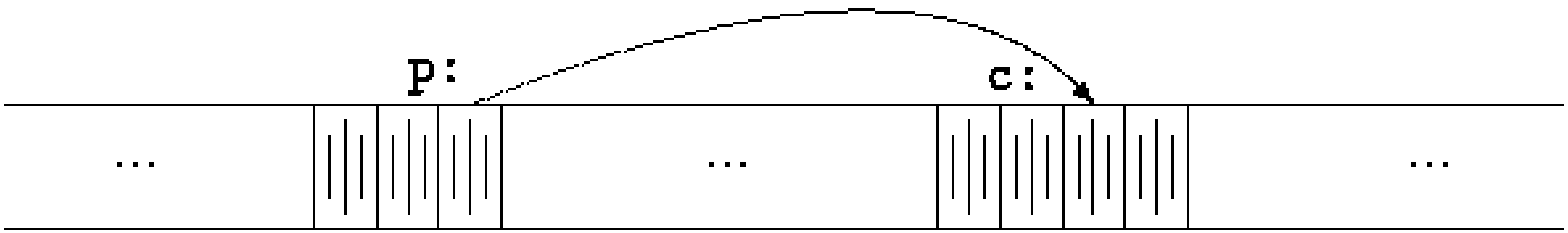
Pointers

CS102

Concept of Pointer



•In the above picture, c is a char and p is a pointer that points to it

•unary operator & gives the address of an object, so the statement
p = &c;

•p is said to ``point to'' c

•unary operator * is the *indirection or dereferencing operator,* when applied to a pointer, it accesses the object the pointer points to

Consider the example

```
#include <stdio.h>
void xyz(int *number) {
		*number = *number/2;
}
void xyz2(int number) {
		number = number/2;
}
int main() {
		int i = 8;
		int *ptr = &i;
		xyz(ptr);
		printf("The value of i is %d\n", i);
		xyz(&i);
		printf("The value of i is %d\n", i);
		xyz2(i);
		printf("The value of i is %d\n", i);
		return 0;
}
```

What is the output?

Using const qualifier with pointers

- const qualifier is used to make something READ ONLY
  - Constant pointers
  - Pointer to constant
  - Constant pointer to constant

Constant pointer

- A constant pointer cannot change the address that it is holding
- Declaration syntax
  - <type of pointer> *const <pointer name>
- Example
  - int *const ptr;

```
#include<stdio.h>
 int main(void)
{
    int var1 = 0, var2 = 0;
    int *const ptr = &var1;
    ptr = &var2;
    printf("%d\n", *ptr);
    return 0;
}
```

Constant pointer

- A constant pointer cannot change the address that it is holding
- Declaration syntax
  - <type of pointer> *const <pointer name>
- Example
  - int *const ptr;

```
#include<stdio.h>
 int main(void)
{
    int var1 = 0, var2 = 0;
    int *const ptr = &var1;
    ptr = &var2;
    printf("%d\n", *ptr);
    return 0;
}
```

Pointer to constant

- A pointer through which one cannot change the value of a variable it points
- Can change the address but not the value it points
- Declaration syntax
  - const <type of pointer> * <pointer name>
- Example
  - const int* ptr;

```
#include<stdio.h>
int main(void)
{
  int var1 = 0;
  const int* ptr = &var1;
  *ptr = 1;
   printf("%d\n", *ptr);
   return 0;
}
```

Pointer to constant

- A pointer through which one cannot change the value of a variable it points
- Can change the address but not the value it points
- Declaration syntax
  - const <type of pointer> * <pointer name>
- Example
  - const int* ptr;

```
#include<stdio.h>
int main(void)
{
   int var1 = 0;
   const int* ptr = &var1;
   *ptr = 1;
    printf("%d\n", *ptr);
    return 0;
}
```

Constant pointer to a constant

- It can neither change the address it is holding nor it can change the value kept at that address
- Declaration syntax
  - const <type of pointer> * const <pointer name>
- Example
  - const int* const ptr;

```
#include<stdio.h>
int main(void)
{
  int var1 = 0,var2 = 0;
const int* const ptr = &var1;
*ptr = 1;
  ptr = &var2;
  printf("%d\n", *ptr);
 return 0;
}
```

Constant pointer to a constant

- It can neither change the address it is holding nor it can change the value kept at that address
- Declaration syntax
  - const <type of pointer> * const <pointer name>
- Example
  - const int* const ptr;

```
#include<stdio.h>
int main(void)
{
  int var1 = 0,var2 = 0;
const int* const ptr = &var1;
*ptr = 1;
  ptr = &var2;
  printf("%d\n", *ptr);
 return 0;
}
```

