Strings and Clases

BIL104E: Introduction to Scientific and Engineering Computing

Lecture 9

- Allocating Memory
- More Data Types and Functions

Functions in C

- The malloc() function
- The calloc() function
- The realloc() function
- The free() function

Allocating Memory at Runtime

There are many cases when you do not know the exact sizes of arrays used in your programs, until much later when your programs are actually being executed. You can specify the sizes of arrays in advance, but the arrays can be too small or too big if the numbers of data items you want to put into the arrays change dramatically at runtime.

Fortunately, C provides you with four dynamic memory allocation functions that you can employ to allocate or reallocate certain memory spaces while your program is running. Also, you can release allocated memory storage as soon as you don't need it. These four C functions, malloc(), calloc(), realloc(), and free(), are introduced in the following sections.

What Is a Function?

- *A function is named:* Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as calling the function. A function can be called from within another function.
- A function is independent: A function can perform its task without interference from or interfering with other parts of the program.
- A function performs a specific task: This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root.
- A function can return a value to the calling program: When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program.

Functions in C

```c
/* Demonstrates a simple function */
#include <stdio.h>

long cube(long x);

long input, answer;

main()
{
printf("Enter an integer value: ");
scanf("%d", &input);
answer = cube(input);
/* Note: %ld is the conversion specifier for */
/* a long integer */
printf("\nThe cube of %ld is %ld.\n", input, answer);

return 0;
}

/* Function: cube() - Calculates the cubed value of a variable */
long cube(long x)

long x_cubed;
{
x_cubed = x * x * x;
return x_cubed;
}
```

Declaration of a function

Function call

Function

Functions in C

```c
/* Making function calls */
#include <stdio.h>
int function_1(int x, int y);
double function_2(double x, double y)
{
printf("Within function_2.\n");
return (x - y);
}
main()
{
int x1 = 80;
int y1 = 10;
double x2 = 100.123456;
double y2 = 10.123456;
printf("Pass function_1 %d and %d.\n", x1, y1);
printf("function_1 returns %d.\n", function_1(x1, y1));
printf("Pass function_2 %f and %f.\n", x2, y2);
printf("function_2 returns %f.\n", function_2(x2, y2));
return 0;
}
/* function_1() definition */
int function_1(int x, int y)
{
printf("Within function_1.\n");
return (x + y);
}
```

```
Pass function_1 80 and 10.
Within function_1.
function_1 returns 90.
Pass function_2 100.123456. and 10.123456.
Within function_2.
function_2 returns 90.000000.
```

Functions with No Arguments :

The first case is a function that takes no argument. For instance, the C library function getchar() does not need any arguments. It can be used in a program like this:

```
int c;
c = getchar();
```

As you can see, the second statement is left blank between the parentheses (`(`) and (`)`) when the function is called.

In C, the declaration of the `getchar()` function can be something like this:

```
int getchar(void);
```

Local Variables:

You can declare variables within the body of a function. The term local means that the variables are private to that particular function and are distinct from other variables of the same name declared elsewhere in the program.

A local variable is declared like any other variable, using the same variable types and rules for names that you learned on Day 3. Local variables can also be initialized when they are declared. You can declare any of C's variable types in a function. Here is an example of four local variables being declared within a function:

```
int func1(int y)
{
int a, b = 10;
float rate;
double cost = 12.55;
/* function code goes here... */
```

Functions in C

```
1: /* 15L02.c: Functions with no arguments */
2: #include <stdio.h>
3: #include <time.h>
4:
5: void GetDateTime(void);
6:
7: main()
8: {
9: printf("Before the GetDateTime() function is called.\n");
10: GetDateTime();
11: printf("After the GetDateTime() function is called.\n");
12: return 0; 13: }
14: /* GetDateTime() definition */
15: void GetDateTime(void)
16: {
17: time_t now;
18:
19: printf("Within GetDateTime().\n");
20: time(&now);
21: printf("Current date and time is: %s\n",
22: asctime(localtime(&now)));
23: }
```

```
Before the GetDateTime() function is called.
Within GetDateTime().
Current date and time is: Sat Apr 05 11:50:10 1997
After the GetDateTime() function is called.
```

The Advantages of Structured Programming

- A related advantage of structured programming is the time you can save.
- If you write a function to perform a certain task in one program, you can quickly and easily use it in another program that needs to execute the same task.
- Even if the new program needs to accomplish a slightly different task, you'll often find that modifying a function you created earlier is easier than writing a new one from scratch.
- If your functions have been created to perform a single task, using them in other programs is much easier.

