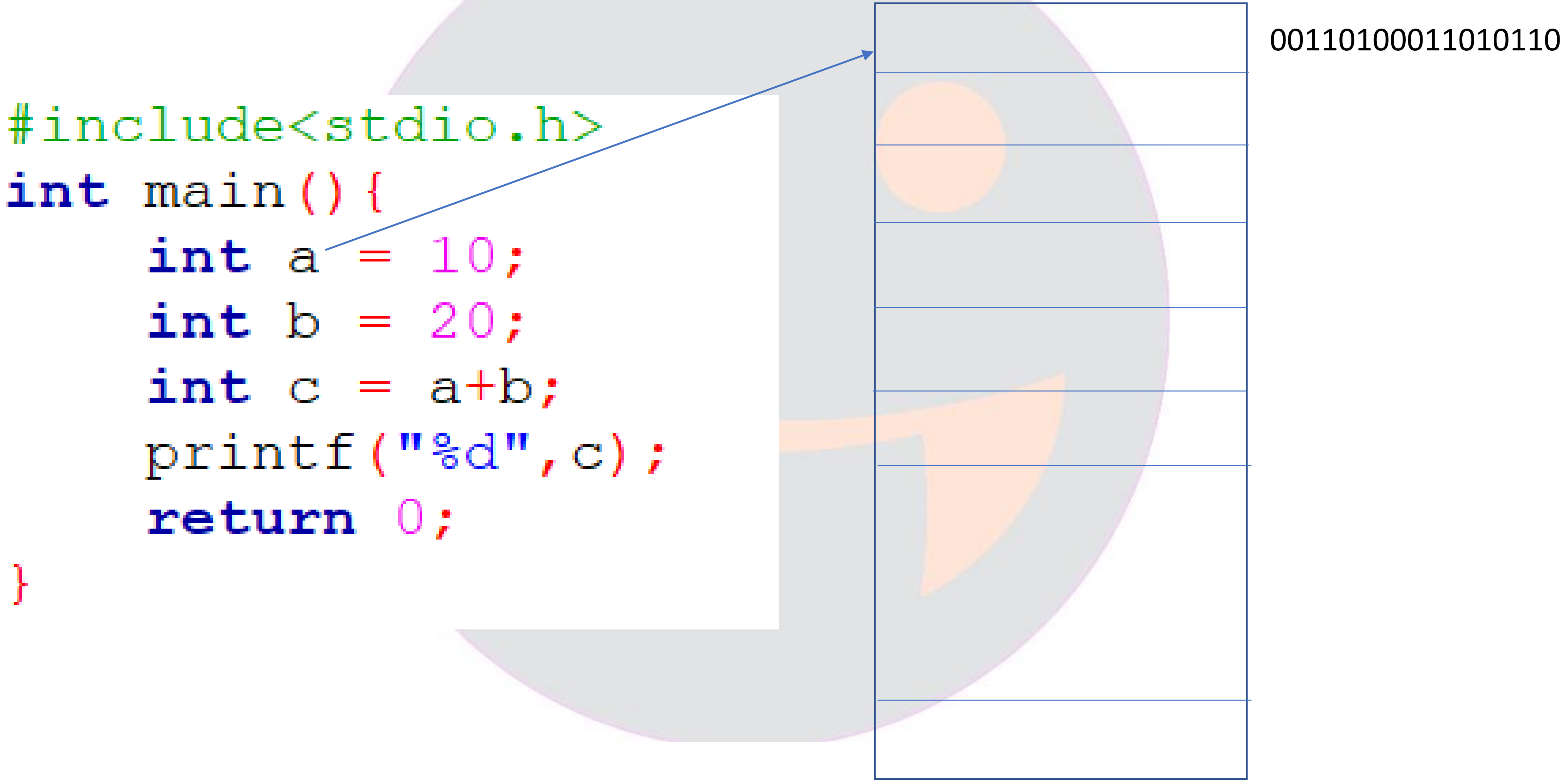
Pointers

CHAPTER 35

SUREH TECHS

C PROGRAMMING COURSE

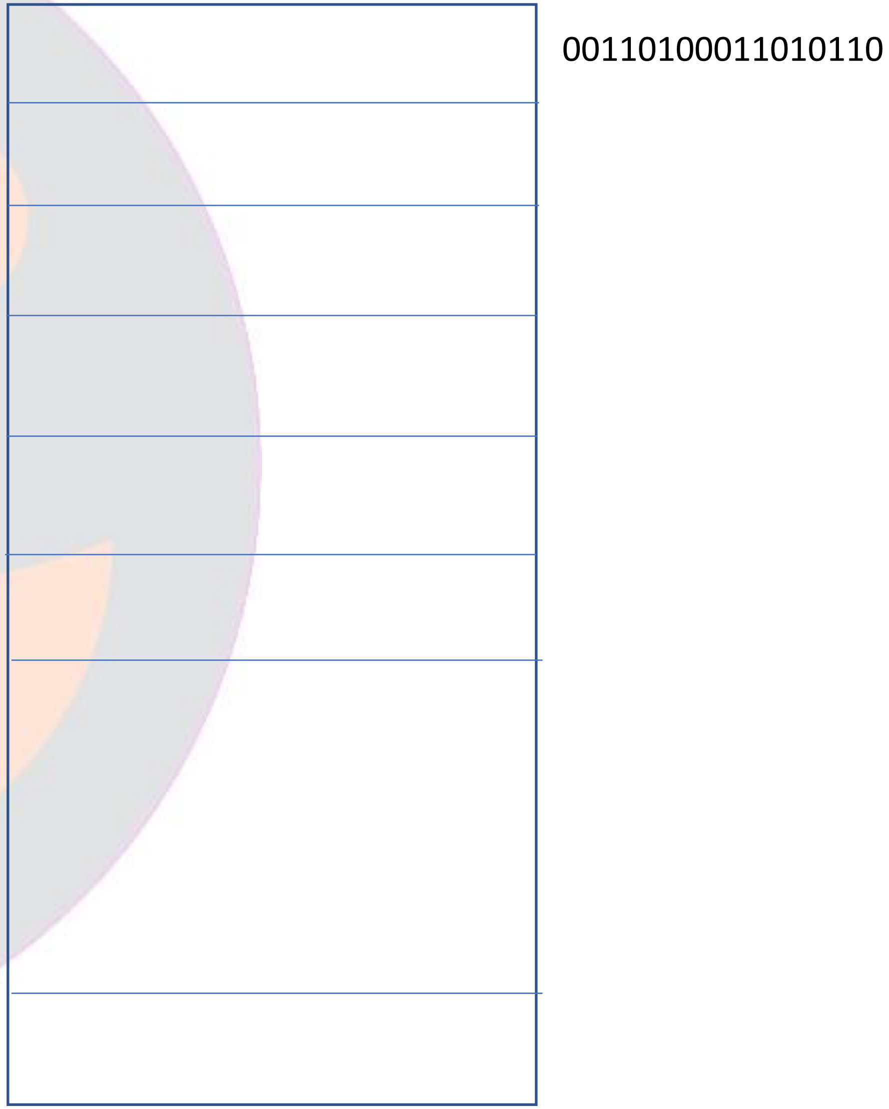
a,b, and c holds values here



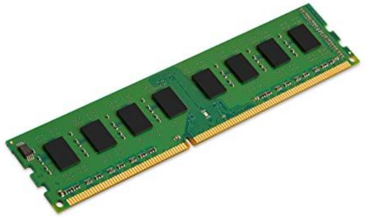
```
#include<stdio.h>
int main(){
     int a = 10;
     int b = 20;
     int c = a+b;
     printf("%d",c);
     return 0;
}
```

