Preview

- Process Environment in Linux
  - Process Termination
  - Environment List
  - Memory Layout of a C Program
  - Dynamic Memory Allocation and Deallocation
    - malloc()
    - calloc()
    - realloc()
    - free()
  - Environment Variables

Process Termination

- There are five ways for a process to terminate.
  - Normal Termination
    - Return from main function
    - Calling exit
    - Calling _exit (by child process)
  - Abnormal Termination
    - Calling abort
    - Terminated by a signal.

Process Termination

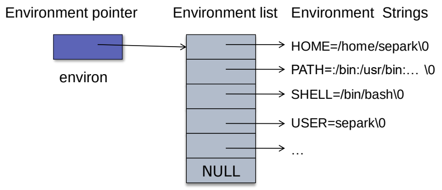
- `exit()` - performs certain cleaning up processing and control returns to kernel.
- `_exit()` – control immediately returns to kernel (used between child and parent processes).

```
#include <stdlib.h>
void exit (int status);

#include <unistd.h>
void _exit(int status);
```

Environment List

- Each program is passed an environment list.
- Like the argument list, the environment list is an array of pointer to c-string.
- Each pointer contains the address of null terminated character string.
- The global variable **environ** contains the address of the array pointers.

Environment List



Environment List

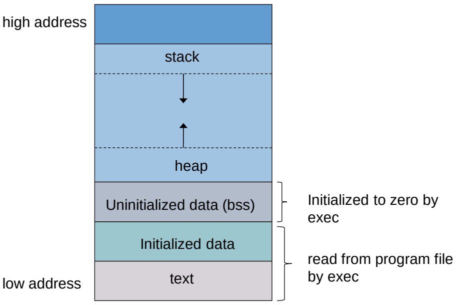
```
/* environ.c */
#include <stdio.h>
#include <unistd.h>

extern char **environ;

int main(int argc, char *argv[])
{
   char **p = environ;

   while (*p != NULL)
   {
     printf("%s (%p)\n", *p, *p);
     *p++;
   }

   return 0;
}
```

```
/* envp.c display environment strings
   with corresponding address */

#include <stdio.h>

int main(int argc, char *argv[], char
*envp[])
{
     char **p = envp;

    while (*p != NULL)
   {
       printf("%s (%p)\n", *p, *p);
      *p++;
   }

     return 0;
}
```

Memory Layout of a C Program

- A C program composed of five components.
  - **Text Segment** – The machine instruction sets, since it might be sharable (only one copy need to be in the memory)
  - **Initialized data Segment** – variable that are initialized in the program. Ex) int max =5;
  - **Uninitialized data Segment (bss)** – variable that are not initiated.
  - **Stack** – saved temporary variables when a function is called, also save the caller's environment value such as return address
  - **Heap** – used for dynamic memory allocation

Memory Layout of a C Program



Memory Layout of a C Program

- The size command reports the sized in bytes of the text data and uninitialized data(bss) segments.

Ex)

$size usr/bin/gcc /bin/sh

```
text      data     bss      dec      hex     file name
115651    1744     1140     118535   1cf07   /usr/bin/gcc
485881    8936     25360    520177   7eff1   /bin/sh
```

bss (Block Started by Symbol) is used by many compilers and linkers for a part of the data segment containing statically-allocated variables

Dynamic Memory Allocation

- There are three functions for memory allocation by C. (located in the heap)
  - **malloc()** – allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
  - **calloc()** – allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
  - **realloc()** – change the size of a previously allocated area.
  - **free()** – deallocates the space pointed to by ptr,

Dynamic Memory Allocation

```
void *malloc(size_t size)
```

size- size of the memory block in bytes

```
void *calloc(size_t nitems, size_t size)
```

