CS1101C Lecture 13
Some Revision Material

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346

`www.comp.nus.edu.sg/~joxan`

`cs1101c@comp.nus.edu.sg`

Semester II, 2016/2017

Lecture Outline

- Variable types and Casts
- Storage Classes for Variables
  `auto`
  `extern`
  `static`
- Functions and Parameter Passing
  call-by-value
  call-by-reference
- Recursion
- Arrays and Pointers
  sharing a address
  navigating arrays
  two dimensional arrays
  strings
- Structures
  complex and self-referential structures
  arrays of structures

Data Types

Basic types: `char, int, float, double` (Not discussed: `void, enum`)
Modifiers: `short, long, signed, unsigned`
Not discussed: qualifiers: `const, volatile`

Type	Bytes	Bits	Range	
short int	2	16	-32,768 -> +32,767	(32kb)
unsigned short int	2	16	0 -> +65,535	(64Kb)
unsigned int	4	32	0 -> +4,294,967,295	(4Gb)
int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
float	4	32		
double	8	64		
long double	12	96		

These figures are indicative, typical of present PC's.

Conversion and Casting

An operator must have operands of the same type before it can carry out the operation. C will perform some automatic conversion of data types. These are the general rules for binary operators (* + / % etc):

- If either operand is long double the other is converted to long double.
- Otherwise, if either operand is double the other is converted to double
- Otherwise, if either operand is float the other is converted to float
- Otherwise, convert char and short to int
- Then, if an operand is long convert the other to long.

You can also write a cast operator within an expression:

```
int sum, count;
double average;
average = (double) sum / count;
```

Storage Classes

- Classes: `auto, extern, static`
  (Not discussed: the clasees `register, typedef`)

- `auto`
  is the default storage class for local variables.

  ```
  int Count;
  auto int Month;
  ```

  The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

- `extern`
  defines a global variable that is visable to all functions. The variable cannot be initalized *elsewhere* as all it does is point the variable name at a storage location that has been previously defined.

  ```
  Source 1                Source 2
  --------                --------
  int count = 5;          void write() {
  main() {                    extern int count;
     write()                  printf("count is %d\n", count);
  }                       }
  ```

  Count in 'source 2' will have a value of 5. If source 1 changes the value of count. source 2 will see the new value.

Storage classes - Static

`static` is the default storage class for global variables.
The two variables below (count and road) both have a static storage class.

```
static int count = 0;
int road = 0;
main() {
  printf("%d\n", count); printf("%d\n", road);
}
```

`static` can also be defined within a function. The variable is then initalised at compilation time and retains its value between calls. Because it is initialsed at compilation time, the initalistation value must be a constant.

```
void func3();

main() {
    func3(); func3(); func3();
}

void func3() {
    static int i = 0;
    printf("%d ", i++);
}
```

Output: 0 1 2

Static

Example of an important use for `static`:

```
char *myfunc(void);

main() {
    char *text1;
    text1 = myfunc();
}

char *myfunc(void) {
    char text2[10] = "joxan";
    return(text2);
}
```

Func' returns a pointer to the memory location where 'text2' starts BUT text2 has a storage class of `auto` and will disappear when we exit the function.

Instead, we should specify:

```
static char text2[10] = "joxan";
```

Functions

- Declaration of Prototype
  Eg: `int add( int, int);`

- Definition of Function

```
int add( int a, int b) { /* Function definition  */
    int c;
    c = a + b;
    return c;
}
```

- Passing Parameters
  call-by-value
  call-by-reference

- Not discussed: having an *arbirtrary* number of arguments

Function Flow of Control

```
int main() {
    func1();
}
void func1() {
    printf("one ");
    func2();
    printf("three ");
}
void func2() {
    printf("two ");
}

Call sequence:

    start main:
    call func1();
    start func1():
        printf("one ");
        call func2();
             func2():
             printf("two ");
             end func2();
        printf("three ");
        end func1();
    end main

Output: one two three
```

Function Parameters - Call by Value

Straightforward: copy *actual parameters* into the local variables called *formal parameters*.

```
int add(int x, int y);

int main() {
int a = 1, b = 2;
    printf("%d ", add(a, b));
    a++; b++;
    printf("%d\n", add(a, b));
}

int add(int x, int y) {
    return x + y;
}

Output: 3 5
```

Call by Reference

The actual parameter is an *address*. This can be explicitly given, eg:

```
void swap(int *a, int *b);

int main(void) {
    int x = 1, y = 2;
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}

void swap(int *a, int *b) {
    int hold;
    hold = *a; *a = *b; *b = hold;
}
```

Output: `x = 2, y = 1`.