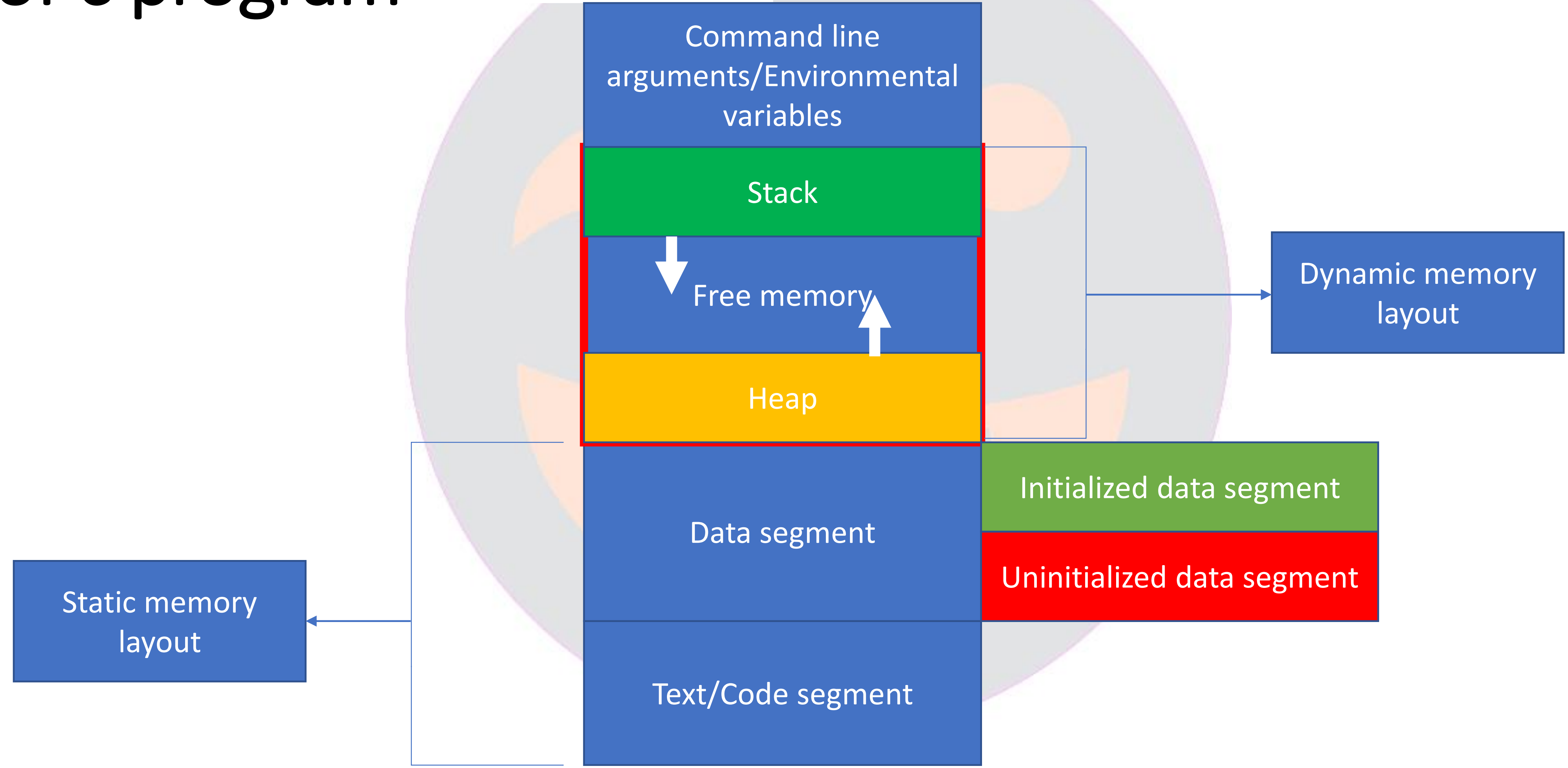
Pointer

- Pointer is like a variable but holds **address of that variable not the value**
- We use them when we want to allocate some memory on the heap which is also known as **dynamic memory allocation**

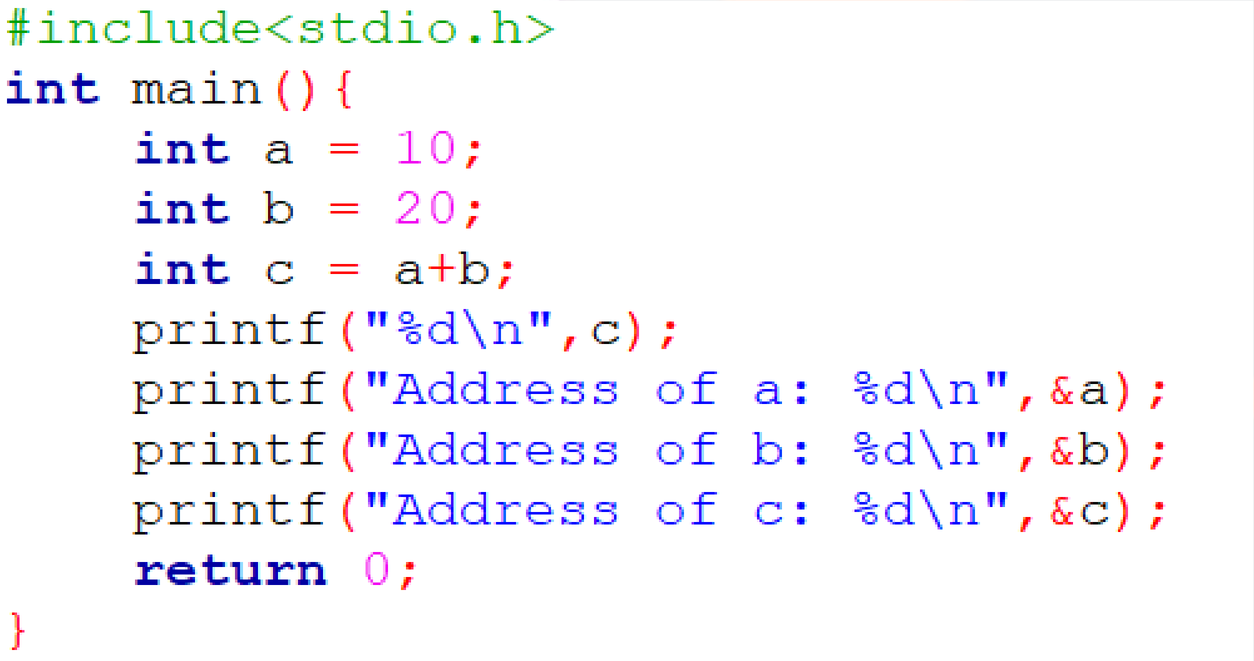


Memory layout of c program





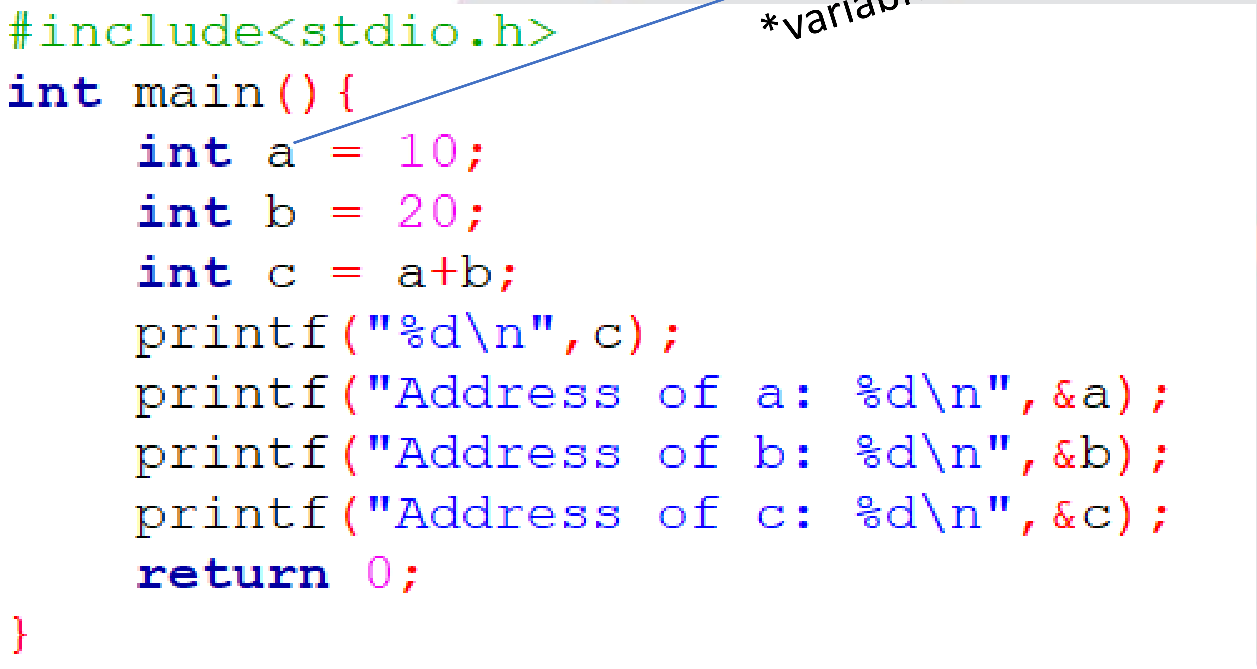
We already know how to see the address of the variable



