Dynamic Memory

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Memory areas in C

C has four distinct areas of memory:

- the constant data area;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Constant data area:

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Constant data area:

- stores strings and constants and data whose values are known at compile time;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Constant data area:

- stores strings and constants and data whose values are known at compile time;
- is read only, the result of trying to modify it are undefined.

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Static-extent data area:

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Static-extent data area:

- is used to store variables that are defined extern or static;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Static-extent data area:

- is used to store variables that are defined extern or static;
- exists for the lifetime of the program;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Static-extent data area:

- is used to store variables that are defined extern or static;
- exists for the lifetime of the program;
- can be modified.

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Constant and static-extent data area are managed by the compiler, are allocated when program begins and destroyed when it terminates.

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Stack memory:

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Stack memory:

- is used to store local variables (the ones with automatic extent);

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Stack memory:

- is used to store local variables (the ones with automatic extent);
- is allocated at the point a variable is defined and released when it goes out-of-scope;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Stack memory:

- is used to store local variables (the ones with automatic extent);
- is allocated at the point a variable is defined and released when it goes out-of-scope;
- follows a LIFO policy:

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Stack memory:

- is used to store local variables (the ones with automatic extent);
- is allocated at the point a variable is defined and released when it goes out-of-scope;
- follows a LIFO policy:
  - when variables are defined they are pushed onto the stack;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Stack memory:

- is used to store local variables (the ones with automatic extent);
- is allocated at the point a variable is defined and released when it goes out-of-scope;
- follows a LIFO policy:
  - when variables are defined they are pushed onto the stack;
  - at the end of a block, all the variables that go out-of-scope are popped off the stack.

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Heap memory:

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Heap memory:

- is used for dynamically allocated storage;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Heap memory:

- is used for dynamically allocated storage;
- is managed directly by the programmer;

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Heap memory:

- is used for dynamically allocated storage;
- is managed directly by the programmer;
- there is not any support provided by compiler to manage di area!

Memory areas in C

C has four distinct areas of memory:

- the constant data area;
- the static-extent data area;
- the stack;
- the heap.

Heap memory:

- is used for dynamically allocated storage;
- is managed directly by the programmer;
- there is not any support provided by compiler to manage di area!

WARNING!

Memory Allocation Functions (1/4)

There are two main function for memory allocations:

```
void *malloc( size_t )
void free( void * )
```

Memory Allocation Functions (1/4)

There are two main function for memory allocations:

```
void *malloc( size_t )
void free( void * )
```

Function malloc allocate the number of bytes passed as parameters and returns a pointer to the allocated memory area.

Memory Allocation Functions (1/4)

There are two main function for memory allocations:

```
void *malloc( size_t )
void free( void * )
```

Function malloc allocate the number of bytes passed as parameters and returns a pointer to the allocated memory area.
The returned datatype is void* which represents a generic pointer.

Memory Allocation Functions (1/4)

There are two main function for memory allocations:

```
void *malloc( size_t )
void free( void * )
```

Function malloc allocate the number of bytes passed as parameters and returns a pointer to the allocated memory area.
The returned datatype is void* which represents a generic pointer.

Function free allows to release memory that has been allocated with malloc.

Memory Allocation Functions (2/4)

To dynamically create an array of 10 integers, one can write:

```
int *p = malloc( 10 * sizeof( int ) );
```

Memory Allocation Functions (2/4)

To dynamically create an array of 10 integers, one can write:

```
int *p = malloc( 10 * sizeof( int ) );
```

Explicit cast is not needed, however it is common to add it:

```
int *p = (int*) malloc( 10 * sizeof( int ) );
```

Memory Allocation Functions (2/4)

To dynamically create an array of 10 integers, one can write:

```
int *p = malloc( 10 * sizeof( int ) );
```

Explicit cast is not needed, however it is common to add it:

```
int *p = (int*) malloc( 10 * sizeof( int ) );
```

Memory allocation may fail! In this case value NULL is returned.

Memory Allocation Functions (2/4)

To dynamically create an array of 10 integers, one can write:

```
int *p = malloc( 10 * sizeof( int ) );
```

Explicit cast is not needed, however it is common to add it:

```
int *p = (int*) malloc( 10 * sizeof( int ) );
```

Memory allocation may fail! In this case value NULL is returned.

To release the memory allocated above, function free is used:

```
free( p );
```

Memory Allocation Functions (3/4)

Another function that can be used to allocate memory in the heap is:

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

Memory Allocation Functions (3/4)

Another function that can be used to allocate memory in the heap is:

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

While malloc allocates an area of memory and fills it with unspecified values, calloc guarantees that all the items in the allocated area are set to 0

Memory Allocation Functions (3/4)

Another function that can be used to allocate memory in the heap is:

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

While malloc allocates an area of memory and fills it with unspecified values, calloc guarantees that all the items in the allocated area are set to 0:

- n is the number of copies to allocate;
- size is the size of each copy.