NOTE: When an array is passed as an actual parameter to a function, its *address* is passed, and not the array itself.

Call by Reference

```
int function1(char * array);

main() {
    char array1[10] = "987654321";     /* check it fits */
    function1(array1);
    printf("%s ", array1);
}

function1(char * array) {
    printf("%s\n", array);
    array += 4;            /* Point to the 5th element */
    *array = 'x';          /* Modify the 5th element */
}
```

The output is `987654321 9876x4321`

Call by Reference

```
int set(char *items[]);

main() {
    char *items[]={"apple", "pear", "banana", "grape"};
    set (items);
}

int set(char *items[]) {
    printf ("/t%s\n", items[2]);
}
```

The output is `banana`.

Recursive Functions

```
void recursiveFunction(int num) {
   if (num < 5) {
      printf("%d ", num);
      recursiveFunction(num + 1);
   }
}
```

1	recursiveFunction (0)				
2	printf (0)				
3		recursiveFunction (0+1)			
4		printf (1)			
5			recursiveFunction (1+1)		
6			printf (2)		
7				recursiveFunction (2+1)	
8				printf (3)	
9					recursiveFunction (3+1)
10					printf (4)

Output: `0 1 2 3 4`

Recursive Functions

```
void recursiveFunction(int num) {
    if (num < 5) {
        recursiveFunction(num + 1);
        printf("%d ", num);
    }
}
```

1	recursiveFunction (0)				
2		recursiveFunction (0+1)			
3			recursiveFunction (1+1)		
4				recursiveFunction (2+1)	
5					recursiveFunction (3+1)
6					printf (4)
7				printf (3)	
8			printf (2)		
9		printf (1)			
10	printf (0)				

Output: `4 3 2 1 0`

Recursion - Divide and Conquer

Recall the *factorial* example:

```
int fact(int n) {
    return (n <= 1) ? 1 : n * fact(n - 1);
}
```

The formulation of this program was straightforward from the *definition* of the factorial function:

$$fact(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n * fact(n-1) & \text{if } n > 1 \end{cases}$$

Sometimes the formulation of a recursive function is not so straightforward.

Recursion - Divide and Conquer

- A perfect number is one which is the sum of its proper positive divisors. Eg 6 = 1 + 2 + 3 is the first perfect number. Next 28 = 1 + 2 + 4 + 7 + 14.

- Write a recursive function to detect a perfect number.

- The idea:
  a function `sumdiv(n, k)` sums up the divisors of n *up to k*.
  The base case is easy: `sumdiv(n, 1)` = 1.
  The recursive case:

$$sumdiv(n,k) = \begin{cases} sumdiv(n,k-1)+k & if\ k\ divides\ n \\ sumdiv(n,k-1) & otherwise \end{cases}$$

Perfect Numbers

```
int int sumdiv(int n, int div);

int main() {
    int n = 2, i;
    do {
        i = sumdiv(n, n-1);
        if (n == i) printf("%d ", n);
    } while(++n < 8129);
}

int sumdiv(int n, int div) {
    if (div <= 1) return 1;
    return (n % div == 0) ?
        div + sumdiv(n, div - 1) : sumdiv(n, div - 1);
}
```