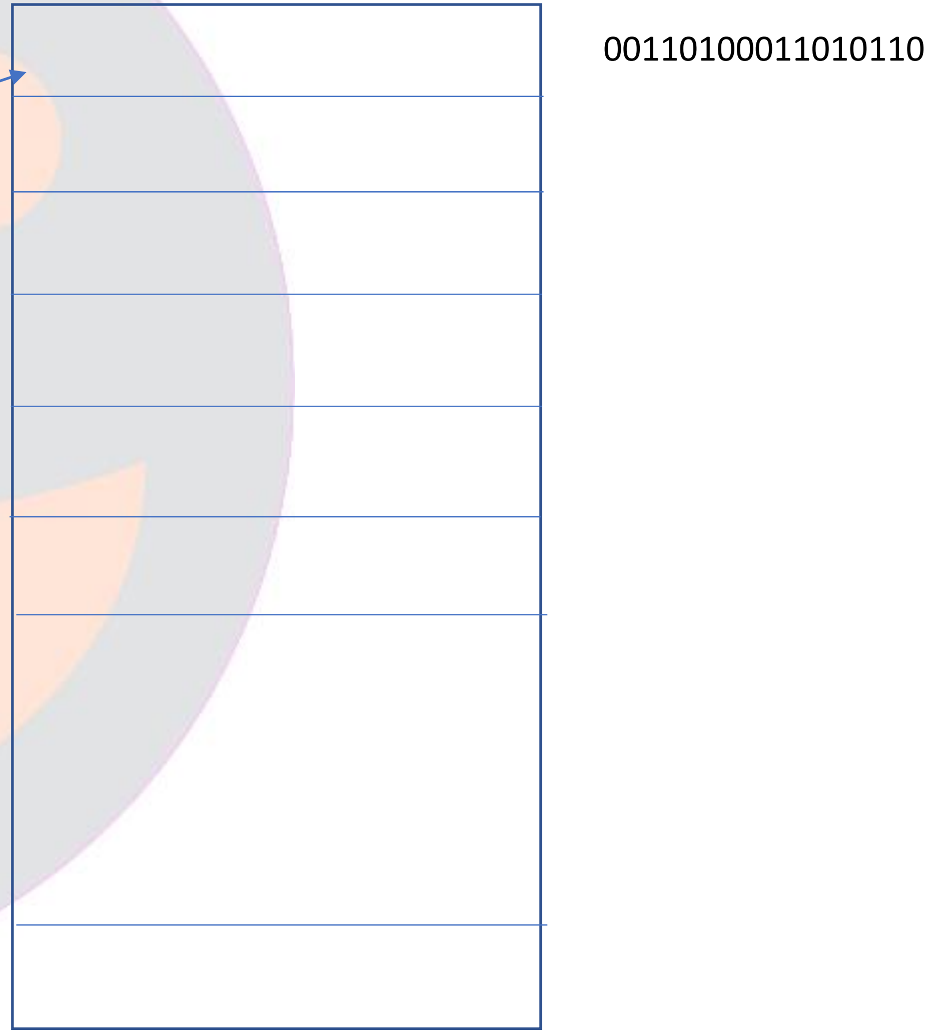
```
#include<stdio.h>
int main(){
    int a = 10;
    int b = 20;
    int c = a+b;
    printf("%d\n",c);
    printf("Address of a: %d\n",&a);
    printf("Address of b: %d\n",&b);
    printf("Address of c: %d\n",&c);
    return 0;
}
```

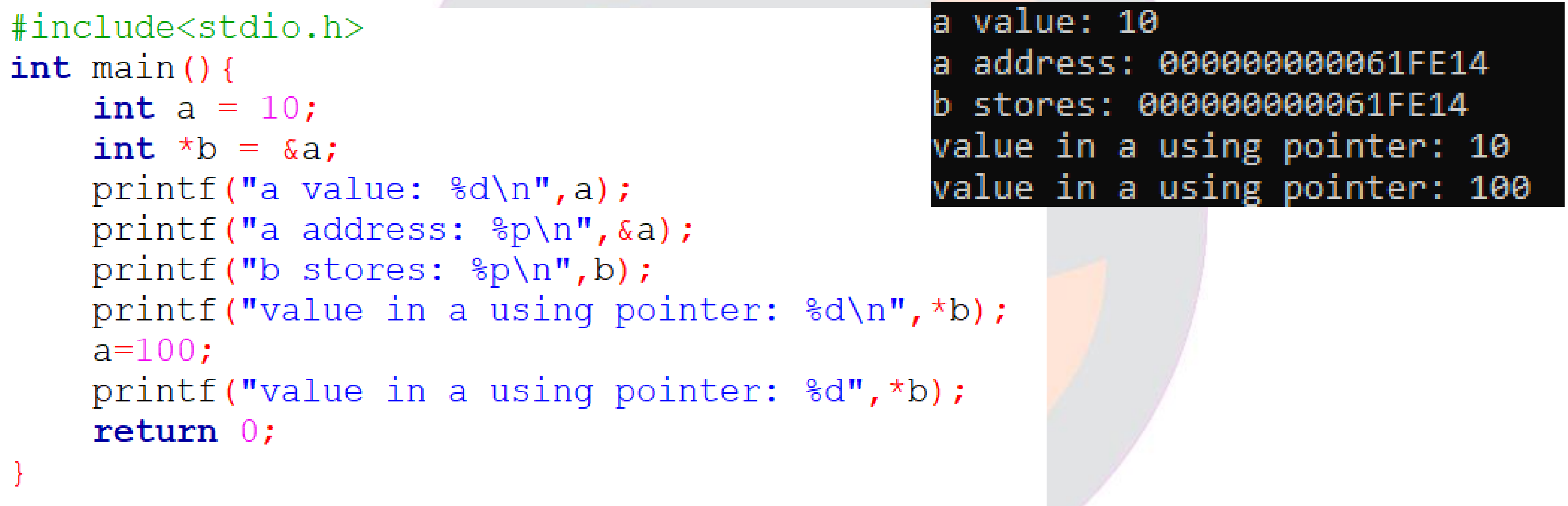
Syntax

- To declare a pointer variable we use “*” before the variable name.



```
#include<stdio.h>
int main(){
    int a = 10;
    int b = 20;
    int c = a+b;
    printf("%d\n",c);
    printf("Address of a: %d\n",&a);
    printf("Address of b: %d\n",&b);
    printf("Address of c: %d\n",&c);
    return 0;
}
```

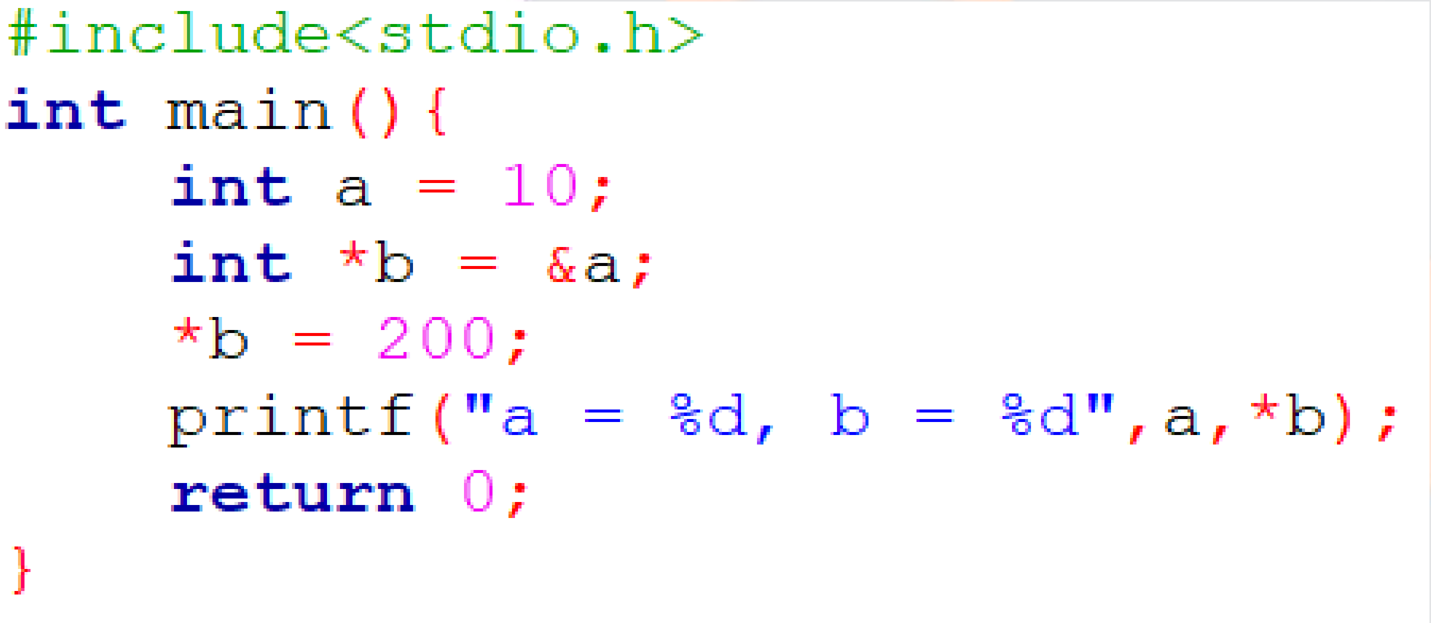




```c
#include<stdio.h>
int main(){
    int a = 10;
    int *b = &a;
    printf("a value: %d\n",a);
    printf("a address: %p\n",&a);
    printf("b stores: %p\n",b);
    printf("value in a using pointer: %d\n",*b);
    a=100;
    printf("value in a using pointer: %d",*b);
    return 0;
}
```

```
a value: 10
a address: 000000000061FE14
b stores: 000000000061FE14
value in a using pointer: 10
value in a using pointer: 100
```