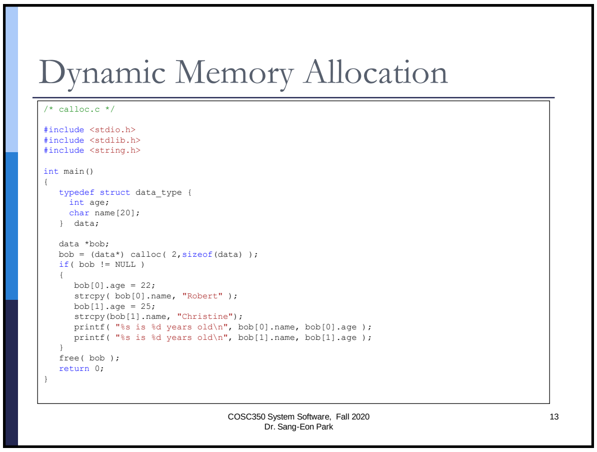
size- size of the memory block in bytes
nitem-the number of elements to be allocated

```
void *realloc(void *ptr, size_t size)
```

ptr =The pointer to a memory block previously allocated
size − This is the new size for the memory block, in bytes

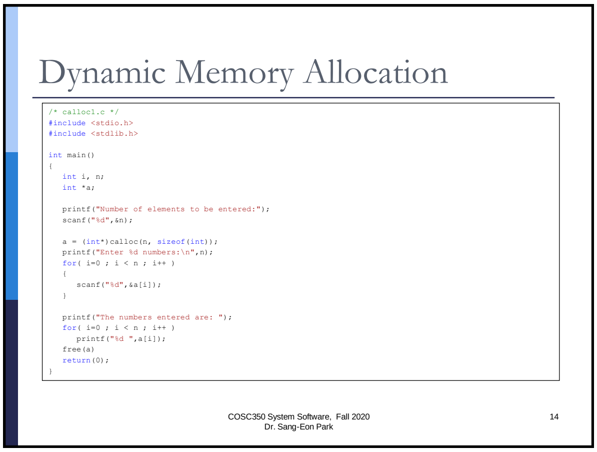
Dynamic Memory Allocation

```
/* malloc.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   typedef struct {
     int age;
     char name[20];
   }  data;

   data *bob;
   bob = (data*) malloc( sizeof(data) );
   if( bob != NULL )
   {
      bob->age = 22;
      strcpy( bob->name, "Robert" );
      printf( "%s is %d years old\n", bob->name, bob->age );
   }
   free( bob );
   return 0;
}
```



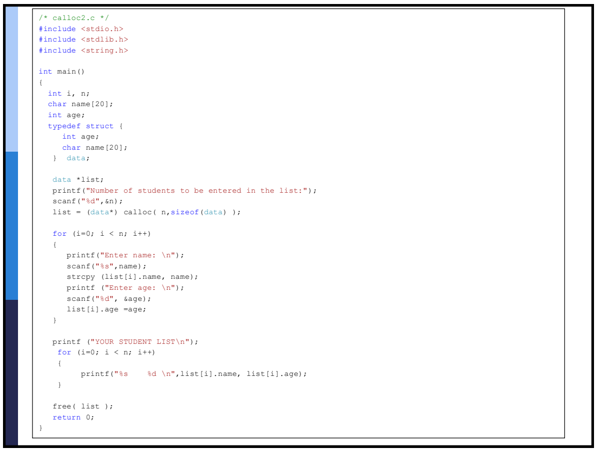
Dynamic Memory Allocation

```
/* calloc.c */

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

int main()
{
   typedef struct data_type {
     int age;
     char name[20];
   }  data;

   data *bob;
   bob = (data*) calloc( 2,sizeof(data) );
   if( bob != NULL )
   {
      bob[0].age = 22;
      strcpy( bob[0].name, "Robert" );
      bob[1].age = 25;
      strcpy(bob[1].name, "Christine");
      printf( "%s is %d years old\n", bob[0].name, bob[0].age );
      printf( "%s is %d years old\n", bob[1].name, bob[1].age );
   }
   free( bob );
   return 0;
}
```

