the other end of memory into the data segment. Again when “*p” is used, the machine finds out what is stored in location 0x91A2 and finds 122. This is converted by the “%c” format specifier, printing ‘z’.

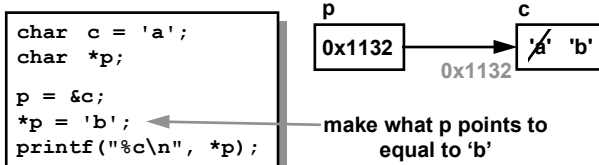
Writing Down Pointers

- It is not only possible to *read* the values at the end of a pointer as with:

```
char c = 'a';
char *p;

p = &c;
printf("%c\n", *p);
```

- It is possible to *write* over the value at the end of a pointer:



© Cheltenham Computer Training 1994/1997

sales@cctrain.demon.co.uk

Slide No. 8

Writing Down Pointers

We have just seen an example of reading the value at the end of a pointer. But it is possible not only to *read* a value, but to *write* over and thus *change* it. This is done in a very natural way, we change variables by using the assignment operator, “=”. Similarly the value at the end of a pointer may be changed by placing “*pointer” (where “pointer” is the variable containing the address) on the left hand side of an assignment.

In the example above: ***p = 'b';**

literally says, take the value of 98 and write it into wherever “p” points (in other words write into memory location 0x1132, or the variable “c”).

Now you’re probably looking at this and thinking, why do it that way, since

c = 'b';

would achieve the same result and be a lot easier to understand. Consider that the variables “p” and “c” may live in different blocks and you start to see how a function could alter a parameter passed down to it.

Initialisation Warning!

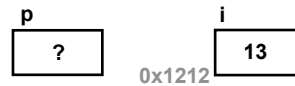
- The following code contains a horrible error:

```
#include <stdio.h>

int main(void)
{
    short i = 13;
    short *p;

    *p = 23;
    printf("%hi\n", *p);

    return 0;
}
```



© Cheltenham Computer Training 1994/1997 sales@ccctrain.demon.co.uk

Slide No. 9

Initialization Warning!

Always Initialize Pointers

The code above contains an all too common example of a pointer bug. The user presumably expects the statement:

`*p = 23;`

to overwrite the variable "i". If this is what is desired it would help if the pointer "p" were first set to point to "i". This could be easily done by the single statement:

`p = &i;`

which is so sadly missing from this program. "p" is an automatic variable, stack based and *initialized with a random value*. All automatic variables are initialized with random values, pointers are no exception. Thus when the statement:

`*p = 23;`

is executed we take 23 and randomly overwrite the two bytes of memory whose address appears in "p". These two random bytes are very unlikely to be the variable "i", although it is theoretically possible. We could write anywhere in the program. Writing into the code segment would cause us to crash immediately (because the code segment is read only). Writing into the data segment, the stack or the heap would "work" because we are allowed to write there (though some machines make parts of the data segment read only).

General Protection Fault

There is also a possibility that this random address lies outside the bounds of our program. If this is the case and we are running under a protect mode operating system (like Unix and NT) our program will be killed before it does any real damage. If not (say we were running under MS DOS) we would corrupt not our own program, but another one running in memory. This could produce unexpected results in *another* program. Under

Windows this error produces the famous “GPF” or General Protection Fault.

Initialise Pointers!

- Pointers are best initialised!
- A pointer may be declared and initialised in a single step

```
short    i = 13;
short    *p = &i;
```

- This does NOT mean “make what p points to equal to the address of i”
- It DOES mean “declare p as a pointer to a short int, make p equal to the address of i”

```
short    *p = &i;
```

```
short    *p = &i;
short    *p = &i;
```

© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 10

Initialize Pointers!

Hours of grief may be saved by ensuring that all pointers are initialized before use. Three extra characters stop the program on the previous page from destroying the machine and transforms it into a well behaved program.

Understanding Initialization

In the line:

```
short *p = &i;
```

it is very important to understand that the “*” is not the “find what is pointed to” operator. Instead it ensures we do not declare a **short int**, but a *pointer to a short int* instead.

This is the case for placing the “*” next to the type, if we had written

```
short* p = &i;
```

It would have been somewhat more obvious that we were declaring “p” to be a pointer to a **short int** and that we were initializing “p” to point to “i”.

NULL

- A special invalid pointer value exists #defined in various header files, called NULL
- When assigned to a pointer, or when found in a pointer, it indicates the pointer is invalid

```
#include <stdio.h>

int main(void)
{
    short    i = 13;
    short    *p = NULL;

    if(p == NULL)
        printf("the pointer is invalid!\n");
    else
        printf("the pointer points to %hi\n", *p);

    return 0;
}
```

© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 11

NULL

We have already seen the concept of preprocessor constants, and how they are **#defined** into existence. A special define exists in the "stdio.h" header file (and a few other of the Standard headers just in case), called NULL. It is a special *invalid* value of a pointer.

