Basis and Practice in Programming

Chapter 12: Dynamic memory allocation

Prof. Tamer ABUHMED
College of Software

성균관대학교
SUNG KYUN KWAN UNIVERSITY(SKKU)

Class Objectives

- **Explain Dynamic Memory Allocation**
- **Explain Dynamic Memory Allocation of Arrays, Strings**
- **Explain Lifetime Of Dynamic Allocated Memory**
- **Explain Pointers to Structures**
- **Explain Pointers to Pointers**

Dynamic memory allocation

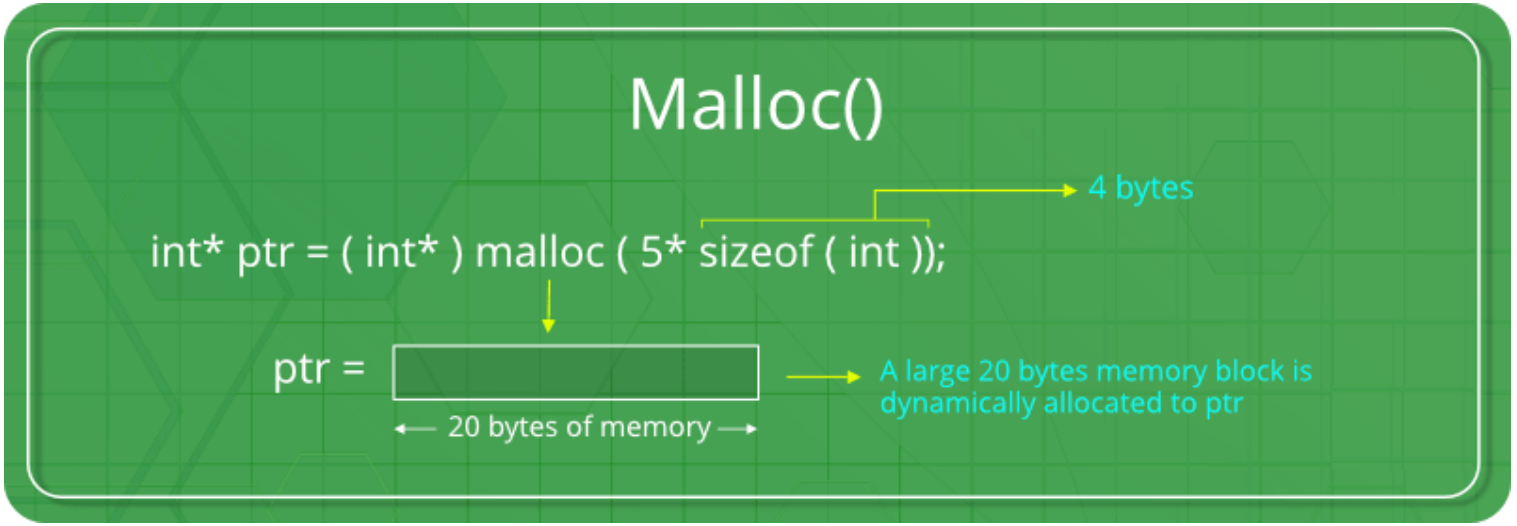
- Variable definitions in C: the C compiler automatically allocates the correct amount of storage (1-n memory locations) where the variable will be stored -> this happens before the execution of the actual program starts
- Sometimes it can be useful to *dynamically* allocate storage while a program is running:
- Suppose we have a program that is designed to read in a set of data from input into an array in memory but we don't know how many data items there are until the program starts execution. We have three choices:
  - Define the array to contain the maximum number of possible elements at compile time.
  - Use a variable-length array to dimension the size of the array at runtime.
  - Allocate the array dynamically using one of C's memory allocation routines.

malloc()