Memory Allocation Functions (3/4)

Another function that can be used to allocate memory in the heap is:

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

While malloc allocates an area of memory and fills it with unspecified values, calloc guarantees that all the items in the allocated area are set to 0:

- n is the number of copies to allocate;
- size is the size of each copy.

```
int *p = calloc(10, sizeof(int));
```

Memory Allocation Functions (4/4)

Function realloc is used change the size of an existing block of dynamically allocated memory:

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

Memory Allocation Functions (4/4)

Function realloc is used change the size of an existing block of dynamically allocated memory:

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

where p is a pointer to the current block of memory (allocated with a malloc, calloc) and size is the new requested size.

Memory Allocation Functions (4/4)

Function realloc is used change the size of an existing block of dynamically allocated memory:

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

where p is a pointer to the current block of memory (allocated with a malloc, calloc) and size is the new requested size.

The return value is a pointer to the resized memory block, or NULL if the request fails.

Memory Allocation Functions (4/4)

Function realloc is used change the size of an existing block of dynamically allocated memory:

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

where p is a pointer to the current block of memory (allocated with a malloc, calloc) and size is the new requested size.

The return value is a pointer to the resized memory block, or NULL if the request fails.

If realloc() is passed a size request of 0, then the memory pointed to by p is released, and realloc() returns NULL.

Memory Allocation Functions: Example

Write function `string_duplicate` that received in input a string performs a copy of the string in a new.

Memory Allocation Functions: Example

Write function string_duplicate that received in input a string performs a copy of the string in a new.

Solution:

```
char *string_duplicate(char *s)
{
  char *p = malloc(strlen(s) + 1);
  return strcpy(p, s);
}
```

Memory Allocation Functions: Example

Write function string_duplicate that received in input a string performs a copy of the string in a new.

Solution:

```
char *string_duplicate(char *s)
{
  char *p = malloc(strlen(s) + 1);
  return strcpy(p, s);
}
```

This solution is not correct!

Memory Allocation Functions: Example

Write function string_duplicate that received in input a string performs a copy of the string in a new.

Solution:

```
char *string_duplicate(char *s)
{
  char *p = malloc(strlen(s) + 1);
  return strcpy(p, s);
}
```

This solution is not correct!The result of malloc may be null!

Memory Allocation Functions: Example

Solution 2:

```
char *string duplicate(char *s)
{
  char *p = malloc(strlen(s) + 1);
  if (p != NULL) {
    strcpy(p,s);
  }
  return p;
}
```

Memory Allocation Functions: Example

Solution 2:

```
char *string duplicate(char *s)
{
  char *p = malloc(strlen(s) + 1);
  if (p != NULL) {
    strcpy(p,s);
  }
  return p;
}
```

Warning: To avoid memory-leak, the calling function has the responsibility to `free` the allocated memory!

Memory Allocation Functions: Example

Solution 2:

```
char *string duplicate(char *s)
{
  char *p = malloc(strlen(s) + 1);
  if (p != NULL) {
    strcpy(p,s);
  }
  return p;
}
```

Warning: To avoid memory-leak, the calling function has the responsibility to `free` the allocated memory!

```
char *s;
s = string_duplicate("this is a string");
...
free(s);
```

List of common errors (1/2)

List of common errors (1/2)

Dereferencing a pointer with an invalid address (Memory Corruption):

```
int *p;
int z = *p;
```

List of common errors (1/2)

Dereferencing a pointer with an invalid address (Memory Corruption):

```
int *p;
int z = *p;
```

Dereferencing a pointer that has been freed:

```
int *p;
...
free(p);
z = *p;
```

List of common errors (1/2)

Dereferencing a pointer with an invalid address (Memory Corruption):

```
int *p;
int z = *p;
```

Dereferencing a pointer that has been freed:

```
int *p;
...
free(p);
z = *p;
```

Dereferencing a NULL pointer:

```
int *p = NULL;
z = *p;
```

List of common errors (2/2)

Freeing memory that has already been freed:

```
...
free(p);
... //No new allocation of p!
free(p);
```

List of common errors (2/2)

Freeing memory that has already been freed:

```
...
free(p);
... //No new allocation of p!
free(p);
```

Freeing a pointer to memory that was not dynamically allocated:

```
int z = 10;
int *p = &z;
...
free(p);
```

List of common errors (2/2)

Freeing memory that has already been freed:

```
...
free(p);
... //No new allocation of p!
free(p);
```

Freeing a pointer to memory that was not dynamically allocated:

```
int z = 10;
int *p = &z;
...
free(p);
```

Failing to free dynamically allocated memory.