Arrays & Pointers

- An array name stores the address of the first element of the array
- Pointers also store address of memory locations
- An array name is an address or pointer that is fixed but the values of pointer variables are not fixed
  - a[0] is equivalent to *a
  - a[i] is equivalent to *(a + i)

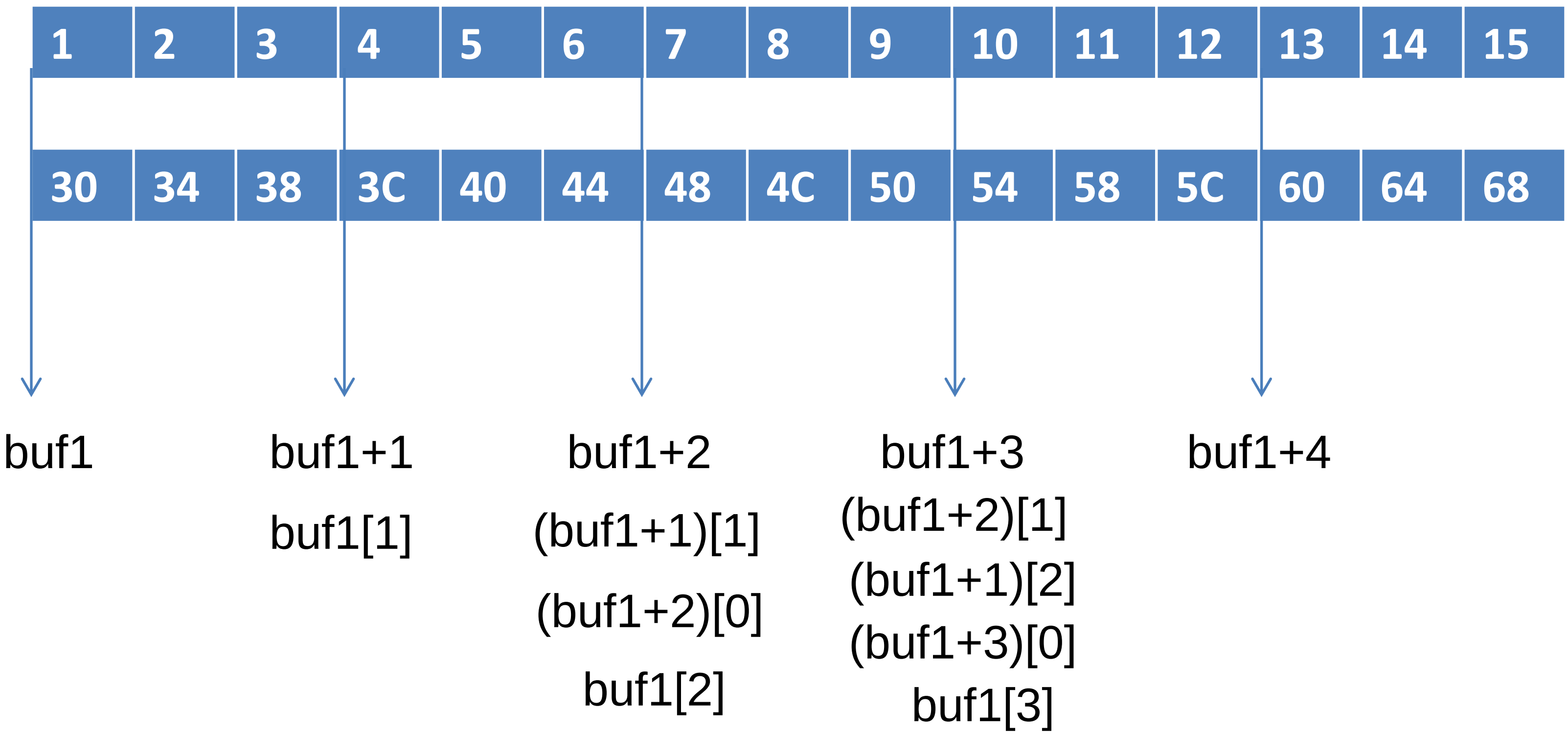
```
#include<stdio.h>
float avg(int a[], int size)
{
        int i;
        float total = 0.;
        for(i = 0; i < size; i++)
            total+=a[i];
        return total/size;
}

int main(){
        int array[4] = {2, 4, 6, 8};
         printf("Average : %f",avg(array,4));
         return 0;
}
```

```
#include<stdio.h>
float avg(int *a, int size)
{
        int i;
        float total = 0.;
        for(i = 0; i < size; i++)
            total+=*(a+i);
        return total/size;
}

int main(){
        int array[4] = {2, 4, 6, 8};
         printf("Average : %f",avg(array,4));
         return 0;
}
```

