Notes 10.0: Pointers

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2013 session 1

Declaring and initialising a pointer

A pointer is a data item whose value is an address.

Getting the value *add* of a pointer *p* is usually done for the purpose of going to *add* and reading or modifying what is stored there; but the notion of "what is stored there" is meaningless unless one knows the type of the data item at location *add* (which determines the number of bytes to read, and how those bytes should be decoded).

Below, `b` is declared and initialised as a pointer to an `int`.

```
int a;
int *b = &a;
```

b	0xbfffb638	0xbffff4c0	int *
a	0xbffff4c0	0...011	int

Operations on pointers

- The unary operator `&` applied to a variable *v* yields the address in memory where the value of *v* is stored.
- The `%p` specifier is used to print out the value of a pointer.
- The unary operator `*` applied to a pointer *p* yields the value of the data item whose address is the value of *p*; it is called the indirection or dereferencing operator.
- Hence `val2 = val1` is equivalent to

```
ptr = &val1;
val2 = *ptr;
```

- To declare a pointer we use `*` and indicate the type of the variable it points to, *e.g.*:
  - `int *pt1;` declares `pt1` to be a pointer to a data item of type `int`.
  - `double *pt2;` declares `pt2` to be a pointer to a data item of type `double`.

Changing variables in the calling function

Remember that all function arguments are passed by value: the called function is given the value of its arguments in temporary variables rather than the originals, as illustrated in address.c

Hence the called function cannot directly alter a variable in the calling function.

Moreover, a called function can return at most one value to the calling function.

But using pointers, a function can indirectly alter any number of variables in the calling function: it just needs to be given as arguments the addresses of the variables whose values have to be changed.

Swapping variables: an unsuccessful attempt

```
#include <stdio.h>
void swap(int, int);

int main(void) {
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x , y);
    swap(x, y);
    printf("Now x = %d and y = %d.\n", x, y);
    return 0;
}
void swap(int u, int v) {
    printf("Originally u = %d and v = %d.\n", u , v);
    int temp = u;
    u = v;
    v = temp;
    printf("Now u = %d and v = %d.\n", u, v);
}
```


Swapping variables: a successful attempt

```
#include <stdio.h>
void swap(int *, int *);

int main(void) {
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x, y);
    swap(&x, &y);
    printf("Now x = %d and y = %d.\n", x, y);
    return 0;
}

void swap(int *u, int *v) {
    int temp = *u;
    *u = *v;
    *v = temp;
}
```


Size of pointers versus size of data items pointed to

- Distinguish between the size of a pointer and the size of the object pointed to.
- The value of a pointer occupies a fixed number of bytes, usually 4.
- The `%p` conversion specifier to `printf()` prints the value of a pointer (representing a location in memory) in hexadecimal.
- The size of the object being pointed to varies from 1 byte (for data items of type `char`) to large numbers of bytes (*e.g.*, for arrays as data items).
- Adding 1 to a pointer increases the value of the pointer by the size, in bytes, of the type of the object that is pointed to.

Arrays and pointers (1)

How do we get the address of an array of name `my_array`?

- Like with any other variable: by preceding it with an ampersand: `&my_array`.
- Or by taking the address of the first element of the array: `&my_array[0]`.
- Or using the name of the array *itself*, because in some respects, arrays are treated as pointers, and in many contexts, arrays are cast to pointers: `my_array`.

The value of an array of a given type can be assigned to a pointer of that type (but not the other way around). This means that an array can be cast to a pointer, but not the other way around. This is rather obvious, as a pointer can know the *type* of elements that make up the array, but not the *number of elements* in the array.

arrays_as_pointers.c illustrates how arrays can be "treated as pointers."