- <stdlib.h>

```
void * malloc(int n);
```

- malloc allocates *n* bytes of memory and returns a pointer to them if the allocation was succesful, NULL otherwise
- Before using the pointer returned by malloc, it has to be checked if it is not NULL !!
- The pointer returned by malloc is of the generic type void *; it has to be converted to a concrete pointer type



malloc() Example

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char* line;
    int linelen;
    printf("How long is your line?\n");
    scanf("%d\n", &linelen);
    line = malloc(linelen*sizeof(char));
    /* incomplete here-malloc's return value not checked*/
    gets(line);
    puts(line);
    // another example
    char* somestring = "how are you";
    char* copy;
    copy = malloc(strlen(somestring) + 1);
    /* incomplete -- malloc's return value not checked */
    strcpy(copy, somestring);
    puts(copy);
    return 0;
}
```

Reserve memory location after deciding the string size linelen

Reserve memory location after deciding the string size linelen

Checking what malloc returns !

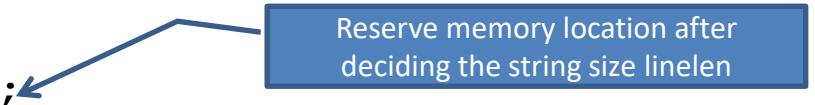
- When malloc is unable to allocate the requested memory, it returns a *null pointer*. Therefore, whenever you call malloc, it's vital to check the returned pointer before using it !

```
char * line = malloc(linelen); ;
if(line == NULL) {
        printf("out of memory\n");
        return; // exits current function
}
```

```
char * line = malloc(linelen);
if(line == NULL) {
        printf("out of memory\n");
        exit(1); // exits all nested function calls,
                 //  terminates program
}
```

Checking what malloc returns !

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char* line;
    int linelen;
    printf("How long is your line?\n");
    scanf("%d\n", &linelen);
    line = malloc(linelen*sizeof(char));
    if(line == NULL) {
        printf("out of memory\n");
        return; // exits current function
    }
    gets(line);
    puts(line);

    return 0;
}
```



Dynamic allocation of arrays

- Dynamic allocation of an array of N elements of type TIP:
- `TIP * p;`
- `p= (TIP *) malloc(N*sizeof(TIP));`
- Pointer p will point to a memory block big enough to hold N elements of type TIP.
- Variable p can be used in the same way as if it was declared:
- `TIP p[N];`

Example: dynamic allocation of arrays

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int n;
    int * tab;
    int i;
    printf("Input number of elements: \n");
    scanf("%d", &n);
    if ((tab=malloc(n * sizeof(int)))==NULL) {
        printf("Memory allocation error !\n");
        exit(1);
    }
    for (i=0; i<n; i++)
        scanf("%d", &tab[i]);
    for (i=0; i<n; i++)
        printf("%d ", tab[i]);
    free(tab);
    return 1;
}
```

dynamic allocation of character strings

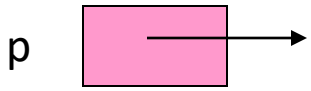
```
void main(void)
{
    char sir1[40];
    char *sir2;
    printf("Enter a string: \n");
    scanf("%40s", sir1);
    if ((sir2=(char *)malloc(strlen(sir1)+1))==NULL) {
        printf("memory allocation error !\n");
        exit(1);
        }
    strcpy(sir2, sir1);
    printf("The copy is: %s \n", sir2);

}
```

Lifetime of dynamic allocated memory

- Memory allocated with malloc lasts as long as you want it to. It does not automatically disappear when a function returns, as automatic variables do:

```
void fct(void) {
        int *p;
        p=(int*) malloc(10*sizeof(int));
        free(p);
        return;
}
```

The memory area allocated here remains occupied also after the function call is ended !
Only the pointer variable p accessing it disappears.



Lifetime example

- Have a function create and return an array of values:

```
int * fct(void) {
        int *p;
        p=(int*) malloc(10*sizeof(int));
        // fill p with values . . .
        return p;
}
```

YES !

```
int * fct(void) {
        int p[10];
        // fill p with values . . .
        return p;
}
```

NO !

adding 2 vectors example

```
 int * add_vector4(int *a, int *b, int n) {
    int * r;
    r=(int *) malloc(sizeof (int) * n);
    if (r==NULL) exit(1);
     int i;
     for (i=0; i<n; i++, a++, b++)
             r[i]=*a+*b;
      return r;
      }

 main(){
    int a[3]={1,2,3};
    int b[3]={4,5,6};
    int * rr;
    rr=add_vector4(a,b,3);
    print_vector(rr,3);
}
```

```
void print_vector(int* a, int n) {
    int i;
    for (i = 0; i < n; i++, a++)
        printf("%d\n", *a);

