Computer Science 1510

Lecture 28

Lecture Outline

- Dynamic memory allocation

Dynamic Memory Allocation

- Recall that dynamic memory allocation refers to the ability to request memory from the operating system at run-time.

- The use of dynamic memory avoids having to declare more memory than necessary to ensure that there is sufficient space to store data.

- A program can determine how much memory is required at run-time and allocate only that amount.

- As in Fortran, dynamic memory allocation in C makes use of pointers.

- Two functions that are used for dynamic memory allocation are called `malloc` and `calloc`.

- Both of these functions are declared in `stdlib.h`.

C: malloc and calloc

- Syntax:

```
pt=(type_cast*)calloc(quantity,size_of_element);
pt=(type_cast*)malloc(size_of_memory_required);
```

  where:

  - `pt` is a pointer,
  - `type_cast` is the type that the memory provided by the operating system will be changed to (`calloc` and `malloc` both return pointers of type `void*` which must be changed to the required type),
  - `quantity` is the number of elements required, each of a size equal to `size_of_element`,
  - `size_of_memory_required` is the number of bytes for the total amount of memory required.

- The difference between these two functions is that `calloc` initializes the memory to zero, while `malloc` does not initialize.

C: malloc and calloc

- Example:

```
int *ptc,*ptm;
ptc=(int*) calloc(10,sizeof(int));
ptm=(int*) malloc(10*sizeof(int));
```

- Both examples allocate space for 10 ints.

- If there is not enough memory to fulfill the allocation request, then both `malloc` and `calloc` will return `NULL` (ie. a null pointer).

- Example:

```
int *pt;
pt=(int*) calloc(10,sizeof(int));
if(pt==NULL){
  printf("Insufficient memory for calloc\n");
  return -1;
}
```

C: malloc and calloc

- Note that we have returned -1 to main rather than 0.

- A return value of 0 is generally used to indicate that the program executed successfully. Alternative return values could be used external to the main function to determine where things went wrong.

- To free (or deallocate) memory that has been allocated using `malloc` or `calloc`, we can use the `free` function,

  ```
  free(pt);
  ```

  where `pt` is the pointer that references the memory to be deallocated.

Type casting

- A variable of a given data type can be converted into another data type using an explicit *cast*.

- There is a possibility of losing information during such a conversion, such as when a float is cast as an int (the fractional portion is truncated).

- Type casting can be a useful technique. For example, to obtain the whole number portion of a real number.

Example: Type casting

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   float f,frac;
   int whole;
   printf("Please enter some random real number");
   printf(", include decimal part.\n");
   scanf("%f",&f);
   whole=(int)f; /* Cast f as an int */
   frac=f-whole; /* Compute the fractional part of f */
   printf("%f = %d + %f\n",f,whole,frac);
   return 0;
}
```

Sample output:

```
Please enter some random real number, include decimal part.
3.6
3.600000 = 3 + 0.600000
```

Example 1: calloc

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

int main(int argc, char *argv[]) {
  int i;
  int *array;
  array=(int*)calloc(10,sizeof(int));
  if (array==NULL){
    printf("Insufficient memory for allocation\n");
    return -1;
  }
  for(i=0;i<10;i++){
    scanf("%d",&array[i]);
  }
  for(i=0;i<10;i++){
    printf("%d\n",array[i]);
  }
  free(array);
  return 0;
}
```

Example 2: calloc

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

int main(int argc, char *argv[])
{
  int *top, *pt;
  int i,sum=0;

  top=(int*)calloc(10,sizeof(int));
  if (top==NULL){
    printf("Insufficient memory for allocation\n");
    return -1;
  }
  pt=top; /* Set pt to beginning of allocated memory */

  for(i=0;i<10;i++){
    scanf("%d",pt);  /* no & since pt is a pointer */
    pt++; /* Move to the next element */
  }
  pt=top; /* Reset pt to beginning of allocated memory */

  for(i=0;i<10;i++){
    sum+=*pt;
    pt++; /* Move to the next element */
  }
  printf("The sum of the elements is %d\n",sum);
  free(top);