Arrays and pointers (2)

So "treating arrays as pointers" means that instead of using array indexes, we can use pointer arithmetic on the name of the array:

```
dates == &dates[0]          // same address
*dates == dates[0]        // same value
dates + 2 == &dates[2]      // same address
*(dates + 2) == dates[2]    // same value
```

```
#include <stdio.h>
#define MONTHS 12
int main(void) {
    int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
    for (int i = 0; i < MONTHS; ++i)
        printf("Month %2d has %d days.\n", i + 1, *(days + i));
    return 0;
}
```


Arrays cannot be passed as arguments (1)

- Even though you can write functions that pretend to take arrays as argument, that function actually takes a pointer as arguments.
- If the array passed as an argument to a function and cast to a pointer needs to be "remembered" to come from an array, then an additional parameter that represents the length of the array is usually necessary.
- So the following prototypes are equivalent.

  ```
  int f(int *, int);
  int f(int [], int);
  ```

  The first one is more faithful to the true type of the argument; the second might better convey the programmer's intentions: an argument such as `arg[]` reminds the reader that the function is expected to be used by passing an array of `int`s (cast to a pointer to an `int`) as argument. Still `f()` treats its first argument as a pointer.

The fact that a function takes as argument a pointer to an `int`, not as an array of `int`s, is exemplified in function_args.c

Passing arrays as arguments (2)

Rather than passing the size of the array as an argument, we can instead pass a pointer that points to where the array stops. In the previous program, the prototype for `sum()` would change to

```
int sum(int *, int *);
```

`main()` would call `sum()` as

```
int answer = sum(array, array + SIZE);
```

and the definition of `sum()` would change to

```
int sum(int *start, int *end) {
    int total = 0;
    while (start < end)
        total += *(start++);
    return total;
}
```


Incrementing and decrementing

The dereference operator has lower priority than the postfix increment and decrement operators, hence `*p1++` is equivalent to `*(p1++)` and `*p1--` is equivalent to `*(p1--)`. See inc_dec.c for an illustration.

Note that the increment operator can only be applied to variables declared as pointers, not as arrays:

```
#include <stdio.h>
#define SIZE 3
int main(void) {
    int array[] = {1, 2, 3};
    printf("%d\n", *(array + 1)); // OK
    printf("%d\n", *(array++)); // NOT OK
    return 0;
}
```

Operations on pointers (1) pt_ops.c

The operations that can be performed on pointers are:

- assignment: assign an address to a pointer;
- dereferencing: find the value stored in the variable the pointer points to;
- pointer address: find the address of a pointer;
- incrementing: using either `+` or `++`;
- decrementing: using either `-` or `--`;
- differencing: for pointers that point into the same array.

See op_on_pointers.c for an illustration.

A mistake to avoid...

Beware the following mistake: do not dereference an uninitialized pointer:

```
int *pt;
*pt = 2;  // NO!!!
```

- The declaration `int *pt` allocates some memory to store the value of the pointer `pt`.
- But the pointer is uninitialized: hence its value is random, and might be the address of some data or program.
- The assignment `*pt = 2` overwrites the value stored at that random address, potentially creating damage.
- Remember that a pointer should always been assigned a memory location that has been allocated before the pointer is used.

The const keyword (1)

The same memory location can be shared by many different data items. The keyword `const` expresses that the contents of memory cannot be changed through the variable qualified by `const`, but it can possibly be modified through other variables, as demonstrated in const.c.

C passes variables by value in order to prevent the called function from modifying the original data. This means that with arrays, a called function cannot modify the address of the first element of the array; but it can modify the elements of the array.

To prevent that, we can use the first of the following constructs:

- `const double *` or `double const *`: a pointer to `const` data items of type `double`;
- `double *const`: a `const` pointer to data items of type `double`;
- `const double *const` or `double const *const`: a `const` pointer to `const` data items of type `double`;

The const keyword (2)

In const_arrays.c, the function `show_array()` treats its argument *as though* it were constant; it does not require the original array to be constant. It uses `const` for safety, to avoid an unintentional modification of the data, in contrast to the function `mult_array()` which has to modify the values of the elements of the array.

The program const_pointers.c does not compile as it tries to change the value of a constant pointer.

Functions and multi-dimensional arrays

We can pass as argument to a function a 2-dimensional array ar, say of ints, that will be cast to a pointer to a one-dimensional array of ints.

More generally, we can pass as argument to a function an N-dimensional array of data items of some type, that will be cast to a pointer to an $(N-1)$–dimensional array of data items of that type.