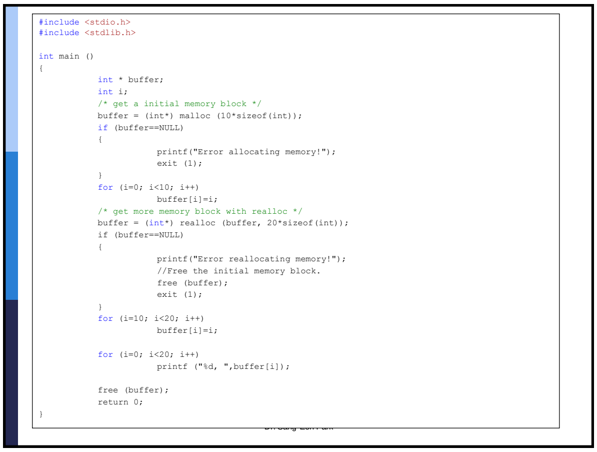


Dynamic Memory Allocation

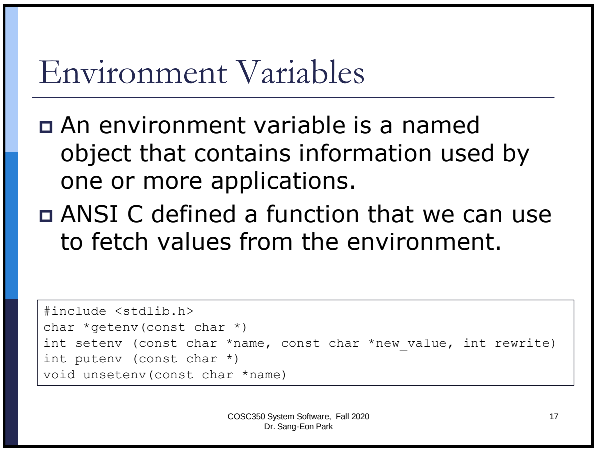
```
/* calloc1.c */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i, n;
   int *a;

   printf("Number of elements to be entered:");
   scanf("%d",&n);

   a = (int*)calloc(n, sizeof(int));
   printf("Enter %d numbers:\n",n);
   for( i=0 ; i < n ; i++ )
   {
      scanf("%d",&a[i]);
   }

   printf("The numbers entered are: ");
   for( i=0 ; i < n ; i++ )
      printf("%d ",a[i]);
   free(a)
   return(0);
}
```



```
/* calloc2.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  int i, n;
  char name[20];
  int age;
  typedef struct {
     int age;
     char name[20];
   }  data;

   data *list;
   printf("Number of students to be entered in the list:");
   scanf("%d",&n);
   list = (data*) calloc( n,sizeof(data) );

   for (i=0; i < n; i++)
   {
      printf("Enter name: \n");
      scanf("%s",name);
      strcpy (list[i].name, name);
      printf ("Enter age: \n");
      scanf("%d", &age);
      list[i].age =age;
   }

   printf ("YOUR STUDENT LIST\n");
    for (i=0; i < n; i++)
    {
        printf("%s    %d \n",list[i].name, list[i].age);
    }

   free( list );
   return 0;
}
```



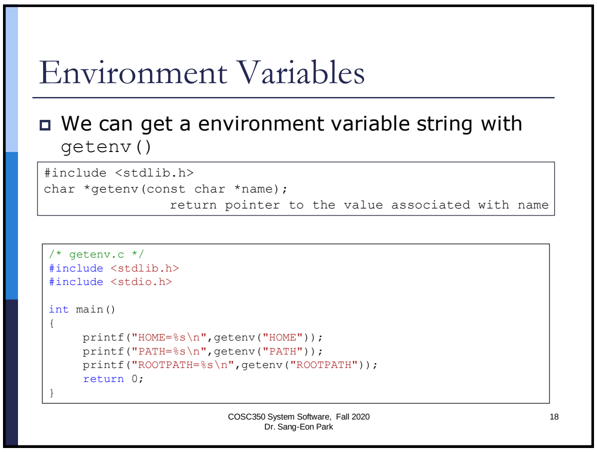
```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
            int * buffer;
            int i;
            /* get a initial memory block */
            buffer = (int*) malloc (10*sizeof(int));
            if (buffer==NULL)
            {
                        printf("Error allocating memory!");
                        exit (1);
            }
            for (i=0; i<10; i++)
                        buffer[i]=i;
            /* get more memory block with realloc */
            buffer = (int*) realloc (buffer, 20*sizeof(int));
            if (buffer==NULL)
            {
                        printf("Error reallocating memory!");
                        //Free the initial memory block.
                        free (buffer);
                        exit (1);
            }
            for (i=10; i<20; i++)
                        buffer[i]=i;

            for (i=0; i<20; i++)
                        printf ("%d, ",buffer[i]);

            free (buffer);
            return 0;
}
```