Output: 6 28 496 8128

Recursive Functions - Towers of Hanoi

See: `http://www.mazeworks.com/hanoi/index.htm`

```
void hanoi(int n, char LEFT, char RIGHT, char CENTER) {
    if( n > 0) {
        hanoi(n-1, LEFT, CENTER, RIGHT);
        printf("\ndisk %d from %c to %c", n, LEFT, RIGHT);
        hanoi(n-1, CENTER, RIGHT, LEFT);
    }
}

void main()
{
    int n;
    printf("\nEnter no. of disks: ");
    scanf("%d",&n);
    hanoi(n,'L','R','C');
}
```

```
Enter no. of disks: 5

disk 1 from L to R
disk 2 from L to C
disk 1 from R to C
disk 3 from L to R
disk 1 from C to L
disk 2 from C to R
disk 1 from L to R
disk 4 from L to C
disk 1 from R to C
disk 2 from R to L
disk 1 from C to L
disk 3 from R to C
disk 1 from L to R
disk 2 from L to C
disk 1 from R to C
disk 5 from L to R
disk 1 from C to L
disk 2 from C to R
disk 1 from L to R
disk 3 from C to L
disk 1 from R to C
disk 2 from R to L
disk 1 from C to L
disk 4 from C to R
disk 1 from L to R
disk 2 from L to C
disk 1 from R to C
disk 3 from L to R
disk 1 from C to L
disk 2 from C to R
```

Pointers

A pointer is a variable whose value is an *address*.
A pointer is declared by specifying the *type* of object that it points to.
There is a special address called `NULL`.
NOT DISCUSSED: *void* pointers

Two main uses of pointers are to allow:

- two different variables to refer to *common* memory.
  A classic example is a variable in a main program which is to be modified by a formal parameter in a function.

- *dynamic data structures*.
  In conjunction with dynamic memory allocation (recall `malloc`), this allows the program to flexibly use amount the memory depending on its need, and to build complex relationsips between variables.
  (Not Examinable)

Pointers and Memory

```
int *ptr; /* declaration of ptr as pointing to integer */
double *ptr2;
...
*ptr = 2; /* use of pointer, allocates 4 byes for integer 2 */
*ptr2 = 2; /* allocates 8 byes for double 2 */
```

NOTE: ensure that `ptr` and `ptr2` have *valid* addresses before assigning them.
How does one do this?

Some common errors (why?):

```
int i, *ptr1;
char *ptr2;

i = &ptr;
ptr1 = ptr2;
ptr2 = ptr1;
ptr1 = i;
*ptr2 = 'c';
```

Pointers and Arrays

Pointers do not have to point to single variables. They can also point at the cells of an array.

```
int *ptr1, *ptr2;
int a[10];
ptr1 = &a[0]; /* ptr1 points to 1st int in array a */
ptr2 = &a[3]; /* ptr2 points to 4th int in array a */
```

NOTE: The expression `a` in the array declaration `int a[10];` is also a pointer (to an integer). However, the two declarations

```
int a[10];
int *a;
```

are not the same. In the former case, `a` *cannot be changed*.

Similarly, `a` and `ptr` are interchangeable in usage, the declaration `int a[10];` provides *valid memory of 10 ints*, whereas the declaration `int *ptr1;` does not.

Pointer Arithmetic

Valid expressions: where `ptr` and `ptr2` are pointers (to anything):

- Adding/Subtracting a constant offset: `ptr + ` *k* where *k* is an integer.
- Increment/Decrement: `ptr++`, `++ptr`, `ptr--`, `--ptr`
- Pointer subtraction: `ptr - ptr2`
- NOTE: there is no pointer addition

```
int a[5], b[5][10], *ap, *bp, i, j;
double d[5], *dp;
printf("a: ");    for (i = 0; i < 5; i++) printf("%d ", a + i);
printf("\nd: "); for (i = 0; i < 5; i++) printf("%d ", d + i);
printf("\nb: "); for (i = 0; i < 5; i++) printf("%d ", b + i);
i = (int) &a[0];
j = (int) &a[1];
printf("\ni = &a[0] = %d, j = &a[1] = %d\nj - i = %d, &a[1]-&a[0] = %d\n",
    i, j, j-i, &a[1]-&a[0]);
```

Output:

```
a:  2289312 2289316 2289320 2289324 2289328  (difference 4)
d:  2289040 2289048 2289056 2289064 2289072  (difference 8)
b:  2289104 2289144 2289184 2289224 2289264  (difference 40)
i = &a[0] = 2289312, j = &a[1] = 2289316,
j - i = 4, &a[1]-&a[0] = 1
```

NOTE: why are the last 2 numbers different?