List of common errors (2/2)

Freeing memory that has already been freed:

```
...
free(p);
... //No new allocation of p!
free(p);
```

Freeing a pointer to memory that was not dynamically allocated:

```
int z = 10;
int *p = &z;
...
free(p);
```

Failing to free dynamically allocated memory.

Attempting to access memory beyond the bounds of the allocated block.

Good practices

Good practices

Every malloc() should have an associated free().

Good practices

Every malloc() should have an associated free().

Pointers should be initialised when defined (either with a valid address or NULL).

Good practices

Every malloc() should have an associated free().

Pointers should be initialised when defined (either with a valid address or NULL).

Pointers should be assigned NULL after being freed.

Good practices

Every malloc() should have an associated free().

Pointers should be initialised when defined (either with a valid address or NULL).

Pointers should be assigned NULL after being freed.

If the above rule are used, many of the common errors are avoided with a NULL-check!

To be continued...

Structures and Unions

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Structures

A structure is declared using the keyword `struct`, and the internal organisation of the structure is defined by a set of variables enclosed in braces:

```
struct Point {
    int x;
    int y;
};
```

Structures

A structure is declared using the keyword `struct`, and the internal organisation of the structure is defined by a set of variables enclosed in braces:

```
struct Point {
    int x;
    int y;
};
```

By convention, structures should always be named with an uppercase first letter.

Structures

A structure is declared using the keyword `struct`, and the internal organisation of the structure is defined by a set of variables enclosed in braces:

```
struct Point {
   int x;
   int y;
};
```

By convention, structures should always be named with an uppercase first letter.

The variables x and y are called members of the structure named Point.

Structures

Variables of type Point may be defined as a list of identifiers at the end of the struct definition:

```
 struct Point {
        int x;
        int y;
} p1, p2, p3;
```

Structures

Variables of type Point may be defined as a list of identifiers at the end of the struct definition:

```
 struct Point {
       int x;
       int y;
} p1, p2, p3;
```

or as subsequent definitions using the tag struct Point:

```
struct Point p1, p2, p3;
```

Structures

Variables of type Point may be defined as a list of identifiers at the end of the struct definition:

```
 struct Point {
        int x;
        int y;
} p1, p2, p3;
```

or as subsequent definitions using the tag struct Point:

```
struct Point p1, p2, p3;
```

When a structure is defined, its members may be initialised using brace notation:

```
struct Point topleft = { 320, 0 };
```

Structures

Individual members of a struct may be accessed via the member operator .:

```
struct Point topleft;
topleft.x = 320;
topleft.y = 0;
```

Structures

Individual members of a struct may be accessed via the member operator .:

```
struct Point topleft;
topleft.x = 320;
topleft.y = 0;
```

Structures can be nested:

```
struct Rectangle {
   struct Point topleft;
   struct Point bottomright;
};
```

Structures

Individual members of a struct may be accessed via the member operator .:

```
struct Point topleft;
topleft.x = 320;
topleft.y = 0;
```

Structures can be nested:

```
struct Rectangle {
   struct Point topleft;
   struct Point bottomright;
};
```

To access the lowest-level members of a variable of type Rectangle, therefore, requires two instances of the member operator

```
struct Rectangle rect;
rect.topleft.x = 50;
```

Operations on Structures

Operations on Structures

The operations permitted on structures are a subset of the operations permitted on basic types.

Operations on Structures

The operations permitted on structures are a subset of the operations permitted on basic types.

Structures may be copied or assigned, but it is not possible to directly compare two structures.

```
p2 = p1;         /* Valid. structs may be assigned. */
if (p1 == p2)   /* Invalid. structs may not be compared. */
    printf("Points are equal\n");
if (p1.x == p2.x && p1.y == p2.y)
    /* Valid. May compare basic types. */
    printf("Points are equal\n");
```

Operations on Structures

A structure may be passed to a function and may be returned by a function:

```
struct Point point_difference(struct Point p1, struct Point
    p2)
/* Return the delta (dx, dy) of p2 with respect to p1. */
{
        p2.x -= p1.x;
        p2.y -= p1.y;
        return p2;
}
```

Operations on Structures

A structure may be passed to a function and may be returned by a function:

```
struct Point point_difference(struct Point p1, struct Point
    p2)
/* Return the delta (dx, dy) of p2 with respect to p1. */
{
        p2.x -= p1.x;
        p2.y -= p1.y;
        return p2;
}
```

As with any other variable, structures are passed by value!

```
struct Point a = {5,10}, b = {20,30}, c;
c = point_difference(a, b);
/* c = {15,20}, b is unchanged. */
```

Structures and Poiters

Passing structures by value can be inefficient if the structure is large, and it is generally more efficient to pass a pointer to a struct rather than making a copy.

```
struct Point pt = { 50, 50 };
struct Point *pp;
pp = &pt;
(*pp).x = 100; /* pt.x is now 100. */
```

Structures and Poiters

Passing structures by value can be inefficient if the structure is large, and it is generally more efficient to pass a pointer to a struct rather than making a copy.

```
struct Point pt = { 50, 50 };
struct Point *pp;
pp = &pt;
(*pp).x = 100; /* pt.x is now 100. */
```

The parentheses about (*pp).x are necessary to enforce the correct order-of-evaluation!