Making Function Calls

- When a function call is made, the program execution jumps to the function and finishes the task assigned to the function. Then the program execution resumes after the called function returns.

- A function call is an expression that can be used as a single statement or within other statements.

Functions in C

```c
#include <stdio.h>

int function_1(int x, int y);
double function_2(double x, double y){
   printf("Within function_2.\n");
   return (x - y);
}

main(){
   int x1 = 80;
   int y1 = 10;
   double x2 = 100.123456;
   double y2 = 10.123456;
   printf("Pass function_1  %d and %d.\n", x1, y1);
   printf("function_1 returns %d.\n", function_1(x1, y1));
   printf("Pass function_2  %f and %f.\n", x2, y2);
   printf("function_2 returns %f.\n", function_2(x2, y2));
   return 0;
}
/* function_1() definition */
int function_1(int x, int y)
{
   printf("Within function_1.\n");
   return (x + y);
}
```

```
Pass function_1  80 and 10.
Within function_1.
function_1 returns 90.
Pass function_2  100.123456. and 10.123456.
Within function_2.
function_2 returns 90.000000.
```

Returning a Value

To return a value from a function, you use the return keyword, followed by a C expression. When execution reaches a return statement, the expression is evaluated, and execution passes the value back to the calling program. The return value of the function is the value of the expression. Consider this function:

```
int func1(int var)
{
int x;
/* Function code goes here... */
return x;
}
```

When this function is called, the statements in the function body execute up to the return statement. The return terminates the function and returns the value of x to the calling program. The expression that follows the return keyword can be any valid C expression.

Functions in C

```
/* Demonstrates using multiple return statements in a function. */
#include <stdio.h>
int x, y, z;
int larger_of( int , int );
main()
{

     puts("Enter two different integer values: ");
  scanf("%d%d", &x, &y);
  z = larger_of(x,y);
  printf("\nThe larger value is %d.", z);
  return 0;
}

int larger_of( int a, int b)
{
if (a > b) return a; else
return b;
}
```

```
Enter two different integer
values:
200 300
The larger value is 300.
Enter two different integer
values:
300
200
The larger value is 300.
```

Passing Arrays to Functions

- The special relationship that exists in C between pointers and arrays has been discussed. This relationship comes into play when you need to pass an array as an argument to a function. **The only way you can pass an array to a function is by means of a pointer.**
- An argument is a value that the calling program passes to a function. It can be an int, a float, or any other simple data type, but it must be a **single** numerical value. It can be a single array element, but it can't be an entire array.
- What if you need to pass an entire array to a function? Well, you can have a pointer to an array, and that pointer is a single numeric value (the address of the array's first element). If you pass that value to a function, the function knows the address of the array and can access the array elements using pointer notation. Consider another problem. If you write a function that takes an array as an argument, you want a function that can handle arrays of different sizes. For example, you could write a function that finds the largest element in an integer array. The function wouldn't be much use if it were limited to dealing with arrays of one fixed size.

Functions in C

```c
/* Passing an array to a function. */

#include <stdio.h>

#define MAX 10

int array[MAX], count;

int largest(int x[], int y);

main()
{
    /* Input MAX values from the keyboard. */

    for (count = 0; count < MAX; count++)
    {
        printf("Enter an integer value: ");
        scanf("%d", &array[count]);
    }

    /* Call the function and display the return value. */
    printf("\n\nLargest value = %d\n", largest(array, MAX));

    return 0;
}
```

Functions in C

```c
/* Function largest() returns the largest value */
/* in an integer array */

int largest(int x[], int y)
{
    int count, biggest = -12000;

    for ( count = 0; count < y; count++)
    {
        if (x[count] > biggest)
            biggest = x[count];
    }

    return biggest;
}
```

```
Enter an integer value: 1
Enter an integer value: 2
Enter an integer value: 3
Enter an integer value: 4
Enter an integer value: 5
Enter an integer value: 10
Enter an integer value: 9
Enter an integer value: 8
Enter an integer value: 7
Enter an integer value: 6
Largest value = 10
```