}
```

free()

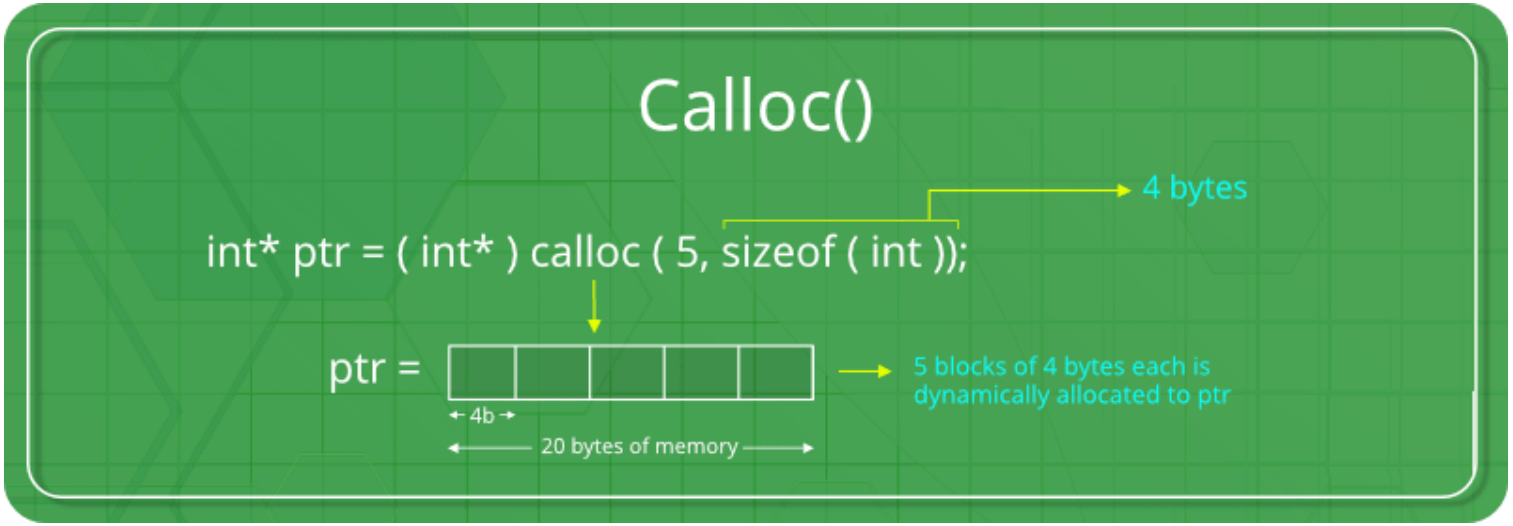
- Dynamically allocated memory is deallocated with the free function. If p contains a pointer previously returned by malloc, you can call