Changing the value of b is also changing value of a because both are pointing to the same address



```c
#include<stdio.h>
int main(){
    int a = 10;
    int *b = &a;
    *b = 200;
    printf("a = %d, b = %d",a,*b);
    return 0;
}
```

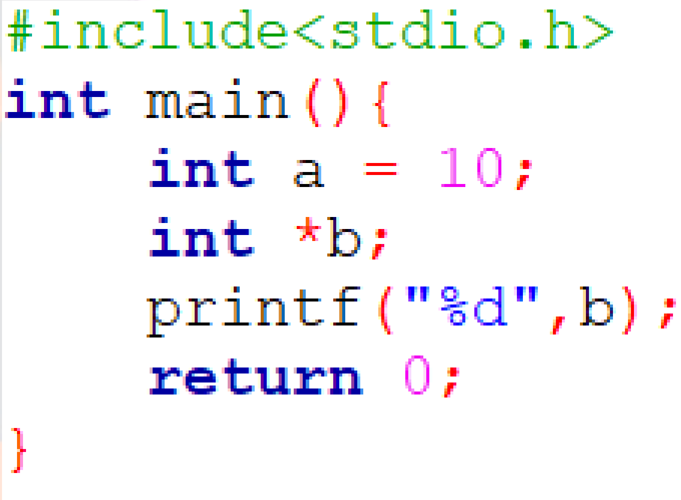
Types of pointers

- **Wild pointer**
- **Null pointer**
- **Void pointer**
- **Dangling pointer**

- Complex pointer
- Near pointer
- Far pointer
- Huge pointer

Wild pointer

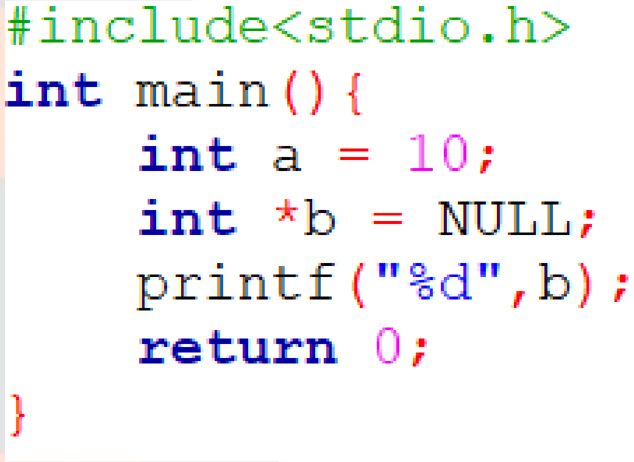
- Wild pointers are basically **uninitialized pointers that points to arbitrary memory location** and may cause a program to crash or behave badly.



```
#include<stdio.h>
int main(){
    int a = 10;
    int *b;
    printf("%d",b);
    return 0;
}
```

Null pointer

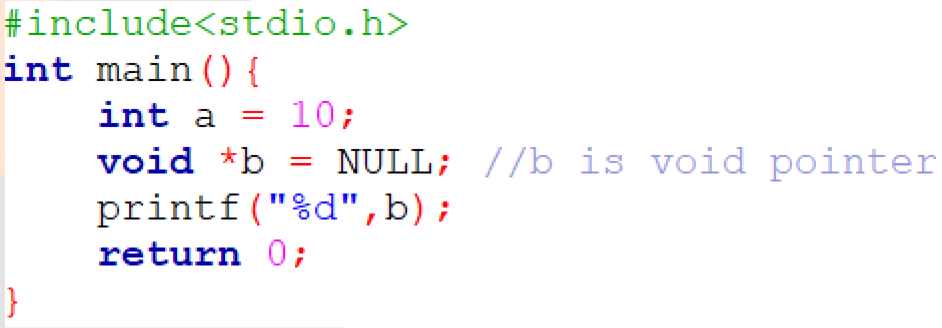
- If we don't initialize a pointer after declaration then **by default the pointer is a Null pointer**.
- We can also explicitly do this by assigning the NULL value at the time of pointer declaration.
- This method is useful when you do not assign any address to the pointer.



```
#include<stdio.h>
int main(){
    int a = 10;
    int *b = NULL;
    printf("%d",b);
    return 0;
}
```

Void pointer

- The void pointer is a **generic pointer** that **does not have any data type** associated with it.
- The **datatype of void pointer** can be of **any type and can be typecast to any type**.



```
#include<stdio.h>
int main(){
    int a = 10;
    void *b = NULL; //b is void pointer
    printf("%d",b);
    return 0;
}
```

Dangling pointer

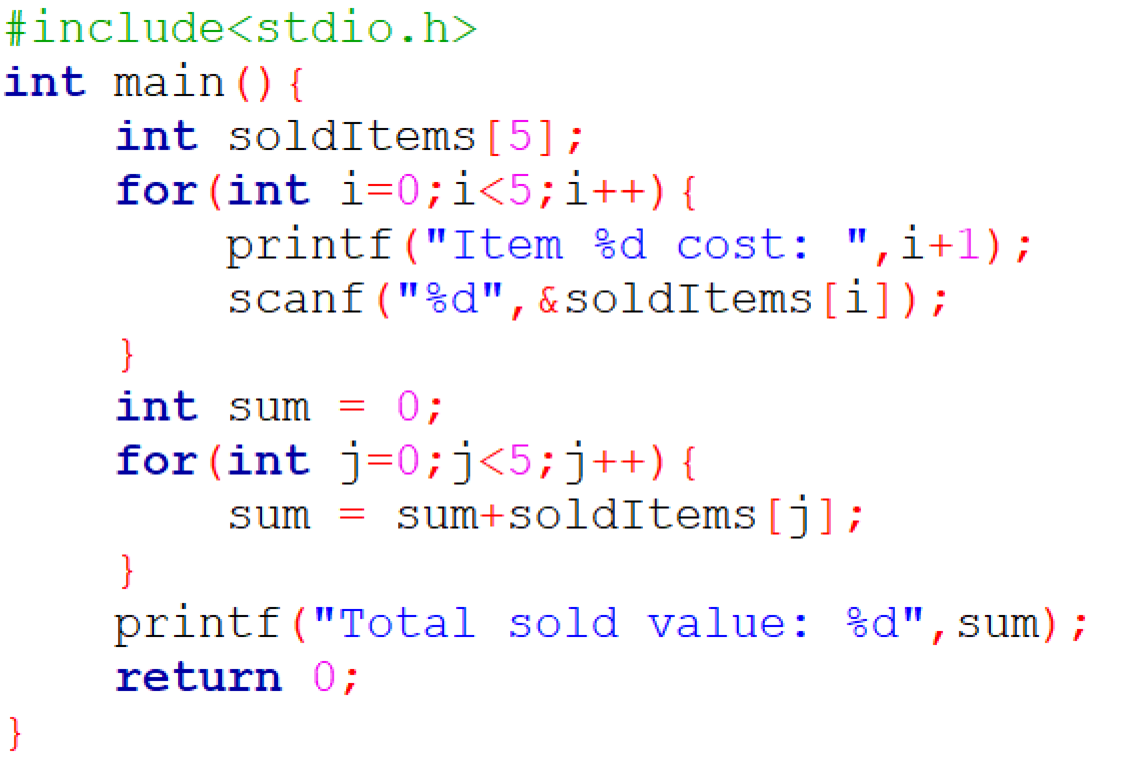
- A dangling pointer is a **pointer that is pointing to a memory location that has been deleted or released.**
- **After free(), pointer variable becomes dangling pointer(Discuss while using free)**
- **We should assign NULL as soon deallocation happens using free**

