`sizeof` Operator

- C provides the special unary operator `sizeof` to determine the number of bytes used for storage of any data type, variable or constant as an integer during program compilation.

- When applied to a data type name, that name should be written in parantheses. Otherwise, the parantheses are not required.

Example:

```
/* Finding the size of each data type */
#include <stdio.h>
int main (void)
{
      printf("Size of (char) = %d\n", sizeof(char));
      printf("Size of (short) = %d\n", sizeof(short));
      printf("Size of (int) = %d\n", sizeof(int));
      printf("Size of (long) = %d\n", sizeof(long));
      printf("Size of (float) = %d\n", sizeof(float));
      printf("Size of (double) = %d\n", sizeof(double));
      printf("Size of (long double) = %d\n", sizeof(long double));
      return(0);
}
```

Output:

```
Size of (char) = 1
Size of (short) = 2
Size of (int) = 4
Size of (long) = 4
Size of (float) = 4
Size of (double) = 8
Size of (long double) = 8
```

- You did not learn some of these data types in CTIS 151. For instance, `short` and `long` can be used instead of `int`. Their differences are, `short` allows you to store whole numbers between **-32767** and **32767** because it uses only 2 bytes. `int` and `long` both allow you to store whole numbers between **-2147483647** and **2147483647** because they use 4 bytes. Their placeholders in `scanf` and `printf` statements are **%hd** and **%ld**, respectively.

- It is similar for `float`, `double` and `long double`. Their placeholders in `scanf` and `printf` statements are **%f**, **%lf** / **%f** and **%Lf**, respectively.

- The sizes of the data types may differ from machine to machine. For instance, some newer machines use 8-byte integers. Because of this reason, pointer arithmetic is also machine-dependent.

- When applied to a variable or constant, `sizeof` returns the size of the data type which that variable or constant belongs to.

Example:

```
int x;
double y;
char c;
printf("Size of x = %d \n", sizeof x);
printf("Size of y = %d \n", sizeof y);
printf("Size of c = %d \n", sizeof c);
printf("Size of 16 = %d \n", sizeof 16);
printf("Size of 1.6 = %d \n", sizeof 1.6);
```

<u>Output:</u>

```
Size of x = 4
Size of y = 8
Size of c = 1
Size of 16 = 4
Size of 1.6 = 8
```

- When applied to the name of an array, `sizeof` returns the total number of bytes in the array.

```
double arr[10];
int table[2][4];
printf("Size of arr = %d \n", sizeof arr);
printf("Size of table = %d \n", sizeof table);
```

<u>Output:</u>

```
Size of arr = 80
Size of table = 32
```

- What about the following?

```
printf("Size of *arr = %d \n", sizeof *arr);
printf("Size of table[0] = %d \n", sizeof table[0]);
printf("Size of *table = %d \n", sizeof *table);
printf("Size of **table = %d \n", sizeof **table);
```

<u>Output:</u>

```
Size of *arr = 8
Size of table[0] = 16
Size of *table = 16
Size of **table = 4
```

➢ *READ from Deitel & Deitel.*

Dynamic Memory Allocation

- *Dynamic memory allocation* is the ability for a program to obtain necessary memory space at execution time, and to release that space, which is no longer needed.

- Functions `malloc` and `free`, in stdlib library, and operator `sizeof`, are essential to dynamic memory allocation.

- Function `malloc` requires a single argument, a number which indicates the amount of memory space needed, and returns a pointer of type `void *` (pointer to void) to the allocated memory.

- Function `malloc` is normally used with the `sizeof` operator.

Example:

```
malloc (sizeof(int))
```

allocates exactly enough space to hold one type `int` value (4 bytes) and returns a pointer of type `void *` to the address of the block allocated.

- As you know, when we are dealing with pointers in C, we always deal with a pointer to some specific type, rather than simply a pointer. Therefore the data type (`void *`) of the value returned by `malloc` should always be cast to the specific type that we need.