Functions in C

The syntax for the time() function is

```
#include <time.h>
time_t time(time_t *timer);
```

The syntax for the localtime() function is

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

The syntax for the asctime() function is

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

Applying Pointers

Pointer Arithmetic

In C, you can move the position of a pointer by adding or subtracting integers to or from the pointer. For example, given a character pointer variable `ptr_str`, the following expression

```
ptr_str + 1
```

indicates to the compiler to move to the memory location that is one byte away from the current position of `ptr_str`.

Note *that for pointers of different data types, the integers* ***added*** *to or* ***subtracted*** *from the pointers have different scalar sizes. In other words, adding 1 to (or subtracting 1 from) a pointer is not instructing the compiler to add (or subtract) one byte to the address, but to adjust the address so that it skips over one element of the type of the pointer. You'll see more details in the following sections.*

Applying Pointers

The Scalar Size of Pointers

The general format to change the position of a pointer is

```
pointer_name + n
```

Here n is an integer whose value can be either positive or negative. pointer_name is the name of a pointer variable that has the following declaration:

```
data_type_specifier *pointer_name;
```

When the C compiler reads the `pointer_name + n` expression, it interprets the expression as

```
pointer_name + n * sizeof(data_type_specifier)
```

Applying Pointers

The Scalar Size of Pointers

Note that the sizeof operator is used to obtain the number of bytes that a specified data type can have. Therefore, for the char pointer variable **`ptr_str`**, the **`ptr_str + 1`** expression actually means

```
ptr_str + 1 * sizeof(char).
```

Because the size of a character is one byte long, ptr_str + 1 tells the compiler to move to the memory location that is 1 byte after the current location referenced by the pointer.

Applying Pointers

```c
/* Pointer arithmetic - Moving pointers of different data types */
#include <stdio.h>

main()
{
char *ptr_ch;
int *ptr_int;
double *ptr_db;
/* char pointer ptr_ch */ printf("Current position of ptr_ch: 0x%p\n", ptr_ch);
printf("The position after ptr_ch + 1: 0x%p\n", ptr_ch + 1);
printf("The position after ptr_ch + 2: 0x%p\n", ptr_ch + 2);
printf("The position after ptr_ch - 1: 0x%p\n", ptr_ch - 1);
printf("The position after ptr_ch - 2: 0x%p\n", ptr_ch - 2);
/* int pointer ptr_int */
printf("Current position of ptr_int: 0x%p\n", ptr_int);
printf("The position after ptr_int + 1: 0x%p\n", ptr_int + 1);
printf("The position after ptr_int + 2: 0x%p\n", ptr_int + 2);
printf("The position after ptr_int - 1: 0x%p\n", ptr_int - 1);
printf("The position after ptr_int - 2: 0x%p\n", ptr_int - 2);
/* double pointer ptr_ch */
printf("Current position of ptr_db: 0x%p\n", ptr_db);
printf("The position after ptr_db + 1: 0x%p\n", ptr_db + 1);
printf("The position after ptr_db + 2: 0x%p\n", ptr_db + 2);
printf("The position after ptr_db - 1: 0x%p\n", ptr_db - 1);
printf("The position after ptr_db - 2: 0x%p\n", ptr_db - 2);

return 0;
}
```

```
Current position of ptr_ch: 0x000B
The position after ptr_ch + 1: 0x000C
The position after ptr_ch + 2: 0x000D
The position after ptr_ch - 1: 0x000A
The position after ptr_ch - 2: 0x0009
Current position of ptr_int: 0x028B
The position after ptr_int + 1: 0x028D
The position after ptr_int + 2: 0x028F
The position after ptr_int - 1: 0x0289
The position after ptr_int - 2: 0x0287
Current position of ptr_db: 0x0128
The position after ptr_db + 1: 0x0130
The position after ptr_db + 2: 0x0138
The position after ptr_db - 1: 0x0120
The position after ptr_db - 2: 0x0118
```

Applying Pointers

Pointer Subtraction

For two pointers of the same type, you can subtract one pointer value from the other. For instance, given two char pointer variables, ptr_str1 and ptr_str2, you can calculate the offset between the two memory locations pointed to by the two pointers like this:

```
ptr_str2 - ptr_str1;
```

However, it's illegal in C to subtract one pointer value from another if they do not share the same data type.