Benefits of pointers

- Major benefit: Dynamic memory allocation
- Disadvantages of static memory allocation

1. Many items will be sold in a day
2. Each item will have a cost
3. How to get the total money received in a day?
4. Sum of the item costs in that day

Ex: int soldItems[50];



```
#include<stdio.h>
int main(){
    int soldItems[5];
    for(int i=0;i<5;i++){
        printf("Item %d cost: ",i+1);
        scanf("%d",&soldItems[i]);
    }
    int sum = 0;
    for(int j=0;j<5;j++){
        sum = sum+soldItems[j];
    }
    printf("Total sold value: %d",sum);
    return 0;
}
```

But the soldItems value can change every day, so we need to make it dynamic

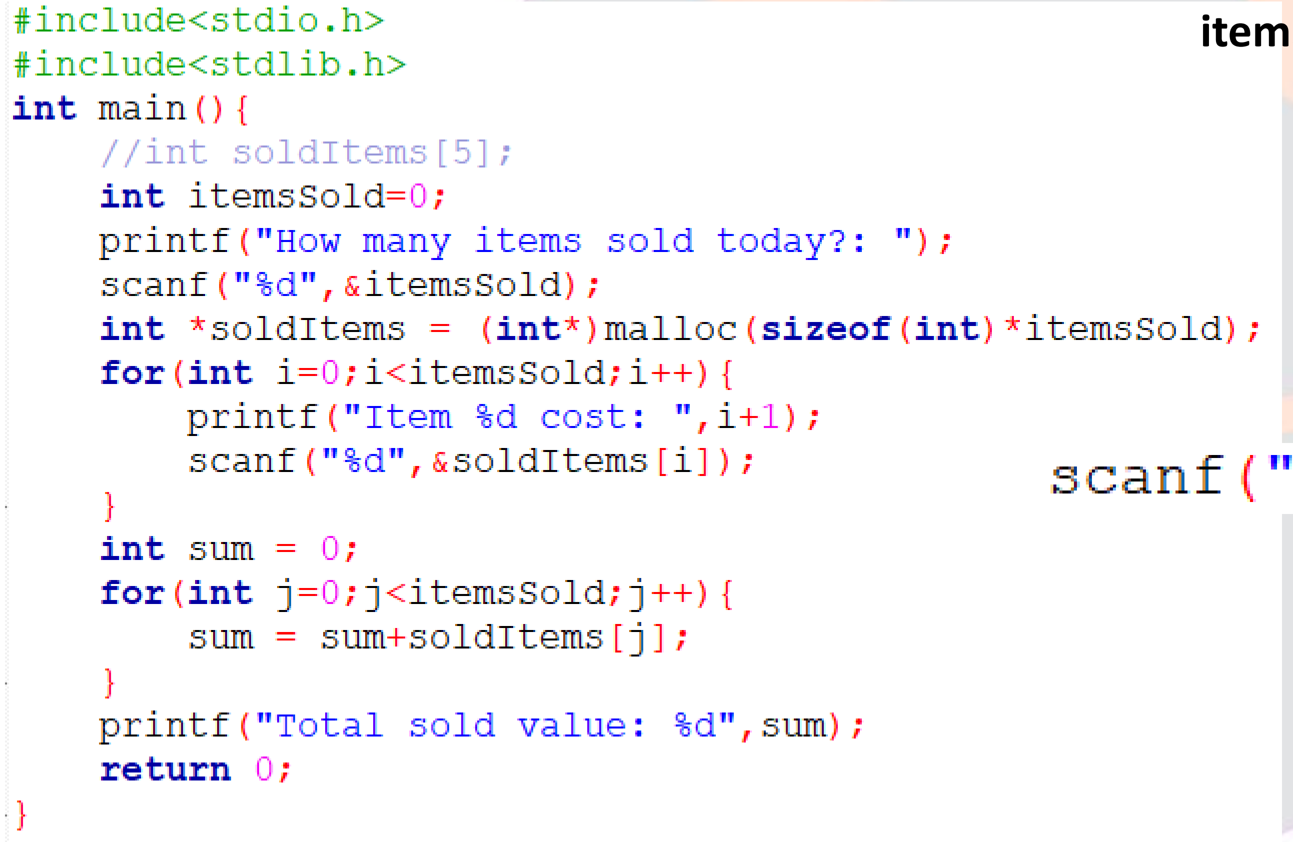
<stdlib.h>

- **malloc()**
- **calloc()**
- **free()**
- **realloc()**

malloc()

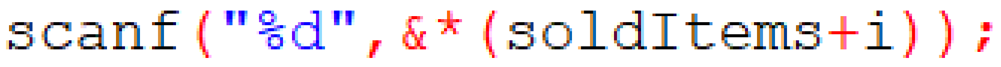
- Also known as **memory allocation**
- Allocates **single large block of memory** with the **specified size**
- **malloc(size in bytes)**
- malloc() returns a pointer of type **void** which can be cast into pointer of any other type
- **(cast-type*)malloc(size in bytes)**
- **ptr = (cast-type*)malloc(size in bytes)**
- **If the allocation fails it returns NULL**

malloc()



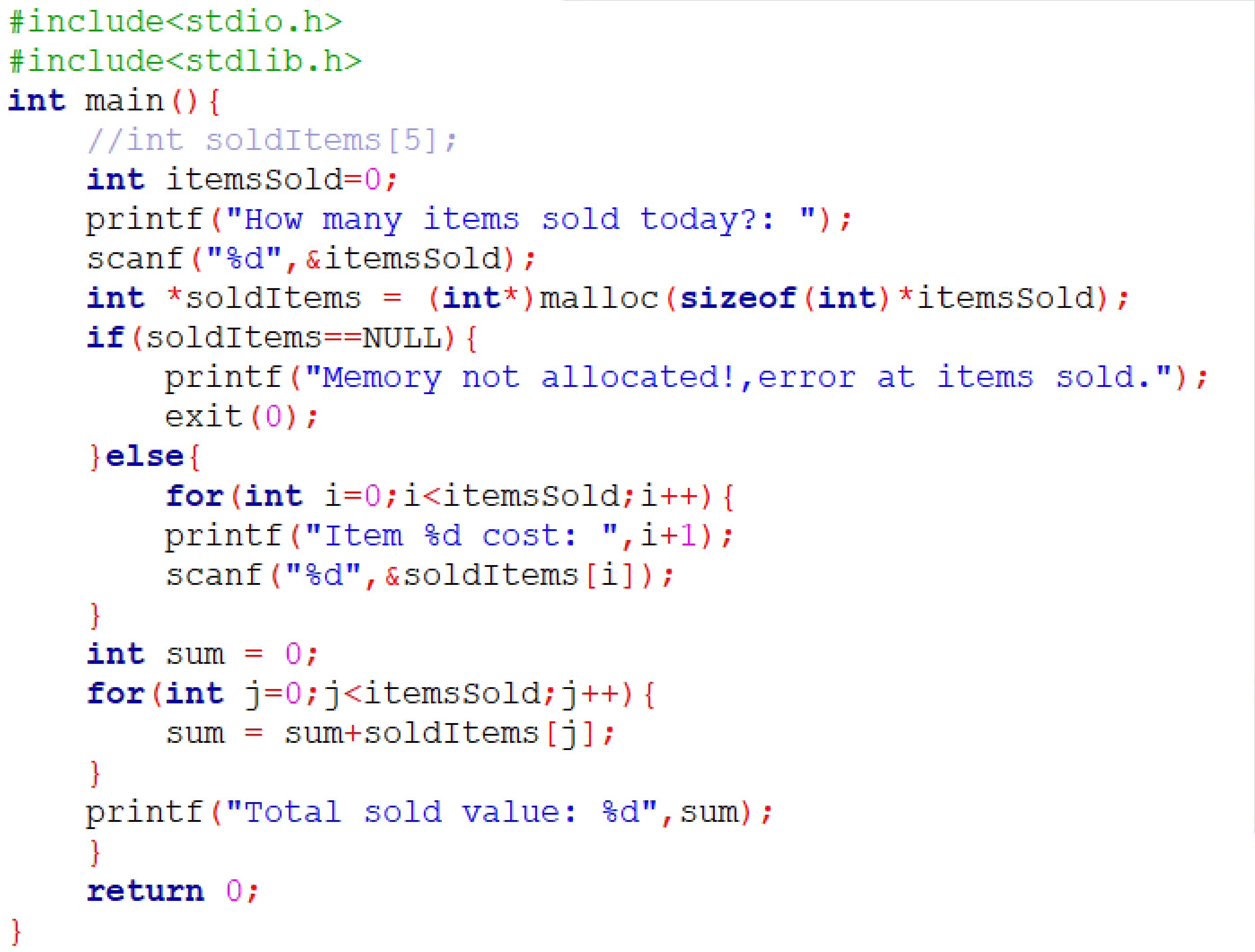
```c
#include<stdio.h>
#include<stdlib.h>
int main(){
    //int soldItems[5];
    int itemsSold=0;
    printf("How many items sold today?: ");
    scanf("%d",&itemsSold);
    int *soldItems = (int*)malloc(sizeof(int)*itemsSold);
    for(int i=0;i<itemsSold;i++){
        printf("Item %d cost: ",i+1);
        scanf("%d",&soldItems[i]);
    }
    int sum = 0;
    for(int j=0;j<itemsSold;j++){
        sum = sum+soldItems[j];
    }
    printf("Total sold value: %d",sum);
    return 0;
}
```

