Dynamic Memory

Dynamic Memory Allocation

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Problem definition

- The compiler allocates a predefined quantity of memory for each variable defined in the source program
  - The **quantity** of the memory depends on the **type** of the variable and the its **size**
- However, in many situations, it is not clear how much memory (or objects) the program will actually need

Problem definition

- For example, with arrays
  - We may declare an array to be large enough to hold the maximum number of elements we expect our application to handle
    - If too much memory is allocated and then not used, there is a waste of memory
    - If not enough memory is allocated, the program is not able to fully handle the input data
  - This is a strong limitation !

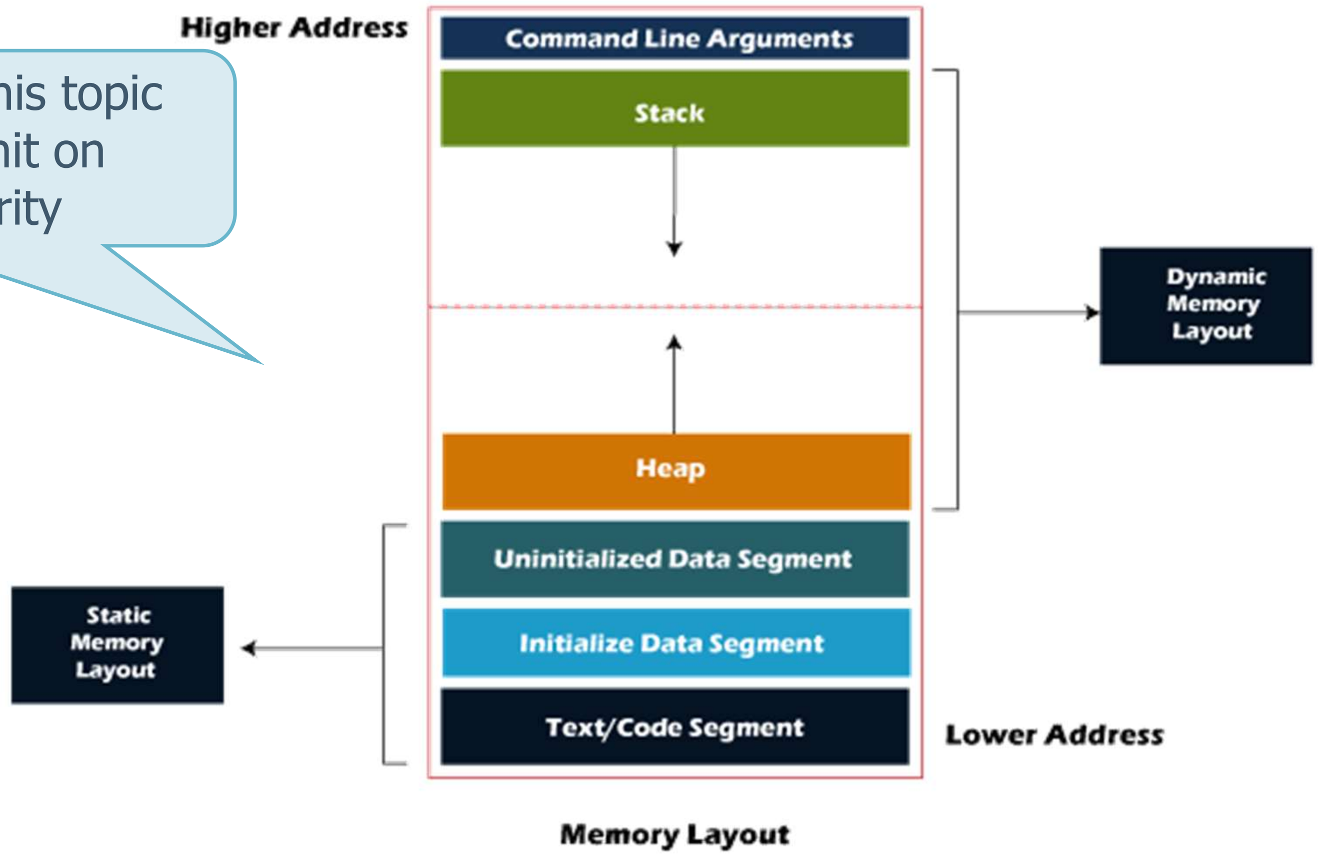
Problem definition

- To make our program more flexible we should be able (during execution) to
  - Allocate initial and additional memory when needed
  - Free up the allocated memory when it is no more needed
    - As allocations can make the system run out of memory
- Allocation of memory during execution is called dynamic memory allocation

