# 6CS005 High Performance Computing

## Workshop 1: Revision of C Programs

Jnaneshwar Bohara

University of Wolverhampton

Herald College, Kathmandu

These programs are an accompaniment to 6CS005 HPC. Even though the task of analyzing them is not assessed you should still do it because later in the module assumptions will be made about you understanding certain aspects of these programs. You should analyze each program in turn and do the following before moving on to the next program.

- Analyze and run the program and look up anything that you do not understand. This should primarily be done using a text book such as K&R. If you still can't understand it, post a query on Google Classroom. If this does not resolve things, bring the problem to the next class.

- For your reference, they are available in google classroom.

hello01.c.html

Not much different to Java! The #include is a bit like a Java import and lets the program know in this case know about standard input and output (stdio) functions, such as printf.

```
1: #include <stdio.h>
2:
3: int main(){
4:   printf("hello world!!\n");
5:   return 0;
6: }
7:
```

---

Output

```
hello world!!
```

hello02.c.html

As well as storing details of functions the files used in includes, called header files, can also store other definitions and constants. In this case EXIT_SUCCESS is defined in stdlib.h. Try typing *grep EXIT /usr/include/stdlib.h* to find all occurrences of EXIT in that header file and hence find out what a successful main method is expected to return.

```
1: #include <stdlib.h>
2: #include <stdio.h>
3:
4: int main() {
5:   int n = 19;
6:   printf("Hello\n My favorite number is %d\n", n);
7:   return EXIT_SUCCESS;
8: }
9:
```

Output

```
Hello
My favorite number is 19
```

control01.c.html

```
01: #include <stdlib.h>
02: #include <stdio.h>
03:
04: int main() {
05:   int i;
06:   for(i=0;i<5;i++){
07:     printf("%d,", i);
08:   }
09:   printf("\n");
10:
11:   while(i<10){
```

```
12:     printf("%d,", i);
13:     i++;
14:   }
15:
16:   do {
17:     printf("%d,", i);
18:     i++;
19:   } while(i<15);
20:   printf("\n");
21:
22:   if(i>13){
23:     printf("custard\n");
24:   } else {
25:     printf("gravy\n");
26:   }
27:
28:   return EXIT_SUCCESS;
29: }
30:
```

---

Output

```
0,1,2,3,4,
5,6,7,8,9,10,11,12,13,14,
custard
```

---

types01.c.html

C integers are just like Java integers.

```
01: #include <stdlib.h>
02: #include <stdio.h>
03:
04: int main() {
05:   int x = 10;
06:   int y = 3;
```

```
07:
08:    printf("%d / %d = %d\n", x, y, x/y);
09:    return EXIT_SUCCESS;
10: }
11:
```

---

Output

```
10 / 3 = 3
```

---

types02.c.html

Long integers are twice the size of integers.

```
01: #include <stdlib.h>
02: #include <stdio.h>
03:
04: int main() {
05:    long int x = 10L;
06:    long int y = 3L;
07:
08:    printf("%ld / %ld = %ld\n", x, y, x/y);
09:    return EXIT_SUCCESS;
10: }
11:
```

---

Output

```
10 / 3 = 3
```

types03.c.html

C floats are just like Java floats. The printf formatters used with them are very similar to Java. Try replacing %f with %5.2f

```
01: #include <stdlib.h>
02: #include <stdio.h>
03:
```

```
04: int main() {
05:    float x = 10.0f;
06:    float y = 3.0f;
07:
08:    printf("%f / %f = %f\n", x, y, x/y);
09:    return EXIT_SUCCESS;
10: }
11:
```

---

Output

```
10.000000 / 3.000000 = 3.333333
```

types04.c.html

C doubles are like Java doubles.

```
01: #include <stdlib.h>
02: #include <stdio.h>
03:
04: int main() {
05:    double x = 10.0;
06:    double y = 3.0;
07:
08:    printf("%lf / %lf = %lf\n", x, y, x/y);
09:    return EXIT_SUCCESS;
10: }
11:
```

---

Output

```
10.000000 / 3.000000 = 3.333333
```

types05.c.html

There are no Booleans in C, but integers are used by conditional operators instead. Work through the code and its output and see if you can work out what integer values are used to represent true and false.