  ```
  free(p);
  ```

- which will ``give the memory back'' to the heap of memory from which malloc requests are satisfied.
- the memory you give back by calling free() is immediately usable by other parts of your program. (Theoretically, it may even be usable by other programs.)
- When your program exits, any memory which it has allocated but not freed should be automatically released by the operating system.
- Once you've freed some memory you must remember not to use it any more. After calling `free(p)` it is probably the case that p still points at the same memory. However, since we've given it back, it's now ``available,'' and a later call to `malloc` might give that memory to some other part of your program.

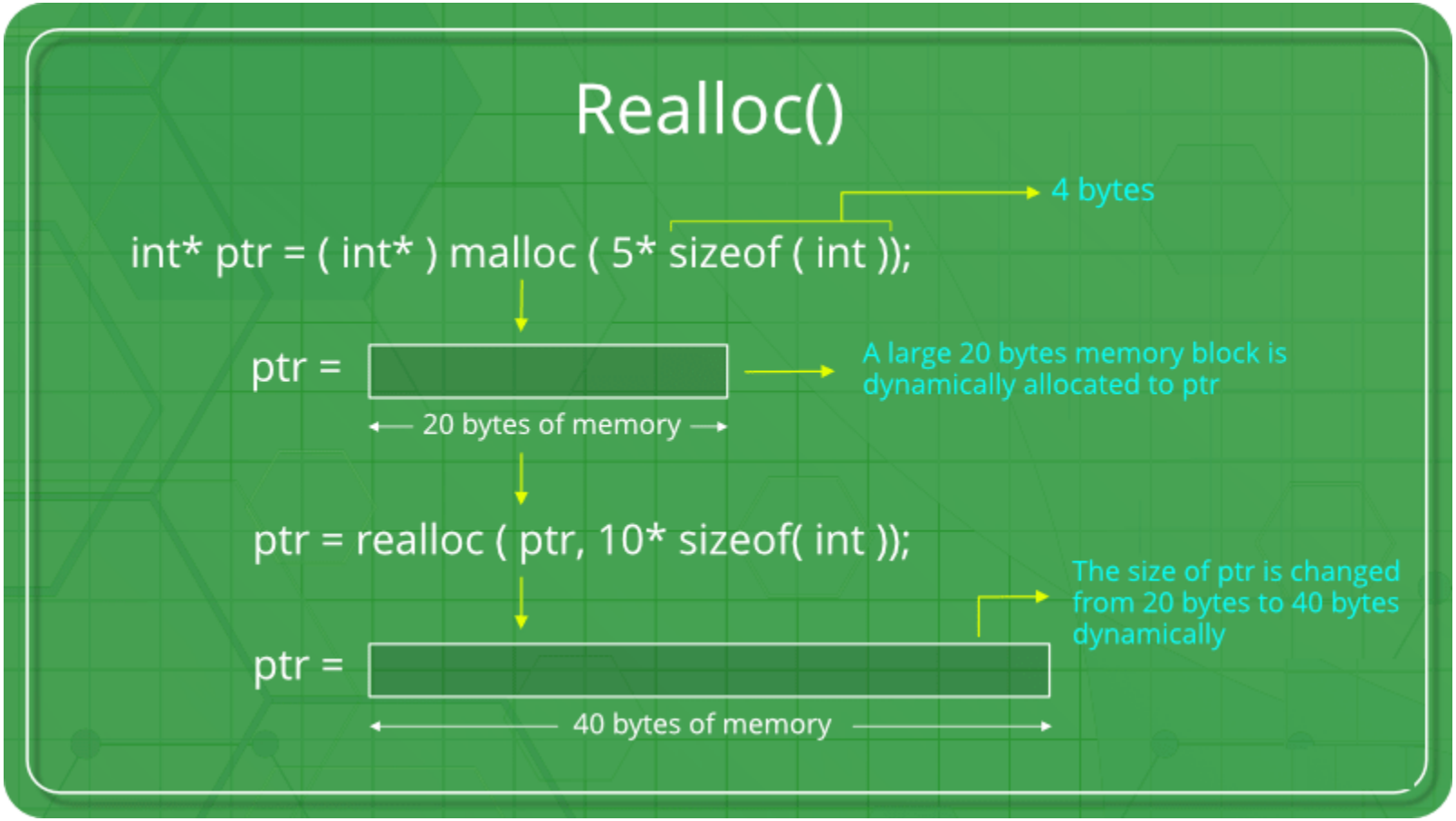
calloc( )

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.



realloc ()

- Change reserved dynamic memory to a new size



realloc () Example

```c
int main()
{

   // This pointer will hold the
   // base address of the block created
   int* ptr;
   int n, i;

   // Get the number of elements for the array
   n = 5;
   printf("Enter number of elements: %d\n", n);

   // Dynamically allocate memory using calloc()
   ptr = (int*)calloc(n, sizeof(int));

   // Check if the memory has been successfully
   // allocated by malloc or not
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   } else {

      // Memory has been successfully allocated
      printf("Memory successfully allocated using calloc.\n");

      // Get the elements of the array
      for (i = 0; i < n; ++i) {
         ptr[i] = i + 1;
      }

      // Print the elements of the array
      printf("The elements of the array are: ");
      for (i = 0; i < n; ++i) {
         printf("%d, ", ptr[i]);
      }

      // Get the new size for the array
      n = 10;
      printf("\n\nEnter the new size of the array: %d\n", n);

      // Dynamically re-allocate memory using realloc()
      ptr = realloc(ptr, n * sizeof(int));

      // Memory has been successfully allocated
      printf("Memory successfully re-allocated using realloc.\n");

      // Get the new elements of the array
      for (i = 5; i < n; ++i) {
         ptr[i] = i + 1;
      }

      // Print the elements of the array
      printf("The elements of the array are: ");
      for (i = 0; i < n; ++i) {
         printf("%d, ", ptr[i]);
      }

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

Structures and pointers

- Structure pointers are just like pointers to ordinary variables. The declaration

```
struct point *pp;
```