What if you enter 100000000000000 in the itemsSold?



```c
scanf("%d",&*(soldItems+i));
```

Handle if memory is not allocated



```c
#include<stdio.h>
#include<stdlib.h>
int main(){
    //int soldItems[5];
    int itemsSold=0;
    printf("How many items sold today?: ");
    scanf("%d",&itemsSold);
    int *soldItems = (int*)malloc(sizeof(int)*itemsSold);
    if(soldItems==NULL){
        printf("Memory not allocated!,error at items sold.");
        exit(0);
    }else{
        for(int i=0;i<itemsSold;i++){
        printf("Item %d cost: ",i+1);
        scanf("%d",&soldItems[i]);
    }
    int sum = 0;
    for(int j=0;j<itemsSold;j++){
        sum = sum+soldItems[j];
    }
    printf("Total sold value: %d",sum);
    }
    return 0;
}
```

But what if you keeps on assigning memories dynamically?

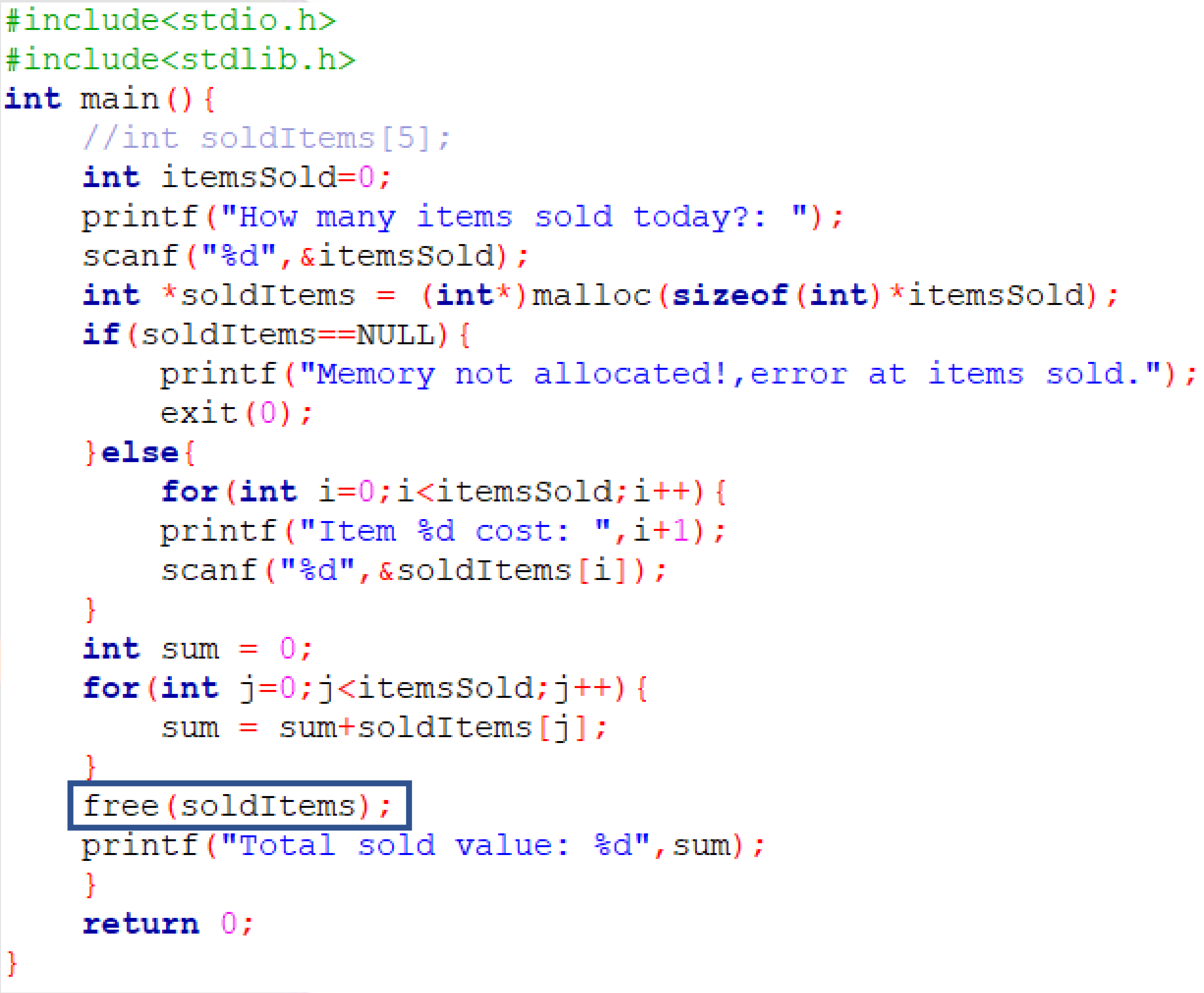
- You may end up wasting memory
- So we need to remove the memory whenever we feel like it is not required
- free()

free()

- Used to **de-allocate** the memory
- **Note:** Memory allocated using malloc() and calloc() is not de-allocated on their own
- free(ptr)

- No indication of success or failure is returned.

- **void** free(void *pointer);



```c
#include<stdio.h>
#include<stdlib.h>
int main(){
    //int soldItems[5];
    int itemsSold=0;
    printf("How many items sold today?: ");
    scanf("%d",&itemsSold);
    int *soldItems = (int*)malloc(sizeof(int)*itemsSold);
    if(soldItems==NULL){
        printf("Memory not allocated!,error at items sold.");
        exit(0);
    }else{
        for(int i=0;i<itemsSold;i++){
        printf("Item %d cost: ",i+1);
        scanf("%d",&soldItems[i]);
    }
    int sum = 0;
    for(int j=0;j<itemsSold;j++){
        sum = sum+soldItems[j];
    }
    free(soldItems);
    printf("Total sold value: %d",sum);
    }
    return 0;
}
```

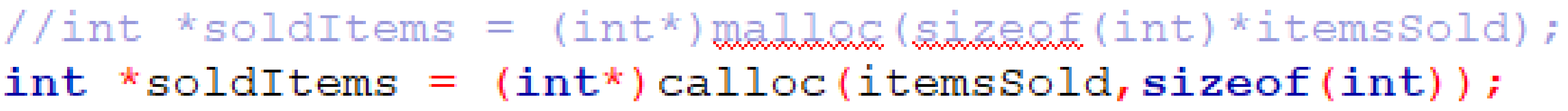
Size of the pointer is always same

- The **size of a pointer** and the **size of what it points to are not related**.
- Ex: Consider them like postal addresses.
- The size of the address of a house has no relationship to the size of the house.