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
  int a = 2;
  int b = 3;
  int c = 2;
  int d = 4;

  printf("There are no booleans in c\n");
  printf("%d\n", a==b);
  printf("%d\n", a==c);
  printf("%d\n", a!=b);
  printf("%d\n", a!=c);
  printf("%d\n", a==b);
  printf("%d\n", !(a==b));

  int e = (a == b) || (a == c);
  int f = (a == b) && (a == c);

  printf("e=%d\n", e);
  printf("f=%d\n", f);

  if(e) {
    printf("e=true\n");
  } else {
    printf("e=false\n");
  }

  if(f) {
    printf("f=true\n");
  } else {
    printf("f=false\n");
  }
```

```
35:
36:    return EXIT_SUCCESS;
37: }
38:
```

---

Output

```
There are no booleans in c
0
1
1
0
0
1
e=1
f=0
e=true
f=false
```

---

types06.c.html

Strings are formed by placing a sequence of characters in adjacent memory locations and the end of a string of is marked with a zero byte. A string is just an array of characters (and an end marker).

```
01: #include <stdlib.h>
02: #include <stdio.h>
03:
04: int main() {
05:    printf("Strings are just arrays of chars\n");
06:
07:    char *message1 = "Hello";
08:    char *message2 = "Gyaneshwar";
09:
10:    printf("%s %s\n", message1, message2);
```

```
11:
12:          printf("Look   in   /usr/include/string.h   for
functions\n");
13:    printf("that can be applied. Each has a man page.\n");
14:    return EXIT_SUCCESS;
15: }
16:
```

---

Output

```
Strings are just arrays of chars
hello kevan
Look in /usr/include/string.h for functions
that can be applied. Each has a man page.
```

---

memory01.c.html

The sizeof function returns the number of bytes used by a variable or data type. This example proves that a long int is twice as big as a normal int.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: int main() {
05:    int x = 123;
06:    long int y = 321;
07:    printf("%d %ld\n",  x, sizeof(x));
08:    printf("%ld %ld\n", y, sizeof(y));
09:    return EXIT_SUCCESS;
10: }
11:
```

---

Output

```
123 4
```

```
321 8
```

---

memory02.c.html

Here we see a function that takes an integer as a parameter, increments it and returns the result.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: int inc(int w) {
05:   return w + 1;
06: }
07:
08: int main() {
09:   int x = 123;
10:   int y = inc(x);
11:   printf("%d,%d\n", x, y);
12:   return EXIT_SUCCESS;
13: }
14:
```

---

Output

```
123,124
```

---

memory03.c.html

Here we have an example of a function that increments the value of the passed to. This example uses **pointers**, which are C's way of doing low level addressing. On line 9 a variable called x is declared and the value 123 copied into it. On line 10 a variable called y is declared and is given a copy of the value stored in x. There are two distinct variables and there will be two copies of the number 123 in memory.

On line 11 we are not passing the y variable to the inc function, but are passing the address in memory of y. In this context & means find the address of. On line 4 the asterisk denotes that w is a pointer - it stores the address of a variable, not actually the variable's value.

The right hand side of the assignment on line 5 finds the value that is pointed to by w and adds one to it. The left hand side is saying store the result of the assignment in the memory location pointed to by w.

The result is as expected - the value stored in y has been incremented.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: void inc(int *w) {
05:   *w = *w + 1;
06: }
07:
08: int main() {
09:   int x = 123;
10:   int y = x;
11:   inc(&y);
12:   printf("%d,%d\n", x, y);
13:   return EXIT_SUCCESS;
14: }
15:
```

---

Output

```
123,124
```

---

memory04.c.html

If you briefly scan this program you may find the output is not what you expected. Initially x stores 123, but when we print it out we get the result 124. How can this be as there are no commands to change the value stored in x?

Line 9 declares a variable, x, and assigns it a value. Line 10 declares a pointer variable. This is not a normal integer variable that would store a value, but one that stores a pointer to an address where an integer value will be stored. At this point, the address pointed to by y is arbitary and must not be assigned to. Not so long ago it would have been easy to crash your computer by assigning a value to an unknown location as the location could be data used by another program or could actually be part of another running program. Operating

systems these days will trap such mistakes and will not result in *the blue screen of death*, but will cause your program to terminate.