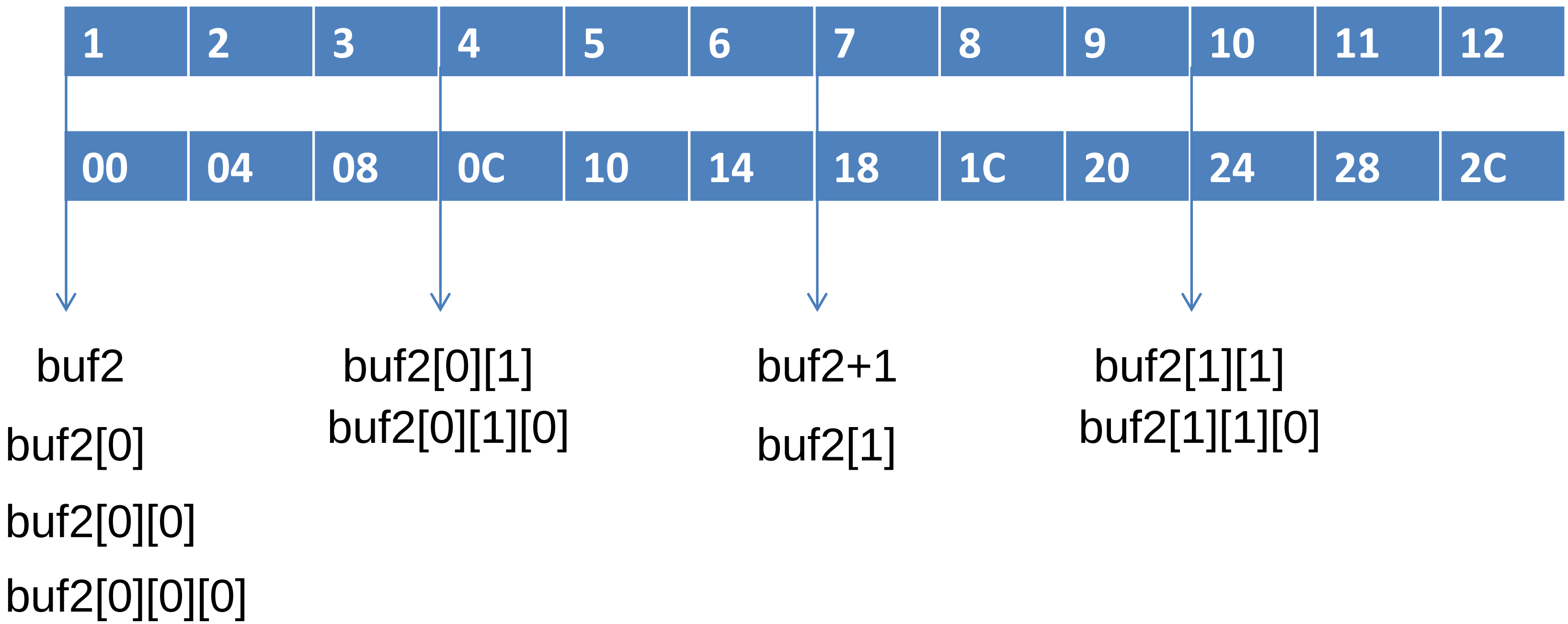
```c
#include<stdio.h>
int main(){

int buf[]={1,2,3,4,5,6,7,8};

int
buf1[][3]={{1,2,3},{4,5,6},{7,8,9},{10,11,12},{13,14,15}
};
int buf2[][2][3]={{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}};
printf("address of buf[2]= %p\n",&buf[2]);
printf("address of (buf+2) = %p\n",(buf+2));
printf("address of (buf+1)[1]= %p\n\n",&(buf+1)[1]);
```

```
printf("address of buf1[1][2]=%p\n",&buf1[1][2]);
printf("address of buf1[1]= %p\n",&buf1[1]);
printf("address of (buf1+1)= %p\n",(buf1+1));
printf("address of buf1[0]= %p\n",&buf1[0]);
printf("address of (buf1+1)[2]=%p\n",&(buf1+1)[2]);
printf("address of (buf1+3)=%p\n\n",(buf1+3));

printf("address of buf2 =%p\n",buf2);
printf("address of buf2[0] =%p\n",buf2[0]);
printf("address of buf2[0][0] =%p\n",buf2[0][0]);
printf("address of (buf2+1) =%p\n",(buf2+1));
printf("address of buf2[1] =%p\n",buf2[1]);
return 0;
}
```