Arrays/Pointers as Function Arguments

Recall that a function call with an array parameter results in a passing of the *address* of the array, and not the array itself. Eg: suppose `getline(line, max)` reads at most a line of `max` chars into the array `line`.

```
int getline(int line[], int max); /* line[] is defined without a size */

int main(void) {
   char line[100], line2[200];
   getline(line, 100);
   getline(line2, 200);
}
```

Thus we can use the same function `getline` in order to get lines of input into two arrays of different sizes. This is because the call to `getline` is supplied not with an array, but with a *pointer* to an array. Further, it is known that the array contains integers.

Thus the two calls above are as if we had written

```
int getline(int *line, int max);

int main(void) {
   char line[100], line2[200];
   getline(&line[0], 100);
   getline(&line2[0], 200);
}
```

getline

```
int getline(char *line, int max) {
    int nch = 0, c;
    max = max - 1; /* leave room for '\0' */
    while((c = getchar()) != EOF) {
        if(c == '\n') break;
        if(nch < max) *(line + nch++) = c;
    }
    if(c == EOF && nch == 0) return EOF;
    *(line + nch) = '\0';
    return nch;
}
```

Arrays/Pointers as Function Arguments

Consider now two-dimensional arrays.

Could a function with a parameter array do without *both* the row and column sizes?

```
void setzero(int box[][], int max1, int max2);

int main(void) {
   char box[100][100], box2[200][200];
   setzero(box, 100, 100);
   setzero(box2, 200, 200);
}

void setzero(int box[][], int max1, int max2) {
   int i, j;
   for (i = 0; i < max1; i++)
      for (j = 0; j < max2; j++) box[i][j] = 0;
}
```

The answer is NO. The problem is how to compute the address of `box[i][j]`. Why?

In order to compute `box[i][j]` (or equivalently `*(box + i) + j`), we require

- The address of the first element of the array (`box`)
- The size of the type of the elements of the array (here, `sizeof(int)`)
- The 2nd dimension of the array (here, 100 or 200)
- The specific index value for the first dimension (here, `i`)
- The specific index value for the second dimension, (here, `j`).

Arrays/Pointers as Function Arguments

A correct way:

```
void setzero1(int box[][100], int max);
void setzero2(int box[][200], int max);

int main(void) {
   char box[100][100], box2[200][200];
   setzero1(box, 100);
   setzero2(box2, 200);
}

void setzero1(int box[][100], int max) {
   int i, j;
   for (i = 0; i < max; i++)
      for (j = 0; j < 100; j++) box[i][j] = 0;
}

void setzero2(int box[][200], int max) {
   int i, j;
   for (i = 0; i < max; i++)
      for (j = 0; j < 200; j++) box[i][j] = 0;
}
```

Reminders on Two-Dimensional Arrays

- Recall that

  ```
  int a[100] = {6, 7, 8, 9}, *ptr = &a[1];
  ```

  results in `a` being a pointer to a one-dimensional array.
  Thus `a[3]`, `*(a + 3)` and `*(ptr + 2)` are the same, equalling 9.