The value may be placed in any kind of pointer, regardless of whether it points to **int**, **long**, **float** or **double**.

NULL and Zero

You shouldn't enquire too closely into what the value of NULL actually is. Mostly it is defined as zero, but you should never assume this. On some machines zero is a legal pointer and so NULL will be defined as something else.

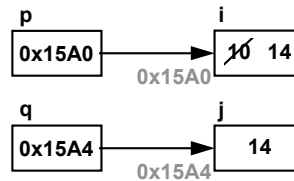
Never write code assuming NULL and zero are the same thing, otherwise it will be non portable.

A World of Difference!

- There is a great deal of difference between:

```
int i = 10, j = 14;
int *p = &i;
int *q = &j;

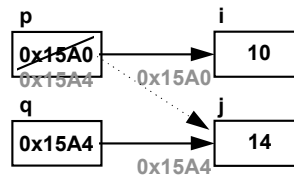
*p = *q;
```



and:

```
int i = 10, j = 14;
int *p = &i;
int *q = &j;

p = q;
```



© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 12

A World of Difference!

**What is
Pointed to vs
the Pointer
Itself**

It is important to understand the difference between:

`*p = *q;`

and

`p = q;`

In the first, "**`*p = *q`**", what is pointed to by "`p`" is overwritten with what is pointed to by "`q`". Since "`p`" points to "`i`", and "`q`" points to "`j`", "`i`" is overwritten by the value stored in "`j`". Thus "`i`" becomes 14.

In the second statement, "**`p = q`**" there are no "**`*`**"s. Thus the value contained in "`p`" itself is overwritten by the value in "`q`". The value in `q` is `0x15A4` (which is 5540 in decimal) which is written into "`p`". If "`p`" and "`q`" contain the same address, `0x15A4`, they must point to the same place in memory, i.e. the variable "`j`".

Fill in the Gaps

```
int main(void)
{
    int i = 10, j = 14, k;
    int *p = &i;
    int *q = &j;

    *p += 1;

    p = &k;

    *p = *q;

    p = q;

    *p = *q;

    return 0;
}
```

i
0x2100

j
0x2104

k
0x1208

p
0x120B

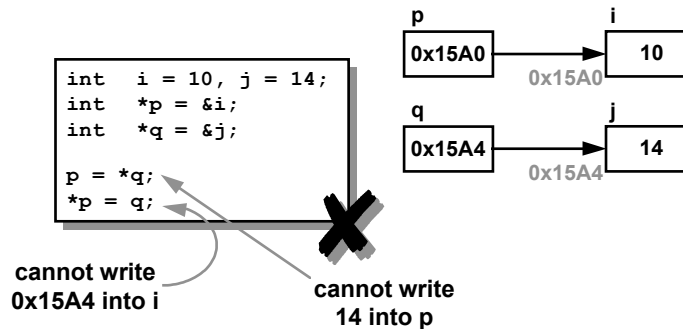
q
0x1210

Fill in the Gaps

Using the variables and addresses provided, complete the picture. Do not attach any significance to the addresses given to the variables, just treat them as random numbers.

Type Mismatch

- The compiler will not allow type mismatches when assigning to pointers, or to where pointers point



© Cheltenham Computer Training 1994/1997

sales@ccctrain.demon.co.uk

Slide No. 14

Type Mismatch

The compiler checks very carefully the syntactic correctness of the pointer code you write. It will make sure when you assign to a pointer, an address is assigned. Similarly if you assign to what is at the end of a pointer, the compiler will check you assign the “pointed to” type.

There are some programming errors in the program above. The statement:

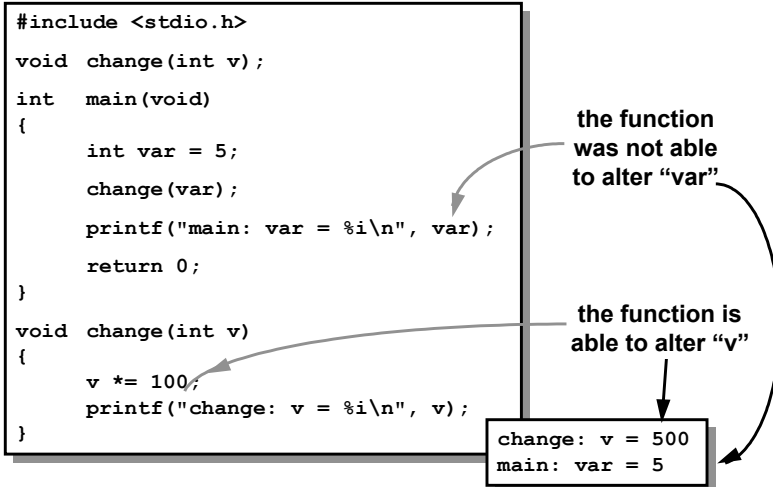
p = *q;