Structure of buf1



Structure of buf2



address of buf[2]= 0x7fffc42fab78

address of (buf+2) = 0x7fffc42fab78

address of (buf+1)[1]= 0x7fffc42fab78

address of buf1[1][2]=0x7fffc42fab44

address of buf1[1]= 0x7fffc42fab3c

address of (buf1+1)= 0x7fffc42fab3c

address of buf1[0]= 0x7fffc42fab30

address of (buf1+1)[2]=0x7fffc42fab54

address of (buf1+3)=0x7fffc42fab54

address of buf2 =0x7fffc42fab00

address of buf2[0] =0x7fffc42fab00

address of buf2[0][0] =0x7fffc42fab00

address of (buf2+1) =0x7fffc42fab18

address of buf2[1] =0x7fffc42fab18

```
#include<stdio.h>
int main(){
int buf[]={1,2,3,4,5,6,7,8};
int *p, *q;
printf("address of buf[2]= %p\n",&buf[2]);
printf("address of (buf+2) = %p\n",(buf+2));
printf("address of (buf+1)[1]= %p\n\n",&(buf+1)[1]);
p=buf;
p=p+1;
printf("address of p= %p\n",p);
q=buf+1;
printf("address of q= %p\n",q);
buf=buf+1;
return 0;
}
```

Assume last two place of hexadecimal address of buf2[0] is 00.

What will be the output?

address of buf[2]= 0x7fff91baa408

address of (buf+2) = 0x7fff91baa408

address of (buf+1)[1]= 0x7fff91baa408

address of p= 0x7fff91baa404

address of q= 0x7fff91baa404

Write a program that takes as input *N* integers and print the numbers in reverse order?

Dynamic Memory Allocation

- So far, memory allocation was handled automatically at compile time.
- Certain cases you don't know how much memory to set aside
  - Dynamically allocate memory to variables at run-time. Example is an unsized array.
- The following four functions are used:

  malloc(), calloc(), realloc() and free()

malloc()

- Requires one argument, the number of bytes you want to allocate dynamically
- If successful, returns a void pointer
  - assign this to a pointer variable
  - void *p;

    p = malloc(10 * sizeof(int));
- If memory allocation fails, malloc will return a NULL pointer.

free()

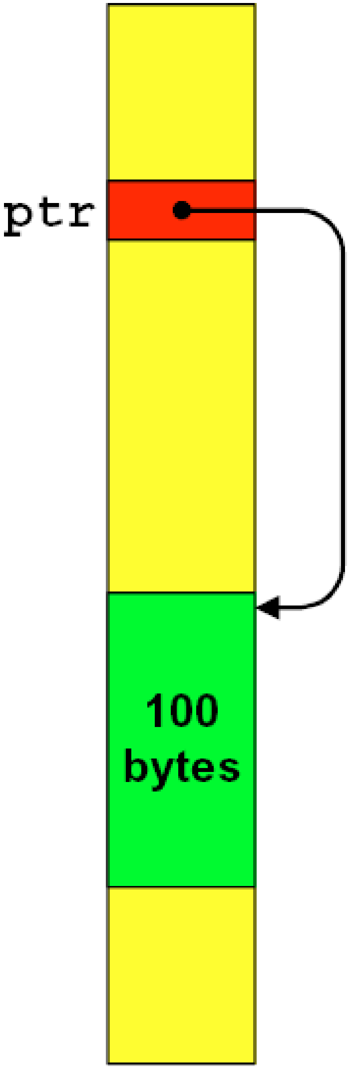
- free(ptr) will release the memory that was allocated to the pointer variable ptr.
- It is good practice to free memory when you are done with it.

```
#include <stdio.h>
#include <stdlib.h> /* required for the malloc and free
                        functions */

int main() {
        int number, i;
        int *ptr;

        printf("How many ints would you like store? ");
        scanf("%d", &number);
         /* allocate memory */
        ptr = (int *)malloc(number*sizeof(int));
        if(ptr!=NULL) {
                for(i=0 ; i<number ; i++)
                        scanf("%d",(ptr+i));
```

```
/* print out in reverse order */
for(i=number - 1 ; i>=0 ; i--) {
      printf("%d\n", *(ptr+i));
    }
/* free allocated memory */
    free(ptr);
    return 0;
  }
  else {
    printf("\nMemory allocation failed.\n");
    return 1;
  }
}
```