Environment Variables

- An environment variable is a named object that contains information used by one or more applications.
- ANSI C defined a function that we can use to fetch values from the environment.

```
#include <stdlib.h>
char *getenv(const char *)
int setenv (const char *name, const char *new_value, int rewrite)
int putenv (const char *)
void unsetenv(const char *name)
```

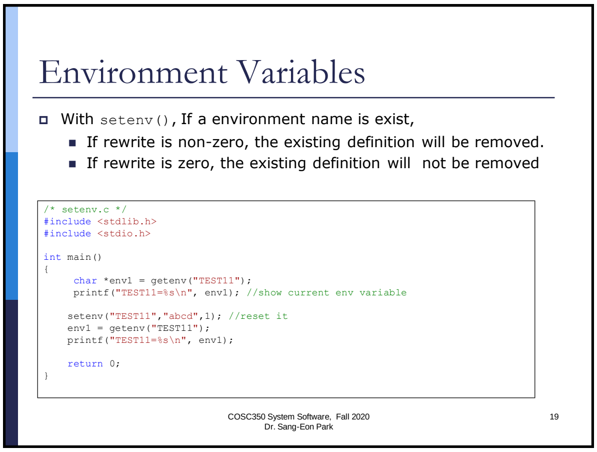


Environment Variables

- We can get a environment variable string with getenv()

```
#include <stdlib.h>
char *getenv(const char *name);
                  return pointer to the value associated with name
```

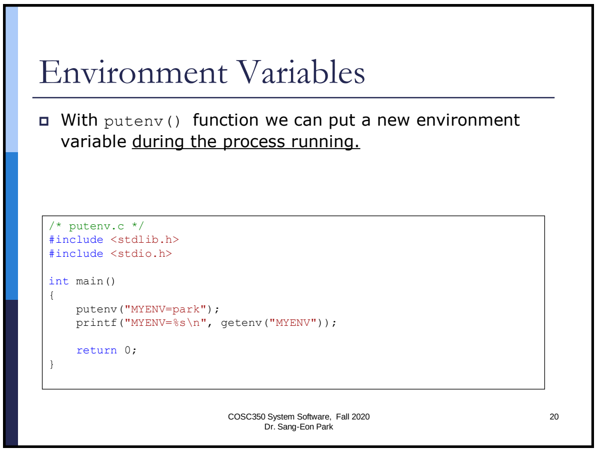
```
/* getenv.c */
#include <stdlib.h>
#include <stdio.h>

int main()
{
     printf("HOME=%s\n",getenv("HOME"));
     printf("PATH=%s\n",getenv("PATH"));
     printf("ROOTPATH=%s\n",getenv("ROOTPATH"));
     return 0;
}
```



Environment Variables

- With `setenv()`, If a environment name is exist,
  - If rewrite is non-zero, the existing definition will be removed.
  - If rewrite is zero, the existing definition will not be removed

```
/* setenv.c */
#include <stdlib.h>
#include <stdio.h>

int main()
{
     char *env1 = getenv("TEST11");
     printf("TEST11=%s\n", env1); //show current env variable

    setenv("TEST11","abcd",1); //reset it
    env1 = getenv("TEST11");
    printf("TEST11=%s\n", env1);

    return 0;
}
```



Environment Variables

- With `putenv()` function we can put a new environment variable during the process running.

```
/* putenv.c */
#include <stdlib.h>
#include <stdio.h>

int main()
{
    putenv("MYENV=park");
    printf("MYENV=%s\n", getenv("MYENV"));

    return 0;
}
```