Memory leak

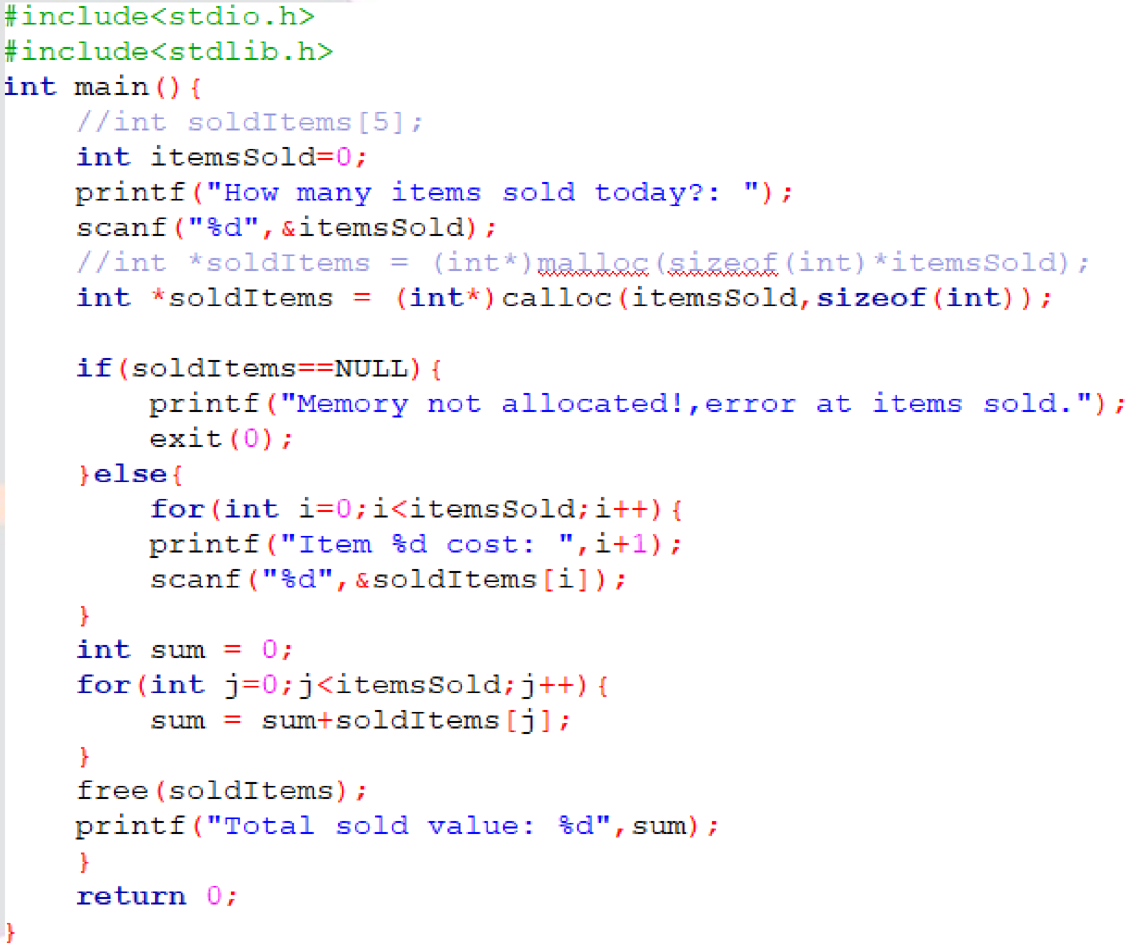
- When we assign a variable it takes space of our RAM (either heap or RAM)dependent on the size of data type, however, if a programmer uses a memory available on the heap and forgets to a delete it, at some point all the memory available on the ram will be occupied with no memory left this can lead to a memory leak

calloc()

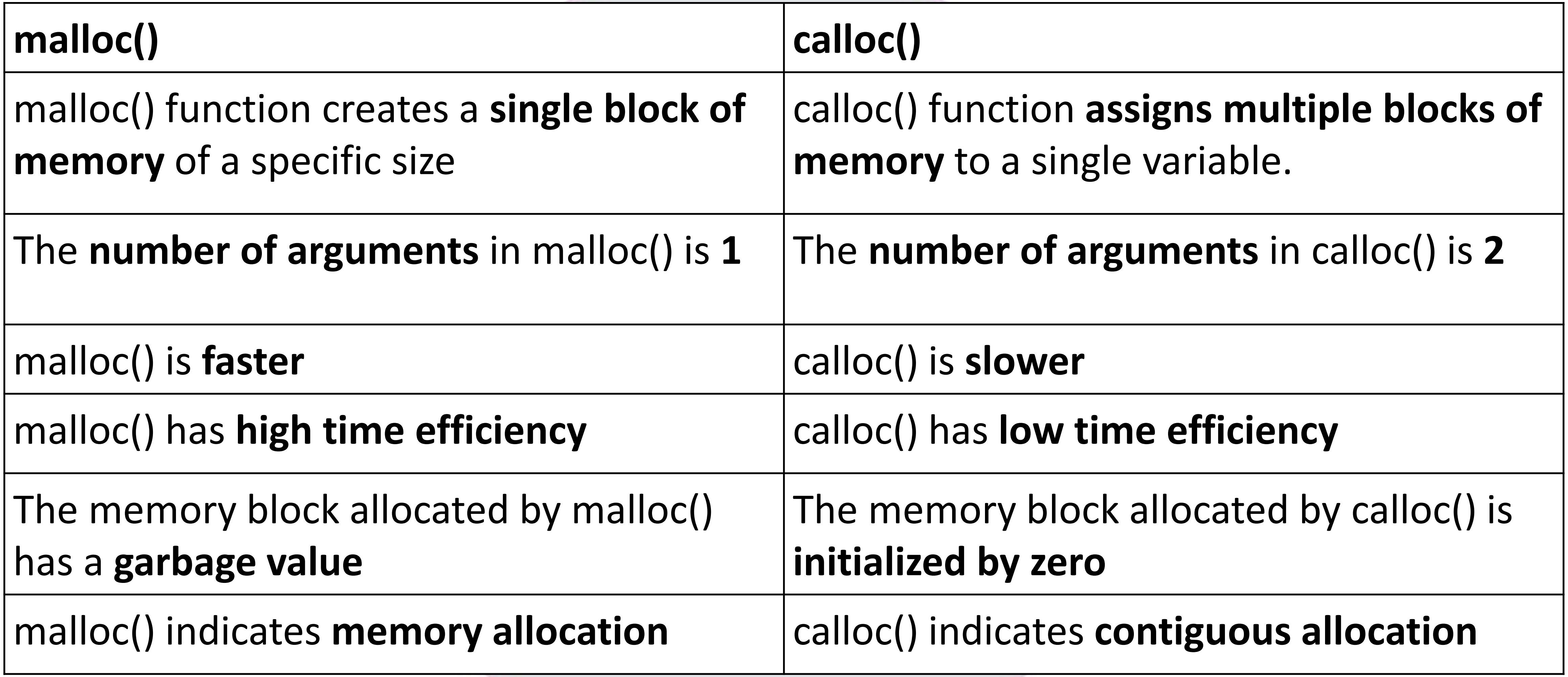


```
//int *soldItems = (int*)malloc(sizeof(int)*itemsSold);
int *soldItems = (int*)calloc(itemsSold,sizeof(int));
```

- Also known as **contiguous allocation**
- Used to **allocate specified number of blocks** of memory of the specified type
- malloc() allocates single block of memory where as calloc() allocates specified number of blocks
- Each block will have a default value of zero
- **ptr = (cast-type*)calloc(n,element-size)**
- If space is insufficient, allocation fails and returns a NULL pointer