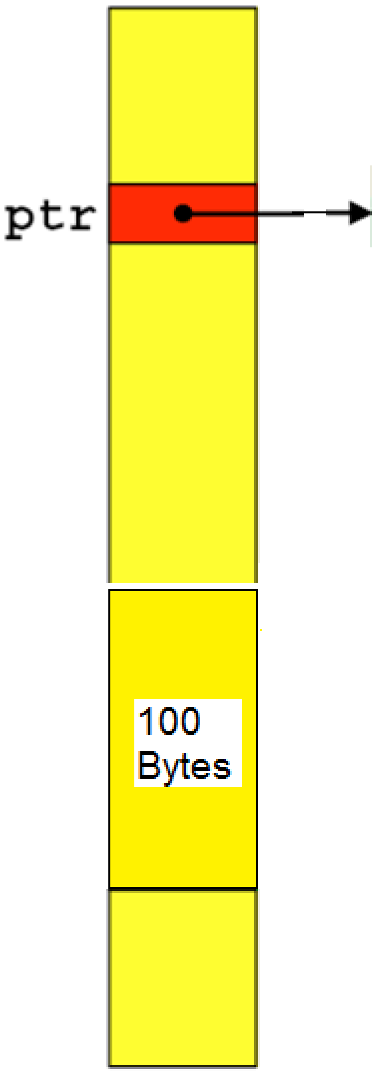
Review
Allocating memory @run-time

```
char *ptr;
ptr = (char *)malloc(100);
if (ptr == NULL)
printf("…...!!");
```



- *malloc* allocates contiguous block of memory of at least 100 bytes and returns the address of the first byte
- Returns NULL if the allocation fails.

Review
De-allocating memory




```
free(ptr);
```

- De-allocates the block of memory pointed to by *ptr*.
- After calling free, *ptr* is uninitialized and using ptr will result in error
- The lifetime of an allocated block is determined by malloc/calloc/realloc and free; other functions have no effect on its existence.

calloc()

- Allocates continuous space for an array of elements.
- Requires two arguments,
  - calloc(n, el_size), allocate space for n elements each of *el_size*
  - space shall be all initialized to 0 bits
- If successful, returns a void pointer else NULL

```
#include <stdio.h>
#include <stdlib.h> /* required for the malloc and free
                              functions */

int main() {
        int number, i;
        int *ptr;

        printf("How many ints would you like store? ");
        scanf("%d", &number);
         /* allocate memory */
        ptr = (int *) malloc(number*sizeof(int));
        if(ptr!=NULL) {
                for(i=0 ; i<number ; i++)
                        scanf("%d",(ptr+i));
```

```
#include <stdio.h>
#include <stdlib.h> /* required for the malloc and free
                                  functions */

int main() {
        int number, i;
        int *ptr;

        printf("How many ints would you like store? ");
        scanf("%d", &number);
         /* allocate memory */
        ptr = (int *)calloc(number, sizeof(int));
        if(ptr!=NULL) {
                for(i=0 ; i<number ; i++)
                        scanf("%d",(ptr+i));
```

malloc Vs. calloc

malloc	calloc
• Allocated space is not initialized • Takes single argument	▪ Allocated space initialized to zero ▪ Takes two arguments

realloc()

- Allows to allocate more memory space without losing data.
- Requires two arguments,
  - realloc(*ptr, Total_byte),
  - First is the pointer referencing memory
  - Second is the total no. of bytes you want to reallocate.
- If successful, returns a void pointer else NULL

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

int main() {
  int *ptr;
  int i;

  ptr = calloc(5, sizeof(int));

  if(ptr!=NULL) {
        *ptr = 1;
        *(ptr+1) = 2;
        ptr[2] = 4;
        ptr[3] = 8;
        ptr[4] = 16;
```

```
ptr = realloc(ptr, 7*sizeof(int));

    if(ptr!=NULL) {
      printf("Now allocating more memory... \n");
      ptr[5] = 32; /* now it's legal! */
      ptr[6] = 64;

      for(i=0 ; i<7 ; i++) {
        printf("ptr[%d] holds %d\n", i, ptr[i]);
      }
      realloc(ptr,0); /* same as free(ptr); - just fancier!
*/
      return 0;
    }
}
```

Can we allocate space for single variable?

- malloc can be used to allocate memory for single variable also
  - ptr = (int *) malloc (sizeof(int));
  - Allocates space for a single int, which can be accessed as *ptr
- Single variable allocation is just a special case of array allocations
  - Array with only one element