  return 0;
}
```

Example 1: Dynamic memory allocation

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

/* The following program reads a list of failure times from a user
 * specified file, calculates the mean time to failure, and then prints
 * a list of failure times that are greater than the mean. */

int main(int argc, char *argv[])
{
   float *failure;
   int numtimes,i,ret;
   float sum, mean;
   char filename[20];
   FILE *fp;

   /* Get the filename from the user */
   printf("Please enter the name of the file to be read: ");
   scanf("%s",filename);

   /* Open the file */
   fp = fopen(filename,"r");
   if (fp==NULL) {
      printf("Unable to open file\n");
      return -1;
   }

   /* Get the number of failure times (located on first line of file) */
   ret = fscanf(fp,"%d",&numtimes);
   /* Check that the correct number of input items were
    * successfully assigned */
   if (ret != 1){
      printf("Error on read\n");
      return -2;
   }
```

```
   /* Allocate an array with numtimes elements to store the failure times */
   failure=(float*)calloc(numtimes,sizeof(float));
   if (failure==NULL) {
      printf("Unable to allocate memory\n");
      return -3;
   }

   /* Read the failure times and store them in array failure */
   for (i=0;i<numtimes;i++){
      ret = fscanf(fp,"%f",&failure[i]);
      /* Check that the correct number of input items were
       * successfully assigned */
      if (ret != 1){
         printf("Error on read\n");
         return -2;
      }
   }

   /* Calculate the mean time to failure */
   sum = 0.0;
   for (i=0;i<numtimes;i++){
      sum = sum + failure[i];
   }
   mean = sum/numtimes;
   printf("Mean time to failure = %f\n",mean);

   /* Print list of failure times greater than the mean */
   printf("List of failure times greater than the mean:\n");
   for (i=0;i<numtimes;i++){
      if (failure[i] > mean) printf("%f\n",failure[i]);
   }

   /* Deallocate the array of failure times */
   free(failure);

   return 0;
}
```

Example 2: Dynamic memory allocation

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

int main(int argc, char *argv[]){
    double failure, bin_size;
    int i, j, num_bins, num_pts, *bins, ret=0;
    FILE *fp;

    /* Check that the user entered the filename */
    if (argc!=2){
        printf("USAGE: %s filename\n",argv[0]);
        return -1;
    }

    /* Open the specified file */
    fp = fopen(argv[1],"r");
    if (fp==NULL){
        printf("Unable to open file %s\n",argv[1]);
        return -2;
    }

    /* Determine size of each bin */
    printf("How many bins would you like?\n");
    scanf("%d",&num_bins);
    printf("What should the bin size be?\n");
    scanf("%lf",&bin_size);

    /* Allocate space for the bins and initialize */
    bins = (int*)calloc(num_bins,sizeof(int));
    if (bins==NULL) {
        printf("Unable to allocate memory\n");
        return -3;
    }

    num_pts = 0;
```

```
    /* Read failure time and increment appropriate bin */
    ret = fscanf(fp,"%lf",&failure);
    while (ret != EOF) {   /* Read to end of file */
        num_pts++;
        for (i=num_bins-1;i>=0;i--) {
            if (failure >= i*bin_size) {
                bins[i]++;
                break;
            }
        }
        ret = fscanf(fp,"%lf",&failure);
    }

    /* Draw histogram */
    for (i=0;i<num_bins;i++) {
        if (i != num_bins-1) {
            printf("[%lf, %lf): ",i*bin_size,(i+1)*bin_size);
        } else {
            printf("[%lf, infinity): ",i*bin_size);
        }
        for (j=0;j<bins[i];j++){
            printf("*");
        }
        printf("\n");
    }

    free(bins);  /* Deallocate memory */
    fclose(fp);
    return 0;
}
```

Resizing dynamic memory

- C has a function which changes (usually increases) the amount of memory that was dynamically allocated.

- The function

  ```
  (type_cast*) realloc(ptr, size)
  ```

  resizes the memory block pointed to by `ptr` to `size`.

- It returns a pointer to the resized block (which could be `ptr` or a new pointer if it succeeds, and the `null` pointer if it fails. If it fails, the original memory block is unchanged.

- If possible, it allocates the new memory "in place" using adjacent unallocated memory. If this is not possible, since the dynamically allocated memory locations should be contiguous, a new block of the requested size is allocated, and the contents of the previous block copied to the new block.

Using realloc properly

- Consider the following code:

```
char *buffer;
.
.
buffer = (char*)malloc(buffer_size);
.
.
buffer = realloc(buffer, buffer_size*2);
```

This will actually work "most of the time" but if realloc fails, the original structure would be lost. The correct thing to do is to is:

```
temp = realloc(buffer, buffer_size*2);
if (temp==NULL)
   reportError();
buffer_size *= 2;
buffer = temp;
```

`realloc` **continued**

- If `realloc` succeeds, and has to actually move data to another block of memory, then the original block is freed by the `realloc` function.

- The reallocated block is usually given the same name as the original block (after testing for success), but this is not necessarily the case.

- If `realloc` is called with the `NULL` pointer as argument, it behaves exactly like `malloc()`.

- If the `size` is zero, and the pointer is not `NULL` then it behaves like `free`.

- If `size` is less than the original, the new memory block only holds `size` bytes of the original block.

- For non-character data types, the `sizeof(type)` operator is used to help determine the amount of memory allocated, in exactly the same was as the function `malloc()`.