would assign what is pointed to by “q” (i.e. 14), into “p”. Although this would seem to make sense (because “p” just contains a number anyway) the compiler will not allow it because the types are wrong. We are assigning an **int** into an **int***. The valid pointer 0x15A0 (5536 in decimal) is corrupted with 14. There is no guarantee that there is an integer at address 14, or even that 14 is a valid address.

Alternatively the statement: ***p = q;**

takes the value stored in “q”, 0x15A4 (5540 in decimal) and writes it into what “p” points to, i.e. the variable “i”. This might seem to make sense, since 5540 is a valid number. However the address in “q” may be a different size to what can be stored in “i”. There are no guarantees in C that pointers and integers are the same size.

Call by Value - Reminder



© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 15

Call by Value - Reminder

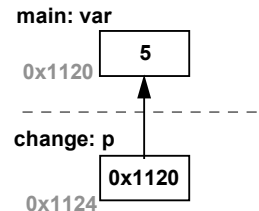
This is a reminder of the call by value program.

The **main** function allocates a variable "var" of type int and value 5. When this variable is passed to the **change** function a copy is made. This copy is picked up in the parameter "v". "v" is then changed to 500 (to prove this, it is printed out). On leaving the **change** function the parameter "v" is thrown away. The variable "var" still contains 5.

Call by Reference

prototype "forces" us to pass a pointer

```
#include <stdio.h>
void change(int* p);
int main(void)
{
    int var = 5;
    change(&var);
    printf("main: var = %i\n", var);
    return 0;
}
void change(int* p)
{
    *p *= 100;
    printf("change: *p = %i\n", *p);
}
```



```
change: *p = 500
main: var = 500
```

© Cheltenham Computer Training 1994/1997

sales@cctrain.demon.co.uk

Slide No. 16

Call by Reference

This program demonstrates call by reference in C. Notice the prototype which requires a single pointer to **int** to be passed as a parameter.

When the **change** function is invoked, the address of "var" is passed across:

change(&var);

The variable "p", declared as the parameter to function **change**, thus points to the variable "var" within **main**. This takes some thinking about since "var" is not *directly* accessible to **main** (because it is declared in another function block) however "p" is and so is wherever it points.

By using the "***p**" notation the **change** function writes down the pointer over "var" which is changed to 500.

When the **change** function returns, "var" retains its value of 500.

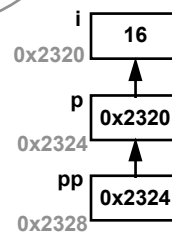
Pointers to Pointers

- C allows pointers to any type
- It is possible to declare a pointer to a pointer

```
#include <stdio.h>
int main(void)
{
    int i = 16;
    int *p = &i;
    int **pp;

    pp = &p;
    printf("%i\n", **pp);
    return 0;
}
```

pp is a "pointer to" a
"pointer to an int"



© Cheltenham Computer Training 1994/1997

sales@ccttrain.demon.co.uk

Slide No. 17

Pointers to Pointers

The declaration: **int i;**

declares "i" to be of type int: **int *p;**

declares "p" to be of type pointer to int. One "*" means one "pointer to". Thus in the declaration:

int **pp;

two *s must therefore declare "pp" to be of type a pointer to a pointer to **int**.

Just as "p" must point to **ints**, so "pp" must point to pointers to **int**. This is indeed the case, since "pp" is made to point to "p". "*"p" causes 16 to be printed

printf("%p", pp);

would print 0x2324 whereas

printf("%p", *pp);

would print 0x2320 (what "pp" points to).

printf("%i", **pp);

would cause what "0x2320 points to" to be printed, i.e. the value stored in location 0x2320 which is 16.

Review

```
int main(void)
{
    int i = 10, j = 7, k;
    int *p = &i;
    int *q = &j;
    int *pp = &p;

    **pp += 1;

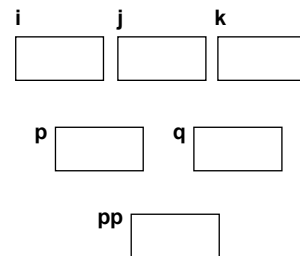
    *pp = &k;

    **pp = *q;

    i = *q***pp;

    i = *q/**pp;    /* headache? */;

    return 0;
}
```



Review Questions

What values should be placed in the boxes?