Problem definition

- Dynamic memory is allocated in a dedicated area of the system
  - The heap

More information on this topic will follow on the unit on ADTs and Modularity



Problem definition

- A family of four functions allows programs to manage dynamically allocated memory on the heap
  - Malloc
  - Calloc
  - Realloc
  - Free
- In order to use these functions you have to include the **stdlib.h** header file in your program
  - The operator **sizeof** is also essential to dynamic memory allocation to compute the quantity of memory to reserve

Function malloc

This represents an unsigned value generally defined as **unsigned int**

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

- Function malloc
  - Takes as an argument the **number of bytes** to be allocated
  - Returns a pointer of type **void *** to the allocated memory
    - This pointer is just a byte address and it does not point to an object of a specific type
    - A void * pointer may be assigned to a variable of any pointer type, i.e., we have to type cast the value to the type of the destination pointer

Examples

After the allocation the only way to access the data is through the pointer

```
char *p;
...
p = (char *) malloc (100);
```

Explicit cast

```
char *p;
...
p = malloc (100);
```

Implicit cast

Function malloc is normally used with the **sizeof** operator to allocate a proper quantity of memory

```
char *p1, *p2;
...
p1 = (char *) malloc (sizeof (char));
p2 = malloc (sizeof (*p2));
```

Type

Data pointed

Examples

Same process with all other data types

Type or data pointed

```
struct mys *p1, *p2;
...
p1 = (struct mys *) malloc (sizeof (struct mys));
p2 = malloc (sizeof (p2 *));
```

```
char *p;
...
p = malloc (100 * sizeof (char));
```

All objects stored in the chunk of memory reserved are usually of the same type

```
struct node *newPtr;
...
newPtr = malloc (n * sizeof (struct node));
```

Function malloc

This represents an unsigned value generally defined as **unsigned int**

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

- When the allocation cannot be performed (there is not enough memory) function malloc returns NULL
  - We must always **check** the correctness of the operation
    - If NULL either exit the program or modify the allocation request

Example

```
int *p;
```

In general, this definition without a proper memory allocation (or a proper assignment to p) is **useless**

It states that there is a variable pointer named **p** somewhere in memory

Example

```
int *p;
int i;
int v[100];

p = &i;

P = &v[10];
```

We can make **p** point to something already in place, i.e., a static object

OR
we can create a new object to with p points

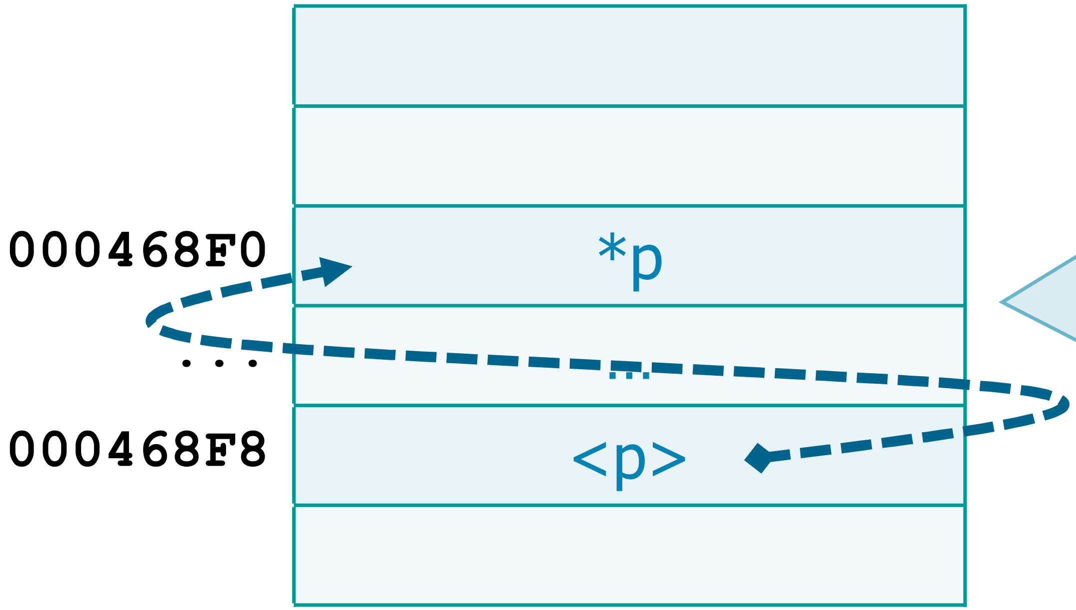
It states that there is a variable pointer named **p** somewhere in memory

