Introduction to Programming (Adv)

School of Computer Science, University of Sydney

Copyright Warning

Lecture 4: C Reference types and basic I/O

Reference types, memory and files

Pointers in C

The pointer is a data type which stores a memory address.

Any pointer type contains a * in its declaration

```
char * variable;
int * variable;
float * variable;
void * variable;
```

All of these are pointer types. The type associated with the pointer (char, int, float etc.) tells the compiler how to treat this kind of memory.

Address of &

The unary & operator returns the address of a variable.

```c
int x = 5;
&x ; // no effect, this is the memory address of x
```

```c
void foo(int x, float y) {
    &x; // this is the memory address of x
    printf("%p\n", &y); // print memory address of y
}
```

Address of & (cont.)

Now that we can get memory addresses, the pointer assignment can take effect

```c
int x;        // declare x as type "int"
int *x_ptr; // declare x_ptr as type "pointer to int"
x = 5;        // init x with a value
x_ptr = &x; // init x_ptr with a value (a memory address)
printf("x is: %d\n", x);
printf("x_ptr is: %p\n", x_ptr);
```

Dereference operation *

If we have a memory address, then we need to get the value stored there.

Memory addresses are arbitrary bytes, but with the type information of the pointer, the compiler will know how much data should be fetched.

```c
int x;       // declare x as type "int"
int *x_ptr; // declare x_ptr as type "pointer to int"
x = 5;       // init x with a value
x_ptr = &x; // init x_ptr with a value (a memory address)
printf("x is: %d\n", x);
printf("x_ptr is: %p\n", x_ptr);
printf("value pointed to by x_ptr is: %d\n", *x_ptr);
```

Dereference operation * (cont.)

```c
int x;       // declare x as type "int"
char *x_ptr; // declare x_ptr as type "pointer to char"
x = 300;     // init x with a value

// init x_ptr with a value (a memory address)
// explicitly treat memory address as pointer to type char
x_ptr = (char*)&x;

printf("x is: %d\n", x);
printf("x_ptr is: %p\n", x_ptr);
printf("value pointed to by x_ptr is: %d\n", *x_ptr);
```

Benefit of pointers

Why have pointers?

Don't copy data around. Copy its memory address instead!

At runtime, we can change the memory address being referred to.

```c
int x = 5; // declare and init x
int y = 7; // declare and init y
int *my_ptr; // declare my_ptr as type "pointer to int"
my_ptr = &x; // set my_ptr as memory address of x
printf("my_ptr is: %p\tvalue is: %d\n", my_ptr, *my_ptr);
...
my_ptr = &y; // set my_ptr as memory address of y
printf("my_ptr is: %p\tvalue is: %d\n", my_ptr, *my_ptr);
```

Benefit of pointers (cont.)

```c
// set the value at a particular memory address
void set_value(int *x_ptr, int new_value) {
    *x_ptr = new_value;
}
int x = 1;
int y = 5;
foo( &x, 7 );
foo( &y, 14 );
```

```c
char array[] = { 'a', 'b', 'c', 'd' };

// this is the memory address of where
// the 'b' character is stored
char *mychar = &(array[1])
```

Benefit of pointers (cont.)

```
int array[] = { 1, 2 , 3, 4 };
&array // memory address of array at index zero
&(array[0]) // memory address of array at index zero
&(array[1]) // memory address of array at index one
&(array[2]) // memory address of array at index two
&(array[3]) // memory address of array at index three
```

Special cases for statically allocated array with address operator.
`&array == array == &(array[0])`

Strings in C

There are no objects.

A string is defined as an array of characters terminated by a null character.

```c
char array[] = { 'a', 'b', 'c', 'd', '\0' };
```

This can be treated as a C string because it contains the null character.

When referring to a string, we refer to the memory address of the first character in that string

Reading the entire string requires visiting all subsequent memory locations, and testing whether or not a null character is present.

Strings in C

Another hardcoded string

```c
char *array = "abcd";
```

Moving through a string in C

```c
int i = 0;
while ( *(array + i) != '\0' )
{
   printf("i: %d\tchar is %c\tvalue is: %d\n",
          i, *(array + i), *(array + i));
   i++;
}
```

Relationship array access and dereference operator

[ ] and * are interchangeable

Accessing Values

Declare:

`char array[5];` OR `char *array;`

Then:

`array[1]` is the same as `*(array + 1)`

Example:

```c
char array[5];
char c1 = array[3];
char c2 = *(array + 3);

printf("%d == %d\n", c1, c2);
```

(cont.)

Accessing Addresses

Declare:

`int array[5];` OR `int *array;`

Then:

`&(array[3])` is the same as `array + 3`

Example:

```c
int *array;
int *ptr1 = &(array[3]);
int *ptr2 = array + 3;

printf("%p == %p\n", ptr1, ptr2);
```

(cont.)

Functions:

`void foo(char *array)` is the same as `void foo(char array[])`

Different for declarations with fixed sizes in multiple dimensions:

`char array[][3]` is treated differently to `char **array`

Revisit Strings in C with dereference

Moving through a string in C

```
// get the pointer to the array
char *s_ptr = array;
while ( *s_ptr != '\0' )
{
   printf("char is %c\tvalue is: %d\n",
      *s_ptr, *s_ptr);

   // move the pointer to the next address
   s_ptr ++;
}
```

sizeof operator

Compiler needs to know how big all the types are at compile time

`sizeof` is an operator as part of the C language

`sizeof int` gives the number of bytes an int type will occupy in memory, on this architecture.

To be safer with operators, we use parentheses where possible. This is why `sizeof` is commonly confused as a function `sizeof(int)`

sizeof operator (cont.)

`sizeof(int)` is not fixed. It varies with computer architecture. 4 bytes, 8 bytes etc.

`sizeof(pointer type)`. How many bytes will this occupy? Depends on the amount of addressable memory on the system. e.g. 32 bits -¿ $2^32 = 4294967296possiblememoryaddresses.$

Every pointer type occupies the same space, enough to fit a memory address of this system. `sizeof(int *)` is the same as `sizeof(char *)` is the same as `sizeof(void *)`

sizeof operator (cont.)

sizeof operator can be used on expressions, variables, and types.

sizeof(1) gives the size of the evaluated type, being an int.

sizeof(p) gives the size of the evaluated type of `p`, not enough information here

`sizeof(goriila ***)` gives sizeof of the *pointer type* to gorilla. No matter what gorilla type is, this will be the same as sizeof(void*), the generic version of an address value.

sizeof: know its limits

Danger! with sizeof

sizeof correctly reports ”primitives” as it is compile time operator

Statically allocated arrays have a known size at compile time. Hence, they can return the correct number of bytes using sizeof.

Everything else cannot. There is no tracking, or intelligence in the running of a C program to determine the size of a region of memory. The programmer has to do it.

sizeof: know its limits (cont.)

```
char array[5];
char *ptr = array;

printf("%zu\n", sizeof(char)); // 1
printf("%zu\n", sizeof(array)); // 5 -> compiler knows

printf("%zu\n", sizeof(char*)); // 8 -> address
printf("%zu\n", sizeof(ptr)); // 8 -> address
printf("%zu\n", sizeof(*ptr)); // 1 -> one char
printf("%zu\n", sizeof(&ptr)); // 8 -> address

printf("%zu\n", sizeof(array)); // 5 -> compiler knows
printf("%zu\n", sizeof(array+1)); // 8 -> compiler don't
    care
```

Summary

Reference types are fundamental primitive

Understanding data types sizes and memory layout crucial

Basic operations depend on this

More pointers with further concepts to explore