Applying Pointers

```
/* Pointer subtraction - Performing subtraction on pointers */
#include <stdio.h>

main()
{
int *ptr_int1, *ptr_int2;

printf("The position of ptr_int1: 0x%p\n", ptr_int1);
ptr_int2 = ptr_int1 + 5;
printf("The position of ptr_int2 = ptr_int1 + 5: 0x%p\n", ptr_int2);
printf("The subtraction of ptr_int2 - ptr_int1: %d\n", ptr_int2 - Âptr_int1);
ptr_int2 = ptr_int1 - 5;
printf("The position of ptr_int2 = ptr_int1 - 5: 0x%p\n", ptr_int2);
printf("The subtraction of ptr_int2 - ptr_int1: %d\n", ptr_int2 - Âptr_int1);

return 0;
}
```

```
The position of ptr_int1: 0x0128
The position of ptr_int2 = ptr_int1 + 5: 0x0132
The subtraction of ptr_int2 - ptr_int1: 5
The position of ptr_int2 = ptr_int1 - 5: 0x011E
The subtraction of ptr_int2 - ptr_int1: -5
```

Pointers and Arrays

As indicated in previous lessons, pointers and arrays have a close relationship. You can access an array through a pointer that contains the start address of the array. The following subsection introduces how to access array elements through pointers.

Applying Pointers

Accessing Arrays via Pointers

Because an array name that is not followed by a subscript is interpreted as a pointer to the first element of the array, you can assign the start address of the array to a pointer of the same data type; then you can access any element in the array by adding a proper integer to the pointer. The value of the integer is the same as the subscript value of the element that you want to access.

In other words, given an array, array, and a pointer, ptr_array, if array and ptr_array are of the same data type, and ptr_array contains the start address of the array, that is

```
ptr_array = array;
```

then the expression array[n] is equivalent to the expression

```
*(ptr_array + n)
```

Applying Pointers

```c
/* Accessing arrays via pointers - Accessing arrays by using pointers */
#include <stdio.h>

main()
{
char str[] = "It's a string!";
char *ptr_str;
int list[] = {1, 2, 3, 4, 5};
int *ptr_int;

/* access char array */
ptr_str = str;
printf("Before the change, str contains: %s\n", str);
printf("Before the change, str[5] contains: %c\n", str[5]);
*(ptr_str + 5) = `A';
printf("After the change, str[5] contains: %c\n", str[5]);
printf("After the change, str contains: %s\n", str);
/* access int array */
ptr_int = list;
printf("Before the change, list[2] contains: %d\n", list[2]);
*(ptr_int + 2) = -3;
printf("After the change, list[2] contains: %d\n", list[2]);

return 0;
}
```

```
Before the change, str contains: It's a string!
Before the change, str[5] contains: a
After the change, str[5] contains: A
After the change, str contains: It's A string!
Before the change, list[2] contains: 3
After the change, list[2] contains: -3
```

Pointers and Functions

Before I talk about passing pointers to functions, let's first have a look at how to pass arrays to functions.

Passing Arrays to Functions

In practice, it's usually awkward if you pass more than five or six arguments to a function. One way to save the number of arguments passed to a function is to use arrays. You can put all variables of the same type into an array, and then pass the array as a single argument.

Applying Pointers

```c
/* 16L04.c: Passing arrays to functions - Passing arrays to functions */
#include <stdio.h>

int AddThree(int list[]);

main()
{
int sum, list[3];

printf("Enter three integers separated by spaces:\n");
scanf("%d%d%d", &list[0], &list[1], &list[2]);
sum = AddThree(list);
printf("The sum of the three integers is: %d\n", sum);

return 0;
}

int AddThree(int list[])
{
int i;
int result = 0;

for (i=0; i<3; i++)
result += list[i];
return result;
}
```

```
Enter three integers separated by spaces:
10 20 30
The sum of the three integers is: 60
```

Applying Pointers

Passing Pointers to Functions

As you know, an array name that is not followed by a subscript is interpreted as a pointer to the first element of the array. In fact, the address of the first element in an array is the start address of the array.