Example

```
int *p;
p = (int *) malloc (1 * sizeof(int));
```

OR
we can create a new object to with p points

If it is unsuccessful, the programmer has to figure out how to behave



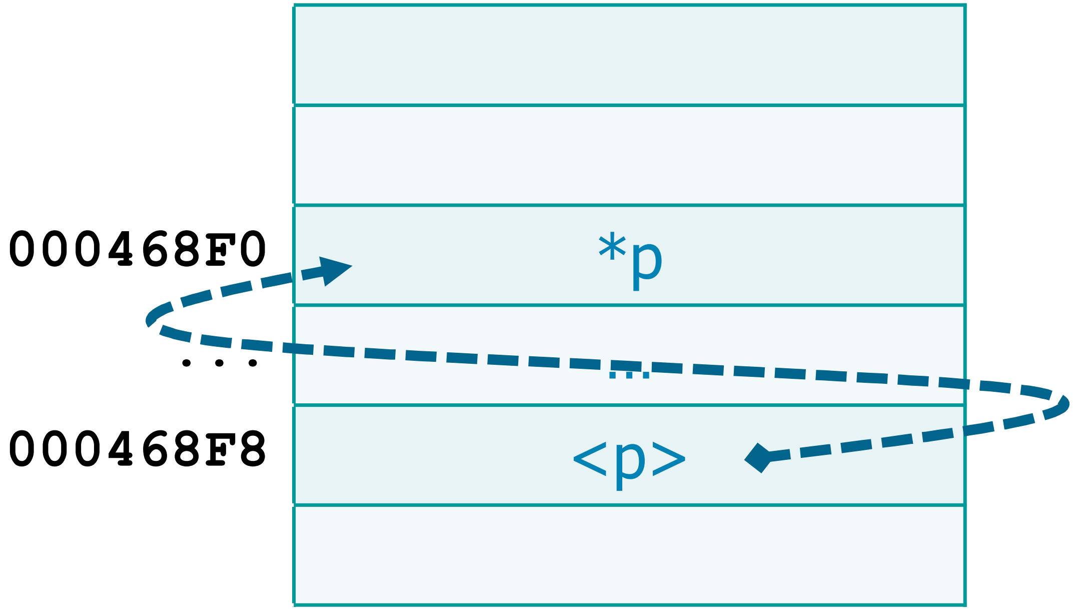
If the allocation is successful, objects in dynamically allocated memory can be accessed indirectly by dereferencing the pointer, appropriately cast to the type of required pointer

Example

```
int *p;
p = (int *) malloc (1 * sizeof(int));
if (p == NULL) {
   fprintf (stderr, "Memory allocation error.\n");
   exit (1);
}
fprintf (stdout, "Introduce an integer value: ");
scanf ("%d", p);
...
```

Always verify the validity of the pointer

p not &p



Common errors

```
int *p1;
char *p2;
...

p2 = malloc (sizeof (int));
...
p1 = (int *) malloc (sizeof (int *));
...
p1 = (int) malloc (sizeof (int));
...
p1 = (int *) malloc (sizeof (char));
...
p1 = (int *) malloc (10);
...
```

p2 is of type char *

Must be (int *) and (int)

Must be (int *) and (int)

p1 is of type int

10 bytes or (10 * sizeof(int)) ?