**On line 11 we make the y pointer point to the address of the x variable. x and y will now effectively be referencing the same variable, so when the call to inc modifies the value pointed to by y it is really changing the x variable.**

**If you don't understand this example get your tutor or a colleage to draw you some diagrams.**

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: void inc(int *w) {
05:    *w = *w + 1;
06: }
07:
08: int main() {
09:    int x = 123;
10:    int *y;
11:    y = &x;
12:    inc(y);
13:    printf("%d,%d\n", x, *y);
14:    return EXIT_SUCCESS;
15: }
16:
```

---

Output

```
124,124
```

---

memory05.c.html

In this example there are no normal variables as we rely on *Dynamic Memory Allocation*. Using this technique we allocate and release memory at runtime using the malloc and free functions respectively. We often use this technique when we don't know how much memory we need before the program starts running so that we can allocate just the right amount when the amount required is known.

Line 9 defines a pointer, x, and points it to an area of memory returned by malloc. Malloc knows how much memory to assign because we have used size of to determine the required size - i.e. this program uses a single integer. The program then proceeds in the same way as the previous example, doing some arithmetic and output.

Just before the end of the program we have to give back the memory that we have borrowed using the free command. If we don't do this we will cause a *memory leak* - every time you run the program it will be allocated memory and not returned. If you run it enough times there will be no memory left to use for other things and you may have to restart your computer.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <malloc.h>
04:
05: void inc(int *w) {
06:    *w = *w + 1;
07: }
08:
09: int main() {
10:    int *x = malloc(sizeof(int));
11:    *x = 123;
12:    int *y;
13:    y = x;
14:    inc(y);
15:    printf("%d,%d\n", *x, *y);
16:    free(x);
17:    return EXIT_SUCCESS;
18: }
19:
```

---

Output

```
124,124
```

---

structs01.c.html

A struct is a bit like a Java class, except it does not have methods. This example a struct to combine together hours, minutes and seconds data to form a time. Note the use of -> when using a pointer struct data.

```
01: #include <stdio.h>
02:
03: struct t {
04:   unsigned int h;
05:   unsigned int m;
06:   unsigned int s;
07: };
08:
09: int main() {
10:   struct t a;
11:   struct t *b;
12:
13:   a.h = 5;
14:   a.m = 9;
15:   a.s = 45;
16:
17:   printf("Time a is %u:%02u:%02u\n", a.h, a.m, a.s);
18:
19:   b = &a;
20:   printf("Time b is %u:%02u:%02u\n", b->h, b->m, b->s);
21:
22:   return 0;
23: }
24:
```

---

Output

```
Time a is 5:09:45
Time b is 5:09:45
```

---

structs02.c.html

Here we define a new type called t which is formed from a structure. Look at the declaration of the a and b variables and note that t can now be used just like the standard data types.

```
01: #include <stdio.h>
02:
03: typedef struct {
04:   unsigned int h;
05:   unsigned int m;
06:   unsigned int s;
07: } t;
08:
09: int main() {
10:   t a;
11:   t *b;
12:
13:   a.h = 5;
14:   a.m = 9;
15:   a.s = 45;
16:
17:   printf("Time a is %u:%02u:%02u\n", a.h, a.m, a.s);
18:
19:   b = &a;
20:   printf("Time b is %u:%02u:%02u\n", b->h, b->m, b->s);
21:
22:   return 0;
23: }
24:
```

---

Output

```
Time a is 5:09:45
Time b is 5:09:45
```

---

structs03.c.html

He we have struct data that is dynamically allocated.

```
01: #include <stdio.h>
02: #include <malloc.h>
03:
04: typedef struct {
05:   unsigned int h;
06:   unsigned int m;
07:   unsigned int s;
08: } t;
09:
10: int main() {
11:   t *a;
12:
13:   a = (t *) malloc(sizeof(t));
14:
15:   a->h = 5;
16:   a->m = 9;
17:   a->s = 45;
18:
19:   printf("Time a is %u:%02u:%02u\n", a->h, a->m, a->s);
20:
21:   free(a);
22:   return 0;
23: }
24:
```

---

Output

```
Time a is 5:09:45
```

---

memory06.c.html

This example shows a struct being declared and its size being calculated. Is the size what you expected?