Example:

```
int *p;
char *ch;
p = (int *) malloc (sizeof(int));
ch = (char *) malloc (sizeof(char));
*p = 3;
*ch = ’x’;
```

- If no memory is available, `malloc` returns a NULL pointer. So, it is a good programming style to test the result of it and give an error message if it is NULL (if the requested memory area couldn’t be allocated).

Example:

```
double *y;
y = (double *)malloc(sizeof(double));
if (y == NULL)
     printf (“No more memory!\n”);
else // The rest of the program
```

- Function `free` deallocates memory, i.e., the memory is returned to the system so that it can be reallocated in the future.

Example:

```
free(p);
free(ch);
free(y);
```

- After freeing a memory location, you are not allowed to refer it anymore.

```
printf("%d\n", *p);
```

will not display 3.

- When you declare an array as

```
int x[10];
```

`x` is automatically assigned a memory block of 40 bytes. However, if you declare it as

```
int *xp;
```

`xp` is not automatically assigned a memory block unless you assign the name of an array to it or assign sufficient memory to it with `malloc`:

```
xp = (int *)malloc(10 * sizeof(int));
```

- Now, you can store the even numbers 2 to 20 in `xp` as follows:

```
for (i = 0; i < 10; i++)
     *(xp + i) = 2 * i + 2;    // x[i] = 2 * i + 2;
```

➢ *READ Sec 12.3 from Deitel & Deitel.*

➢ *READ Sec 14.2 from Hanly & Koffman.*

- Remember the following example from the previous semester:

Example: Declare an array to hold IDs of the students taking a course and another array for their final grades.

- Since we don't know the number of the students taking that course, we should decide an optimum size for our arrays in order to be able to declare them:

```
int    std_id[100];
double final[100];
```

- If the number of the students taking the course is 25, 75x4=300 bytes are reserved unnecessarily.

- It would be nice to ask it to user first, and then use the input as the size of the arrays, but we are not allowed to use a variable as the array size during the array declaration.

- However, if we declare our arrays as pointers, we can use `malloc` to allocate just the necessary space for them, as follows:

```
int    *std_id, num_std;
double *final;

printf("How many students are taking the course? ");
scanf("%d", &num_std);

std_id = (int *)malloc(num_std * sizeof(int));
final = (double *)malloc(num_std * sizeof(double));
```

How to Dynamically Allocate a Two-Dimensional array in C

Assume that a two-dimensional array with 4 rows and 3 columns will be created dynamically and initialized as follows:

```
 5  10  15
20  25  30
35  40  45
50  55  60
```

Method 1:

Using a Single Pointer:

A simple way is to allocate memory block of size row*col and access elements using simple pointer arithmetic.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      int row = 4, col = 3;
      int *arr = (int*)malloc(row*col * sizeof(int));
      int r, c, count = 5;
      for (r = 0; r < row*col; r++) {
            *(arr + r) = count;
            count += 5;
      }
      for (r = 0; r < row; r++){
            for (c = 0; c < col; c++)
                  printf("%4d", *(arr + r * col + c));
            printf("\n");
      }
      free(arr);
      return(0);
}
```

Method 2:

Using Array of Pointers

We can create an array of pointers of size row. C language allows variable sized arrays. After creating an array of pointers, we can dynamically allocate memory for every row.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      int row = 4, col = 3;
      int **arr = (int**)malloc(row * sizeof(int*));
      int r, c, count = 5;

      for (r = 0; r < row; r++)
            arr[r] = (int*)malloc(col * sizeof(int));

      for (r = 0; r < row; r++)
            for (c = 0; c < col; c++) {
                  *(*(arr + r) + c) = count;
                  count += 5;
            }
      for (r = 0; r < row; r++) {
            for (c = 0; c < col; c++)
                  printf("%4d", *(*(arr + r) + c));
            printf("\n");
      }

      for (r = 0; r < row; r++)
            free(arr[r]);    //free all columns of each row

      free(arr);       //free all rows

      return(0);
}
```