- A two-dimensional array eg: `int a[5][5];` is treated as a *one-dimensional array of a one-dimensional array*.
  - `a` refers to a two-dimensional array
  - `*a, *(a + 1), *(a + 2), ...` are pointers to a one-dimensional array.
    Recall that these are the same as `a[0], a[1], a[2], ...`
  - `**a, **(a + 1), **(a + 2)` are the first three elements of the first column of `a`.
  - `*(*(a + 4)), *(*(a + 4) + 1), *(*(a + 4) + 2)` are the first three elements of the fifth row of `a`.

```
main() {
int a[][5] = {{ 0,  1,  2,  3,  4},
              {10, 11, 12, 13, 14},
              {20, 21, 22, 23, 24},
              {30, 31, 32, 33, 34},
              {40, 41, 42, 43, 44}};
int *b[] = {a[0], a[1], a[2], a[3], a[4]};
int *p = a[0];
int **q = b;

printf("Addresses of first three rows of a:\n");
printf("  *a=a[0]=%p, *(a+1)=a[1]=%p, *(a+2)=a[2]=%p\n",
   *a,*(a+1),*(a+2));
printf("Checking their differences:\n");
printf("  a[1]-a[0]=%d,a[2]-a[0]=%d\n", a[1]-a[0], a[2]-a[0]);
}
```

```
OUTPUT:

Addresses of first three rows of a:
  *a=a[0]=0x22ee50, *(a+1)=a[1]=0x22ee64, *(a+2)=a[2]=0x22ee78
Checking their differences:
  a[1]-a[0]=5, a[2]-a[0]=10
```

```
main() {
int a[][5] = {{ 0,  1,  2,  3,  4},
              {10, 11, 12, 13, 14},
              {20, 21, 22, 23, 24},
              {30, 31, 32, 33, 34},
              {40, 41, 42, 43, 44}};
int *b[] = {a[0], a[1], a[2], a[3], a[4]};
int *p = a[0];
int **q = b;

printf("First three elements of first column of a:\n");
printf("  **a = %d, **(a+1) = %d, **(a+2) = %d\n",
    **a, **(a+1), **(a+2));
printf("First three elements of fifth row of a:\n");
printf("  *(*(a+4)) = %d, *(*(a+4)+1) = %d, *(*(a+4)+2) = %d\n",
    *(*(a+4)), *(*(a+4)+1), *(*(a+4)+2));
}

OUTPUT:

First three elements of first column of a:
  **a = 0, **(a+1) = 10, **(a+2) = 20
First three elements of fifth row of a:
  *(*(a+4)) = 40, *(*(a+4)+1) = 41, *(*(a+4)+2) = 42
```

```
main() {
int a[][5] = {{ 0,  1,  2,  3,  4},
              {10, 11, 12, 13, 14},
              {20, 21, 22, 23, 24},
              {30, 31, 32, 33, 34},
              {40, 41, 42, 43, 44}};
int *b[] = {a[0], a[1], a[2], a[3], a[4]};
int *p = a[0];
int **q = b;

printf("First three elements of b:\n");
printf("  *b = %p, *(b+1) = %p, *(b+2) = %p\n", *b,*(b+1),*(b+2));
q += 2;
printf("Last three elements of first column of a:\n");
printf("  **q = %d, **(q+1) = %d, **(q+2) = %d\n",
    **q, **(q+1), **(q+2));
}
```

OUTPUT:

```
First three elements of b:
  *b = 0x22ee50, *(b+1) = 0x22ee64, *(b+2) = 0x22ee78
Last three elements of first column of a:
  **q = 20, **(q+1) = 30, **(q+2) = 40
```

```
main() {
int a[][3] = {{ 0,  1,  2},
              {10, 11, 12},
              {20, 21, 22},
              {30, 31, 32}};
int *p = a[0];

printf("Addresses of a:\n    %p\n", a);
printf("Addresses of a[0] ... a[3]:\n    %p %p %p %p\n",
    &a[0], &a[1], &a[2], &a[3]);
printf("Contents  of a[0] ... a[3]:\n    %p %p %p %p\n",
    a[0], a[1], a[2], a[3]);

printf("Addresses of row 0: %p %p %p\n",
    &a[0][0], &a[0][1], &a[0][2]);
printf("Addresses of row 1: %p %p %p\n",
    &a[1][0], &a[1][1], &a[1][2]);
printf("Addresses of row 2: %p %p %p\n",
    &a[2][0], &a[2][1], &a[2][2]);
printf("Addresses of row 3: %p %p %p\n",
    &a[3][0], &a[3][1], &a[3][2]);

printf("Scan entire array:\n");
for (p = *a; *p != 32; printf("%d ", *p), p++) ;
printf("%d\n", *p);
}
```

```
OUTPUT:

Addresses of a:
   0x22ee90
Addresses of a[0] ...  a[3]:
   0x22ee90 0x22ee9c 0x22eea8 0x22eeb4
Contents of a[0] ...  a[3]:
   0x22ee90 0x22ee9c 0x22eea8 0x22eeb4 ...  WHY?!?

Addresses of row 0:  0x22ee90 0x22ee94 0x22ee98
Addresses of row 1:  0x22ee9c 0x22eea0 0x22eea4
Addresses of row 2:  0x22eea8 0x22eeac 0x22eeb0
Addresses of row 3:  0x22eeb4 0x22eeb8 0x22eebc

Scan entire array:
0 1 2 10 11 12 20 21 22 30 31 32
```

Strings

A string is an array of `char` terminated with the NULL char '\0'.

