malloc & sizeof

(pages 383-387 from Chapter 17)

1 Overview

Whenever you declare a variable in C, you are effectively reserving one or more locations in the computer's memory to contain values that will be stored in that variable. The C compiler automatically allocates the correct amount of storage for the variable. In some cases, the size of the variable may not be known until run-time, such as in the case of where the user enters how many data items will be entered, or when data from a file of unknown size will be read in. The functions **`malloc`** and **`calloc`** allow the program to dynamically allocate memory as needed as the program is executing.

2 `calloc` and `malloc`

The function `calloc` takes two arguments that specify the number of elements to be reserved, and the size of each element in bytes. The function returns a pointer to the beginning of the allocated storage area in memory, and the storage area is automatically set to 0.

The function `malloc` works similarly, except that it takes only a single argument – the total number of bytes of storage to allocate, and also doesn't automatically set the storage area to 0.

These functions are declared in the standard header file `<stdlib.h>`, which should be included in your program whenever you want to use these routines.

3 The `sizeof` Operator

To determine the size of data elements to be reserved by `calloc` or `malloc` in a machine-independent way, the `sizeof` operator should be used. The `sizeof` operator returns the size of the specified item in bytes. The argument to the `sizeof` operator can be a variable, an array name, the name of a basic data type, the name of a derived data type, or an expression.

A few examples:

`sizeof(int)` // would give the number of bytes needed to store an integer
`sizeof(x)` // if x was defined as an array of 100 integers, this would give the amount of storage required for 100 integers
`sizeof(struct dataEntry)` // would be the amount of storage required to store one `dataEntry` structure

Remember `sizeof` is an operator and not a function, even though it looks like a function. This operator is evaluated at compile time and not at run-time, unless a variable-length array is used as its argument. If such an array is not used, the compiler evaluates the size of the item and replaces it with the result of the operation, which is treated as a constant.

Use the `sizeof` operator wherever possible to avoid having to calculate and hard-code sizes into your programs.

4 Using `calloc` and `malloc`

If you want to allocate enough storage in your program to store 1,000 integers, you can call `calloc` as follows:

```
#include <stdio.h>
  …
int *intPtr;
  …
intPtr = (int *) calloc (sizeof (int), 1000);
```

Using `malloc`, the function call looks like this:

```
intPtr = (int *) malloc (sizeof(int) * 1000);
```

If you ask for more memory than the system has available, `calloc` (or `malloc`) returns a null pointer. You should test the pointer that is returned to ensure that the allocation succeeded, perhaps like the following:

```
if (intPtr == NULL)
{
      printf ("calloc failed \n");
      exit (EXIT_FAILURE);
}
```

If the allocation succeeds, the integer pointer variable `intPtr` can be used as if it were pointing to an array of 1000 integers. So, to set all 1000 elements to -1, you could write:

```
int p;
for (p = intPtr; p < intPtr + 1000; ++p)
      *p = -1;
```

To reserve storage for `n` elements of type `struct pixel`, you first need to define a pointer to the appropriate type:

```
struct pixel *pixelPtr;
```

and could then proceed to call the `malloc` function to reserve the appropriate amount of space:

```
pixelPtr = (struct pixel *) malloc (sizeof(struct pixel)*n);
```

After checking to make sure that `pixelPtr` is not NULL, it can be used in the normal fashion. For example, if the `pixel struct` has 3 elements, `r`, `g`, & `b`, you could assign a value to each of them, such as:

```
pixelPtr->r = 255;
pixelPtr->g = 0;
pixelPtr->b = 255;
```

5 The `free` Function

When you have finished working with the memory that has been dynamically allocated by `calloc` or `malloc`, you should give it back to the system by calling the `free` function, as in the following:

```
free (pixelPtr);
```

This will return the memory allocated by `calloc` or `malloc` provided that the value of `pixelPtr` still points to the ***beginning*** of the allocated memory.