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: struct pair {
05:   int a;
06:   int b;
07: };
08:
09: int main() {
10:   struct pair x;
11:   x.a = 12;
12:   x.b = 34;
13:   printf("%d,%d,%ld\n", x.a, x.b, sizeof(struct pair));
14:   return EXIT_SUCCESS;
15: }
16:
```

---

Output

```
12,34,8
```

---

memory07.c.html

This example if functionally the same as the previous but shows a type definition being used.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: typedef struct  {
05:   int a;
06:   int b;
07: } pair;
08:
```

```
09: int main() {
10:   pair x;
11:   x.a = 12;
12:   x.b = 34;
13:   printf("%d,%d,%ld\n", x.a, x.b, sizeof(pair));
14:   return EXIT_SUCCESS;
15: }
16:
```

---

Output

```
12,34,8
```

---

memory08.c.html

In this example we have a new increment function that increments both integers stored in a pair struct. If you look at lines 10 and 11 you will see the syntax that we use to access elements of a structure that is being pointed to.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: typedef struct {
05:   int a;
06:   int b;
07: } pair;
08:
09: void inc(pair *w) {
10:   w->a = w->a + 1;
11:   w->b = w->b + 1;
12: }
13:
14: int main() {
15:   pair x;
16:   x.a = 12;
```

```
17:    x.b = 34;
18:    inc(&x);
19:    printf("%d,%d\n", x.a, x.b);
20:    return EXIT_SUCCESS;
21: }
22:
```

---

Output

```
13,35
```

---

memory09.c.html

This example is similar to the previous. The difference is that the pair variable data is dynamically allocated.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <malloc.h>
04:
05: typedef struct {
06:    int a;
07:    int b;
08: } pair;
09:
10: void inc(pair *w) {
11:    w->a = w->a + 1;
12:    w->b = w->b + 1;
13: }
14:
15: int main() {
16:    pair *x;
17:    x = malloc(sizeof(pair));
18:    x->a = 12;
19:    x->b = 34;
20:    inc(x);
```

```
21:    printf("%d,%d\n", x->a, x->b);
22:    free(x);
23:    return EXIT_SUCCESS;
24: }
25:
```

---

Output

```
13,35
```

---

memory10.c.html

Here we allocate memory for 10 variables at the same time and access them in pretty much the same way as arrys in Java. A difference lies in the level of support and protection for working with arrays in C. There is no .length() method to find out the size of an array and you can use large subscripts to try to access addresses that have not been allocated to you thus corrupting things already in memory.

On line 10 we allocate memory for 10 integers. The actual number of integers to allocate is stored in the global variable, n, because we will need to look this up later to control the loops initiated on lines 15 and 19. Lines 12 and 13 demonstrate that sizeof cannot be used to determine the size of an array and the output given tells us the size of an integer an the size of a pointer.

Lines 16 and 20 show how to store and retrieve elements of an array.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <malloc.h>
04:
05: int n = 10;
06:
07: int main() {
08:    int i;
09:    int *x;
10:    x = malloc(sizeof(int) * n);
11:
```

```
12:    printf("%ld\n", sizeof(x));
13:    printf("%ld\n", sizeof(*x));
14:
15:    for(i=0;i<n;i++){
16:      x[i] = 2 * i;
17:    }
18:
19:    for(i=0;i<n;i++){
20:      printf("%d,%d\n", i, x[i]);
21:    }
22:
23:    free(x);
24:    return EXIT_SUCCESS;
25: }
26:
```

---

Output