- `char s[] = "abc";`
- `char *s = "abc";`
- ```
  char s[5];
  s[0] = 'a'; s[1] = 'b'; s[2] = 'c'; s[3] = '\0';
  ```
- ```
  char s[5];
  strcpy(s, "abc");
  ```
- Not correct:
  ```
  char *s;
  strcpy(s, "abc");
  ```

Pitfalls:

- When a string pointer is used to modify a string, be aware that the memory location changed is valid.
- When declaring and initializing simultaneously eg: `char *s = "abc";` the memory locations for "abc" may NOT be valid for writing. (It is valid for `char s[] = "abc";` - why?)

Strings - examples

```
void strcpy2(char dest[], char src[]) {
    char *dp = &dest[0], *sp = &src[0];
    while(*sp != '\0')
       *dp++ = *sp++;
    *dp = '\0';
}

strcmp2(char *str1, char *str2) {
    char *p1 = &str1[0], *p2 = &str2[0];
    while(1) {
        if(*p1 != *p2)
            return *p1 - *p2;
        if(*p1 == '\0' || *p2 == '\0') return 0;
        p1++;
        p2++;
    }
}
```

NOTE: This precisely defines the meaning of the `strcmp` function in `string.h`

Structures

A structure is a construct that can group together variables of *different* types.
Eg. an "employee" record:

```
    struct emp {
        char lname[20];    /* last name */
        char fname[20];    /* first name */
        int age;           /* age */
        float rate;        /* e.g. $12.75 per hour */
    };
```

To obtain one struct variable:

```
struct emp my_struct;
int main(void)
{
    strcpy(my_struct.lname, "Jaffar");
    strcpy(my_struct.fname, "Joxan");
    my_struct.age = 21;
    my_struct.rate = 999;
}
```

Structures

Structures can be arbitrarily complicated:

```
struct date {
    short day, month, year;
}
struct emp {
    char lname[20];    /* last name */
    char fname[20];    /* first name */
    int age;           /* age */
    float rate;        /* e.g. $12.75 per hour */
    struct date starting_date;
    struct date previous_raise;
    int last_percent_increase;
    struct emp *manager; /* makes emp a SELF-REFERENTIAL struct */
}
```

NOTE: Wrong to use `struct emp manager;` to self-reference. Why?

```
struct emp joxan;
struct emp wong;

... code to fill up variables joxan and wong ...

printf("%s\n", (joxan.manager)->lname);
```

Structure Assignment/Copy

```
struct emp { char name[20]; float rate; };
struct emp2 { char *name;  float rate; };

struct emp joxan, wong;
struct emp2 joxan2, wong2;

int main(void) {
char str[] = "joxan2";
    strcpy(joxan.name, "joxan"); joxan.rate = 111;
    joxan2.name = str; joxan2.rate = 222;
    changename(joxan); changename2(joxan2);
    printf("%s %s\n", joxan.name, joxan2.name);
}
void changename(struct emp x) {
    strcpy(x.name, "wong");
}
void changename2(struct emp2 x) {
    strcpy(x.name, "wong2");
}
```

Output: joxan wong2

NOTE: copying a structure copies all the space allocated to the structure. In the structure `emp`, the attribute `name` is an array with allocated space 20 chars. In the structure `emp2`, the attribute `name` is allocated a *single* pointer.

Array of Structures

```
struct emp {
    char lname[20];    /* last name */
    char fname[20];    /* first name */
    int age;           /* age */
    float rate;        /* e.g. $12.75 per hour */
} database[1000];

main() {
    printf("%d %d %d\n",
        sizeof(struct emp), database, database + 1);
}
```

Output: 48 4206624 4206672

NOTE: pointer increment +1 translates into an address increment of 48.