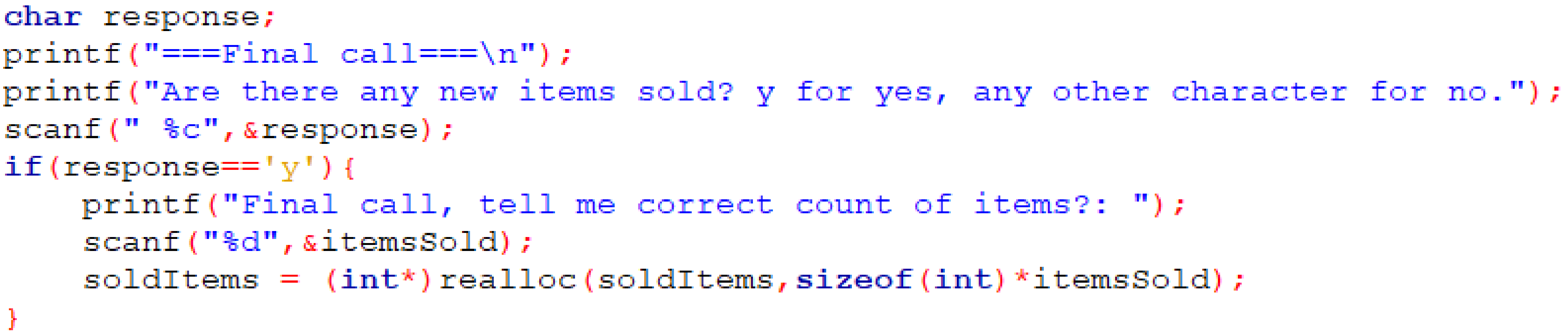
```
#include<stdio.h>
#include<stdlib.h>
int main(){
    //int soldItems[5];
    int itemsSold=0;
    printf("How many items sold today?: ");
    scanf("%d",&itemsSold);
    //int *soldItems = (int*)malloc(sizeof(int)*itemsSold);
    int *soldItems = (int*)calloc(itemsSold,sizeof(int));

    if(soldItems==NULL){
        printf("Memory not allocated!,error at items sold.");
        exit(0);
    }else{
        for(int i=0;i<itemsSold;i++){
        printf("Item %d cost: ",i+1);
        scanf("%d",&soldItems[i]);
    }
    int sum = 0;
    for(int j=0;j<itemsSold;j++){
        sum = sum+soldItems[j];
    }
    free(soldItems);
    printf("Total sold value: %d",sum);
    }
    return 0;
}
```



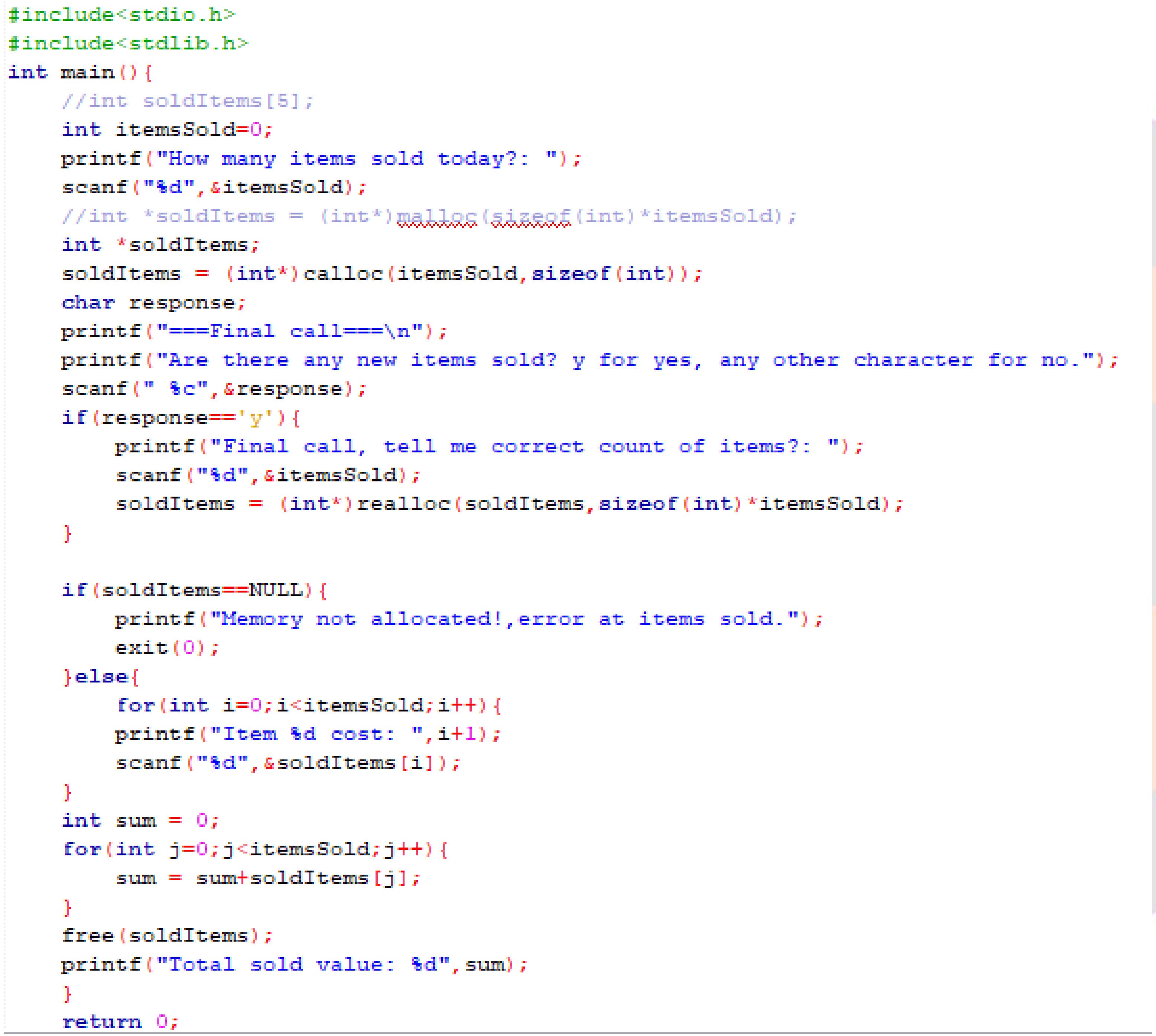
malloc()	**calloc()**
malloc() function creates a **single block of memory** of a specific size	calloc() function **assigns multiple blocks of memory** to a single variable.
The **number of arguments** in malloc() is **1**	The **number of arguments** in calloc() is **2**
malloc() is **faster**	calloc() is **slower**
malloc() has **high time efficiency**	calloc() has **low time efficiency**
The memory block allocated by malloc() has a **garbage value**	The memory block allocated by calloc() is **initialized by zero**
malloc() indicates **memory allocation**	calloc() indicates **contiguous allocation**

realloc()

- What if the memory allocated by malloc() or calloc() is **not sufficient?**
- We can **reallocate** it
- It is also know as **re-allocation**
- reallocation maintains the already present value and new blocks will be initialized with the default garbage value
- **ptr = (cast-type*)realloc(ptr,new-size)**
- If space is insufficient, allocation fails and returns a NULL pointer



```c
char response;
printf("===Final call===\n");
printf("Are there any new items sold? y for yes, any other character for no.");
scanf(" %c",&response);
if(response=='y'){
     printf("Final call, tell me correct count of items?: ");
     scanf("%d",&itemsSold);
     soldItems = (int*)realloc(soldItems,sizeof(int)*itemsSold);
}
```



```
#include<stdio.h>
#include<stdlib.h>
int main(){
    //int soldItems[5];
    int itemsSold=0;
    printf("How many items sold today?: ");
    scanf("%d",&itemsSold);
    //int *soldItems = (int*)malloc(sizeof(int)*itemsSold);
    int *soldItems;
    soldItems = (int*)calloc(itemsSold,sizeof(int));
    char response;
    printf("===Final call===\n");
    printf("Are there any new items sold? y for yes, any other character for no.");
    scanf(" %c",&response);
    if(response=='y'){
        printf("Final call, tell me correct count of items?: ");
        scanf("%d",&itemsSold);
        soldItems = (int*)realloc(soldItems,sizeof(int)*itemsSold);
    }

    if(soldItems==NULL){
        printf("Memory not allocated!,error at items sold.");
        exit(0);
    }else{
        for(int i=0;i<itemsSold;i++){
        printf("Item %d cost: ",i+1);
        scanf("%d",&soldItems[i]);
    }
    int sum = 0;
    for(int j=0;j<itemsSold;j++){
        sum = sum+soldItems[j];
    }
    free(soldItems);
    printf("Total sold value: %d",sum);
    }
    return 0;
```

Dennis Ritchie

1. 10 bores are working very fine

2. Pass percentage of school students are 99.26

3. “SURBALI” became friend of Alistair

4. “SURESH TECHS”

Let’s operate our kingdom

Father

SURBALI

C Programming In Telugu

TELUGU