```
8
4
0,0
1,2
2,4
3,6
4,8
5,10
6,12
7,14
8,16
9,18
```

---

memory11.c.html

This example serves as an introduction to *Pointer Arithmetic*. It does exactly the same as the previous example but does not use array notation.

On line 10, memory of 10 integers is allocated and x is pointed to the start of it. A second pointer, y, is then defined and pointed to the same location. On line 14 the memory location pointed to by 8 is assigned a value as a result of some arithmetic. Next comes the pointer arithmetic on line 15 which basically moves the pointer along to the next integer location, i.e. by 4 bytes. As the loop iterates we populate memory with a sequence of values as we did with the array example. The second program loop uses a similar technique to traverse the integers and print out their values, but before this can be done y needs to be pointed back to the beginning of the data (line 18).

As with array techniques, you can use pointer arithmetic to try to access memory locations that you should not and with potentially hazardous outcomes.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <malloc.h>
04:
05: int n = 10;
06:
07: int main() {
08:   int i;
09:   int *x, *y;
10:   x = malloc(sizeof(int) * n);
11:   y = x;
12:
13:   for(i=0;i<n;i++){
14:     *y = 2 * i;
15:     y++;
16:   }
17:
18:   y = x;
19:
20:   for(i=0;i<n;i++){
21:     printf("%d,%d\n", i, *y);
22:     y++;
23:   }
24:
```

```
25:    free(x);
26:    return EXIT_SUCCESS;
27: }
28:
```

---

Output

```
0,0
1,2
2,4
3,6
4,8
5,10
6,12
7,14
8,16
9,18
```

---

memor y12.c.html

This example shows that array notation and pointer arithmetic can be used interchangeably. The main method uses a pointer technique to assign values and perform output, whilst the inc function uses array notation to increment all allocated integers.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <malloc.h>
04:
05: int n = 10;
06:
07: void inc(int *w) {
08:    int i;
09:    for(i=0;i<n;i++){
10:      w[i] = w[i] + 1;
11:    }
12: }
```

```
13:
14: int main() {
15:   int i;
16:   int *x, *y;
17:   x = malloc(sizeof(int) * n);
18:   y = x;
19:
20:   for(i=0;i<n;i++){
21:     *y = 2 * i;
22:     y++;
23:   }
24:
25:   inc(x);
26:   y = x;
27:
28:   for(i=0;i<n;i++){
29:     printf("%d,%d\n", i, *y);
30:     y++;
31:   }
32:
33:   free(x);
34:   return EXIT_SUCCESS;
35: }
36:
```

---

Output

```
0,1
1,3
2,5
3,7
4,9
5,11
6,13
```

```
7,15
8,17
9,19
```

---

memory13.c.html

The only difference between this example and the previous is that the inc method used pointer arithmetic to update the integer data

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <malloc.h>
04:
05: int n = 10;
06:
07: void inc(int *w) {
08:   int i;
09:   for(i=0;i<n;i++){
10:     *w = *w + 1;
11:     w++;
12:   }
13: }
14:
15: int main() {
16:   int i;
17:   int *x, *y;
18:   x = malloc(sizeof(int) * n);
19:   y = x;
20:
21:   for(i=0;i<n;i++){
22:     *y = 2 * i;
23:     y++;
24:   }
25:
26:   inc(x);
27:   y = x;
```

```
28:
29:    for(i=0;i<n;i++){
30:      printf("%d,%d\n", i, *y);
31:      y++;
32:    }
33:
34:    free(x);
35:    return EXIT_SUCCESS;
36: }
37:
```

---

Output

```
0,1
1,3
2,5
3,7
4,9
5,11
6,13
7,15
8,17
9,19
```

---

memory14.c.html

This example produces the same results as the previous but has been refactored so that the program has been decomposed into several small functions. The program is now easier to read.

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <malloc.h>
04:
05: int n = 10;
06:
```

```
07: void initialise(int *w) {
08:   int i;
09:   for(i=0;i<n;i++){
10:     *w = 2 * i;
11:     w++;
12:   }
13: }
14:
15: void inc(int *w) {
16:   int i;
17:   for(i=0;i<n;i++){
18:     *w = *w + 1;
19:     w++;
20:   }
21: }
22:
23: void output(int *w) {
24:   int i;
25:   for(i=0;i<n;i++){
26:     printf("%d,%d\n", i, w[i]);
27:   }
28: }
29:
30: int main() {
31:   int *x;
32:   x = malloc(sizeof(int) * n);
33:
34:   initialise(x);
35:   inc(x);
36:   output(x);
37:
38:
39:   free(x);
40:   return EXIT_SUCCESS;
```

```
41: }
42:
```

---

Output

```
0,1
1,3
2,5
3,7
4,9
5,11
6,13
7,15
8,17
9,19
```

---