Function calloc

- Function malloc can be sufficient
  - Anyway, the library stdlib.h provides two other functions for dynamic memory allocation
- Function calloc (clear alloc)
  - Dynamically allocates memory
  - Initializes this memory

Function calloc

```
void *calloc (size_t number_of_objects, size_t size);
```

- Function calloc
  - The two arguments represent
    - The number of elements number_of_objects
    - The size of each element size
  - Returns a pointer of type void * to the allocated memory
    - Integer values are initialize with 0
    - Character values with '0' (i.e, NULL, ASCII character 48)

Function calloc

- Function calloc corresponds to a malloc followed by an initialization phase
  - Initialization is not for free, and it has a linear (in the memory size) cost
    - For a chunk of n slots, calloc has a cost of O(n)
  - Thus, is initialization is not required (i.e., done in some other way) malloc is more efficient

Examples

```
int *test;
...
test = calloc (5, sizeof(int));
if (test==NULL) {
  fprintf (stderr, "Memory allocation error.\n");
  exit (1);
}
```

Equivalent

```
int i, *test;
...
test = malloc (5 * sizeof(int));
if (test==NULL) {
  fprintf (stderr, "Memory allocation error.\n");
  exit (1);
}
for (i=0; i<5; i++)
  test[i] = 0;
```

Function realloc

- Function realloc changes the size of an object allocated by a previous call to malloc, calloc or realloc
  - The original block is **extended** and the extra space is placed **at the end** of the previous block
    - The original content is not modified **iff** the amount of memory allocated is larger than the original
  - If this is impossible, it reallocates the previous memory space to a different memory area
    - In this case, it copies all data from the original memory area to the new one
    - The **cost is potentially linear** in the amount of memory copied

Function realloc

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

- Function realloc
  - The two arguments are
    - A pointer to the original object ptr
      - If ptr is NULL, realloc works like malloc
    - The new size of the object size
  - Returns a pointer to the newly allocated memory or NULL

Examples

```
int *v1, *v2, *v3;
...
v1 = malloc (50 * sizeof (int));
if (v1 == NULL) { ... }

...


v2 = realloc (v1, 100 * sizeof (int));
if (v2 == NULL) { ... }

...

v3 = realloc (v2, 200 * sizeof (int));
if (v3 == NULL) { ... }

...
```

Doubling the quantity of bytes is somehow common to many applications

Function free

```
void free (void *pointer);
```

- Function free
  - Deallocates memory, i.e., the memory is returned to the system so that it can be reallocated in the future
  - This function must only be used to release memory assigned with malloc, calloc, realloc
    - Static arrays **cannot** be freed
    - **Never** free a static data structure

Example

```
int n, *p;
int v[100];

fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);

p = (int *) malloc (n * sizeof (int));
if (p == NULL) {
   fprintf (stderr, "Memory allocation error.\n");
   exit (1);
}

...

free (p);
free (v);
```

Use p to access the allocated area

When the ares is no longer needed free it

Wrong ! No !

Final considerations

- Any chunk of memory cannot be used after it has been freed
  - It may be there, thus accessing it may give correct results, but the programmer does not own it any more and the operating system can use it as it likes
- Pay attention to memory leaks
  - Lost pointers imply a memory loss
  - Memory losses accumulate till the exhaust the available memory

Final considerations

- All dynamically allocated data structure must be free
  - The are usually released and returned to the operating system, as soon as the program ends
  - Anyway, it is good practice to **explicitly** free dynamically allocated memory **as soon as** the memory is not required any more
    - This reduces the possibilities to run out of memory
    - As a consequence, the number of **free** in a program should **match** the number of **malloc** plus the number of **calloc**