Structures and Poiters

Passing structures by value can be inefficient if the structure is large, and it is generally more efficient to pass a pointer to a struct rather than making a copy.

```
struct Point pt = { 50, 50 };
struct Point *pp;
pp = &pt;
(*pp).x = 100; /* pt.x is now 100. */
```

The parentheses about `(*pp).x` are necessary to enforce the correct order-of-evaluation!

The `->` operator permits the expression `(*pp).x` to be rewritten more simply as `pp->x`.

Self-referential Structures

A structure definition may not contain an object of its own type.

Self-referential Structures

A structure definition may not contain an object of its own type.

```
struct List {
  int item;
  struct List next; /* Invalid. Cannot define an object of an
     incomplete type. */
}
```

Self-referential Structures

A structure definition may not contain an object of its own type.

```
struct List {
  int item;
  struct List next; /* Invalid. Cannot define an object of an
     incomplete type. */
}
```

However, it may refer to a pointer of its own type:

Self-referential Structures

A structure definition may not contain an object of its own type.

```
struct List {
  int item;
  struct List next; /* Invalid. Cannot define an object of an
     incomplete type. */
}
```

However, it may refer to a pointer of its own type:

```
struct List {
  int item;
  struct *List next;
}
```

Exercise: List Operations

Write *C* library that implements basic list operations:

- `list.h` with type and functions declarations;
- `list.c` with all the definitions.

Typedefs

The keyword typedef provides a means for creating new data type names:

```
typedef int Length;
```

Typedefs

The keyword `typedef` provides a means for creating new data type names:

```
typedef int Length;
```

This makes the name `Length` a synonym for `int`.

Typedefs

The keyword typedef provides a means for creating new data type names:

```
typedef int Length;
```

This makes the name Length a synonym for int.

The ability to define type synonyms permits a significant improvement in structure declaration syntax:

```
typedef struct Point {
  int x;
  int y;
} Point;

Point pt1, pt2;
```

Typedefs and Self-referential Structures

This simplification enabled by typedef is more marked for self-referencing structures:

```
typedef struct list_t List;
struct list_t {
  int item;
  List *next;
};
```

Union Types

The declaration of a union type is similar to the declaration of a struct type:

```
union Utype {
  int ival;
  float fval;
  char *sval;
};

union Utype x, y, z;
```

Union Types

The declaration of a union type is similar to the declaration of a struct type:

```
union Utype {
  int ival;
  float fval;
  char *sval;
};

union Utype x, y, z;
```

Accessing members of a union type is also the same as for structures, with the . member operator for union objects and the -> operator for pointers to union objects.

Structures vs Unions

Differences:

- a struct defines a group of related variables and provides storage for all of its members;
- a union provides storage for a single variable, which may be one of several types.

Structures vs Unions

Differences:

- a `struct` defines a group of related variables and provides storage for all of its members;
- a `union` provides storage for a single variable, which may be one of several types.

In the Utype example, the compiler will allocate sufficient memory to store the largest of the types `int`, `float`, and `char` $*$.

Structures vs Unions

Differences:

- a `struct` defines a group of related variables and provides storage for all of its members;
- a `union` provides storage for a single variable, which may be one of several types.

In the Utype example, the compiler will allocate sufficient memory to store the largest of the types `int`, `float`, and `char` ∗.

A Utype variable holds a value for one of the three possible types!

Structures vs Unions

Differences:

- a `struct` defines a group of related variables and provides storage for all of its members;
- a `union` provides storage for a single variable, which may be one of several types.

In the Utype example, the compiler will allocate sufficient memory to store the largest of the types `int`, `float`, and `char` *.

A Utype variable holds a value for one of the three possible types!

It is the programmers responsibility to keep track of which type that might be!

Union Types: Example

```
typedef union { /* Heterogeneous type. */
 int ival;
 float fval;
} Utype;

enum { INT, FLOAT }; /* Define type tags. */

typedef struct {
 int type;  /* Tag for the current stored type. */
   Utype val; /* Storage for variant type. */
} VariantType;


VariantType array[50];  /* Heterogeneous array. */
array[0].val.ival = 56; /* Assign value. */
array[0].type = INT;    /* Mark type. */
...
```

To be continued. . .