- says that pp is a pointer to a structure of type struct point. If pp points to a point structure, *pp is the structure, and (*pp).x and (*pp).y are the members.
- To use pp:

```
struct point origin, *pp;
pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

- The parentheses are necessary in (*pp).x because the precedence of the structure member operator . is higher then *. The expression *pp.x means *(pp.x), which is illegal here because x is not a pointer.

Pointers to structures

- Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then referring to a particular member can be done by:
- p->*member-of-structure*
- Equivalent with `(*p).member-of-structure`

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

- Both . and -> associate from left to right, so if we have

```
struct rect r, *rp = &r;
```

- then these four expressions are equivalent:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Pointers to structures Example

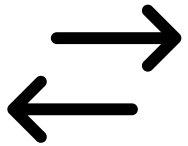
```
#include <stdio.h>
#include <stdlib.h>
struct person {
   int age;
   float weight;
   char name[30];
};

int main()
{
   struct person *ptr;
   int i, n;

   printf("Enter the number of persons: ");
   scanf("%d", &n);

  // allocating memory for n numbers of person
 ptr = (struct person*) malloc(n *
                      sizeof(struct person));
```

```
 for(i = 0; i < n; ++i)
   {
      printf("Enter first name and age
respectively: ");

       // To access members of 1st struct person,
       // ptr->name and ptr->age is used

       // To access members of 2nd struct person,
       // (ptr+1)->name and (ptr+1)->age is used
     scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
   }

   printf("Displaying Information:\n");
   for(i = 0; i < n; ++i)
       printf("Name: %s\tAge: %d\n", (ptr+i)->name,
(ptr+i)->age);

   return 0;
}
```

Precedence of operators

- The structure operators . and ->, together with () for function calls and [] for subscripts, are at the top of the precedence hierarchy and thus bind very tightly.
- For example, given the declaration

```
struct String {
        int len;
        char *str;
    } *p;
```

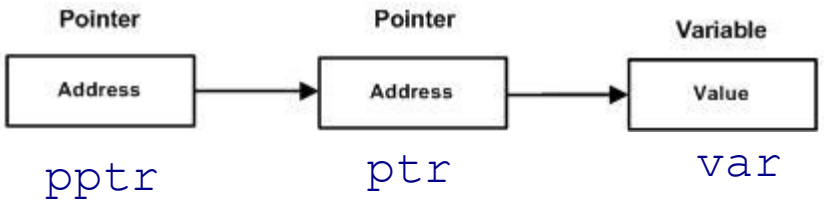


```
struct String {
        int len;
        char *str;
    };
struct String * p;
```

- then `++p->len` increments len, not p, because the implied parenthesizing is `++(p->len)` .
- Parentheses can be used to alter binding: (++p)->len increments p before accessing len, and (p++)->len increments p afterward. (This last set of parentheses is unnecessary.
- `*p->str` fetches whatever str points to;
- `*p->str++` increments str after accessing whatever it points to (just like *s++)
- `(*p->str)++` increments whatever str points to
- `*p++->str` increments p after accessing whatever str points to.

Pointers to Pointers

- Since pointers are variables themselves, they can be stored in arrays just as other variables can
  - The declaration of a pointer-to-pointer looks like




```
int main () {
int var;
int *ptr;
int **pptr;
var = 3000;
/* take the address of var */
 ptr = &var;
/* take the address of ptr using address of operator & */
pptr = &ptr;
/* take the value using pptr */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);
return 0;
 }
```

Pointers to Pointers

```
int getValueOf5(int *p)
{
  *p = 5;
  return 1;//success
}

int get1024HeapMemory(int **p)
{
  *p = malloc(1024);
  if(*p == 0)
    return -1;//error
  else
    return 0;//success
}
```

```
void main (){
int x;
getValueOf5(&x);
//I want to fill the int variable, so I pass
//it's address in
//At this point x holds 5

int *p;
get1024HeapMemory(&p);//I want to fill the int*
//variable, so I pass it's address in
//At this point p holds a memory address where
//1024 bytes of memory is allocated on the heap
}
```

Summary & Discussion

- **Explained Dynamic Memory Allocation**
- **Explained Dynamic Memory Allocation of Arrays, Strings**
- **Explained Lifetime Of Dynamic Allocated Memory**
- **Explained Pointers to Structures**
- **Explained Pointers to Pointers**