For instance, to pass to a function g() an array declared as

```
int ar[5][2][4][3];
```

we can declare g() using either syntax
`void g(int (*ar)[2][4][3], int rows);` or
`void g(int ar[][2][4][3], int rows);`

The first syntax describes the true type of the argument, the second syntax the intentions of the programmer.

The program mult_arrays.c illustrates.

More on pointers versus arrays

Not only can arrays be treated as pointers; pointers can use array syntax: if ptr is a pointer to an int then ptr[i] and *(ptr + 1) can both be used to retrieve the value stored at a location that can be accessed from the value of ptr by moving right i times the number of bytes used to store an int.

pts_to_arrays.c

Command line arguments (1)

- Command line arguments can be passed to a program, by declaring main() to take two arguments:
  - the first argument, conventionally called argc for argument count, is 1 plus the number of command-line arguments;
  - the second argument, conventionally called argv for argument vector, can be thought of as an array of character strings that contain the arguments, one per string.
- argv[0] is the name of the invoked program.
- If argc is equal to 1 then there is no command-line argument after the program name.
- The optional arguments are argv[1],..., argv[argc - 1].
- Moreover argv[argc] is the null pointer.

Command line arguments (2)

The next program implements the echo command.

```
#include <stdio.h>
#include <stlib.h>

int main(int argc, char **argv) {
    for (int i = 1; i < argc; ++i)
        printf("%s ", *(argv + i));
    printf("\n");
    return EXIT_SUCCESS;
}
```

Note that the call to printf() could be replaced by either `printf("%s ", argv[i]);` or `printf("%s ", *++argv);`.

An example

grep is a Unix utility:

- grep *pattern file_name* outputs all lines in *file_name* that contain *pattern*.
- Given the option -n (before the arguments), grep also outputs the line numbers.
- Given the option -v (before the arguments), grep rather outputs the lines that do *not* match the pattern.
- Options can be combined: -nv is equivalent to -n -v.
- Like most Unix utilities, grep has many more options.

The program find.c implements grep with those two options, but getting data from standard input rather than from a file.

Pointers to functions (1)

- A function is a data item! A pointer to a function points to the address marking the start of the function code. The name of the function is treated as a pointer to that address.
- When declaring a function pointer, we have to declare the type of the function pointed to (which is the type of the value the function returns).
- For instance, `void (*pt)()` is a pointer to a function that returns `void`.
- We can be more precise: `void (*pt)(char *)` is a pointer to a function that returns `void` and that takes as argument a pointer to a `char`.
- This should be compared with `void *pt(char *)`, which is a function that takes as argument a pointer to a `char`, and that returns a pointer to (a variable of type) `void`.

Pointers to functions (2)

It is possible to:

- assign a function to a function pointer;
- use a function pointer to access a function;
- pass a function pointer as an argument to a function.

For instance:

```
void to_upper(char *);
void (*pf)(char *);
pf = to_upper;     // the function name is a pointer
char course[] = "comp9021";
(*pf)(course);
pf(course);              // alternative syntax
void show(void (*pf)(char *), char *str);
show(pf, course);
show(to_upper, course);  // alternative syntax
```


Complicated declarations

The syntax for declarations involving pointers, functions and arrays and the rules to decode them can be described as follows, working from inside to outside, possibly starting with a variable.

- `*/type ...[.]`
  `...` array `[.]` of pointers or basic types
- `...[.][..]`
  `...` array `[.]` of arrays
- `*/type/void *...`
  `...` pointer to pointer or basic type or `void`
- `(*...) ()/[.]`
  `...` pointer to function or array
- `*/type/void ...()`
  `...` function returning pointer or type or `void`

Example

- `char **argv`
  `argv`: pointer to pointer to `char`
- `int (*daytab)[13]`:
  `daytab`: pointer to array `[13]` of `int`
- `int *daytab[4]`:
  `daytab`: array `[4]` of pointer to `int`
- `void *comp()`
  `comp`: function returning pointer to `void`
- `void (*comp)()`
  `comp`: pointer to function returning `void`
- `char (*(*x())[7])()`
  `x`: function returning pointer to array `[7]` of pointer to function returning `char`
- `char (*(*x[3])())[5]`
  `x`: array `[3]` of pointer to function returning pointer to array `[5]` of `char`