Therefore, you can assign the start address of an array to a pointer, and then pass the pointer name, instead of the unsized array, to a function.

Applying Pointers

```
/* Passing pointers to functions */
#include <stdio.h>

void ChPrint(char *ch);
int DataAdd(int *list, int max);
main()
{
char str[] = "It's a string!";
char *ptr_str;
int list[5] = {1, 2, 3, 4, 5};
int *ptr_int;

/* assign address to pointer */
ptr_str = str;
ChPrint(ptr_str);
ChPrint(str);

/* assign address to pointer */
ptr_int = list;
printf("The sum returned by DataAdd(): %d\n",
DataAdd(ptr_int, 5));
printf("The sum returned by DataAdd()
DataAdd(list, 5));
return 0;
}
```

```
Enter three integers separated by spaces:
10 20 30
The sum of the three integers is: 60
```

Applying Pointers

```
/* function definition */
void ChPrint(char *ch)
{
printf("%s\n", ch);
}
/* function definition */
int DataAdd(int *list, int max)
{
int i;
int sum = 0;

for (i=0; i<max; i++)
sum += list[i];
return sum;
}
```

```
It's a string!
It's a string!
The sum returned by DataAdd(): 15
The sum returned by DataAdd(): 15
```

Applying Pointers

Passing Multidimensional Arrays as Arguments

"Storing Similar Data Items," you learned about multidimensional arrays. In this section, you're going to see how to pass multidimensional arrays to functions.

As you might have guessed, passing a multidimensional array to a function is similar to passing a one-dimensional array to a function.

You can either pass the unsized format of a multidimensional array or a pointer that contains the start address of the multidimensional array to a function.

Applying Pointers

```
/* Passing multidimensional arrays to functions */
#include <stdio.h>
/* function declarations */
int DataAdd1(int list[][5], int max1, int max2);
int DataAdd2(int *list, int max1, int max2);
/* main() function */
main()
{
int list[2][5] = {1, 2, 3, 4, 5,
5, 4, 3, 2, 1};
int *ptr_int;

printf("The sum returned by DataAdd1(): %d\n",
DataAdd1(list, 2, 5));
ptr_int = &list[0][0];
printf("The sum returned by DataAd
DataAdd2(ptr_int, 2, 5));

return 0;
}
/* function definition */
int DataAdd1(int list[][5], int max1, int max2)
{
int i, j;
int sum = 0;

```

```
It's a string!
It's a string!
The sum returned by DataAdd(): 15
The sum returned by DataAdd(): 15
```

Applying Pointers

```
for (i=0; i<max1; i++)
for (j=0; j<max2; j++)
sum += list[i][j];
return sum;
}
/* function definition */
int DataAdd2(int *list, int max1, int max2)
{
int i, j; int sum = 0;

for (i=0; i<max1; i++)
for (j=0; j<max2; j++)
sum += *(list + i*max2 + j);
return sum;
}
```

```
The sum returned by DataAdd1(): 30
The sum returned by DataAdd2(): 30
```

Applying Pointers

Arrays of Pointers

In many cases, it's useful to declare an array of pointers and access the contents pointed to by the array by dereferencing each pointer. For instance, the following declaration declares an int array of pointers:

```
int *ptr_int[3];
```

In other words, the variable ptr_int is a three-element array of pointers to integers. In addition, you can initialize the array of pointers. For example:

```
int x1 = 10;
int x2 = 100;
int x3 = 1000;
ptr_int[0] = &x1;
ptr_int[1] = &x2;
ptr_int[2] = &x3;
```

Applying Pointers

Passing Multidimensional Arrays as Arguments

"Storing Similar Data Items," you learned about multidimensional arrays. In this section, you're going to see how to pass multidimensional arrays to functions.

As you might have guessed, passing a multidimensional array to a function is similar to passing a one-dimensional array to a function.

You can either pass the unsized format of a multidimensional array or a pointer that contains the start address of the multidimensional array to a function.