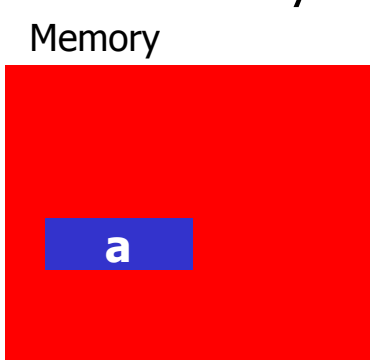
Pointers and Dynamic Memory Allocation

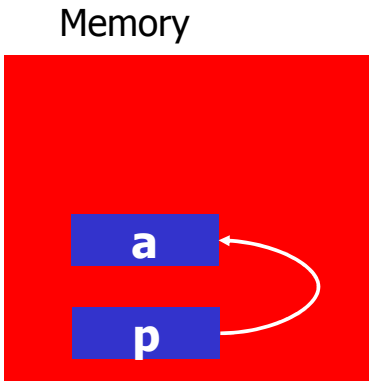
Pointers

Pointers

Pointer-type variables allow accessing memory in an indirect way.




```
int a;
…
a=10;
```




```
int a; int *p;
…
p=&a; *p=10;
```

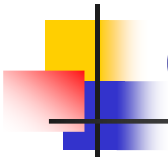

Operators * and &

Operator & gets the pointer to the memory address of a variable :

```
p = &x;
```

Operator * allows accessing the variable referenced by the pointer :

```
*p = 10;
```



Operations on pointers

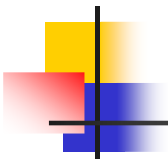
Assignment:

```
p = q;    /* q address is copied in p*/
p = NULL; /* The constant NULL */
```

Increment/decrement

```
p = p+5;
p = p-10;
p++;
```

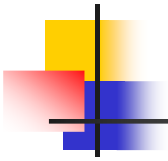
If p points to an int variable, after this statement, p address is incremented of 5*sizeof(int), as p is a pointer to int (int *)



Using pointers

Iterate on a vector to initialize to zero

```
…
int vett[N];
int *p;
…
p=&vett[0];
for (i=0; i<N; i++)
  *p++=0;
…
```



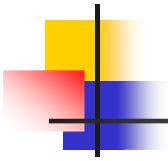
Pointer to struct

When a variable `p` is a pointer to a struct, operator `->` can be used instead:

```
p->field_name
```

is the same as

```
(*p).field_name
```

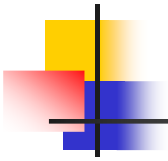



Example

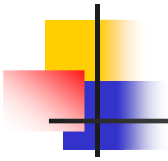
```
struct student{
   int code;
   char name[20];
   char surname[20];
};
struct student *p;
struct student v[N];
…
p=&v[0];
for (int i=0; i<N, i++)
{  p->code=0;
   p++;
}
```

42 bytes needed to store a single student element

Same as `(*p).code=0;`



Pointers and arrays

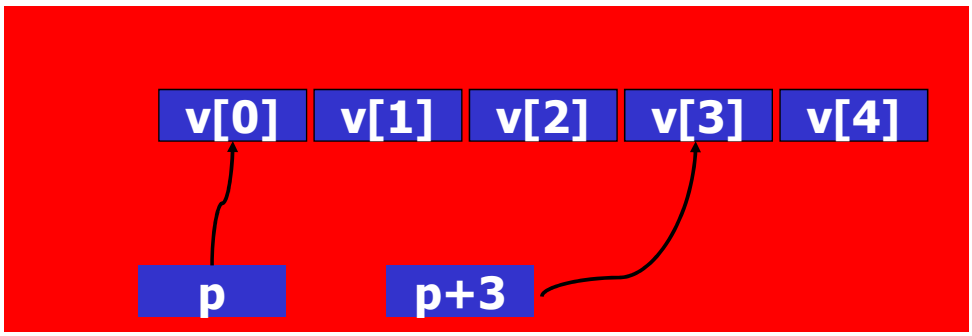


Pointers and Arrays

In C the name of an array variable can be used as a pointer to the first array element.

Arrays are stored in consecutive memory cells.

Pointers and array names can be exchanged.



Example

Definitions:

```
int  v[MAX];
int  *p;
```

Initialization:

```
p = v;        same as        p=&v[0];
```

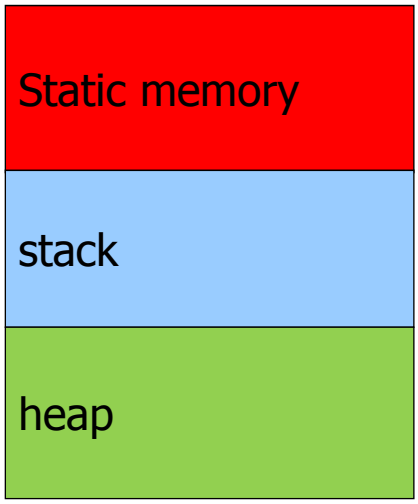
Equivalent forms:

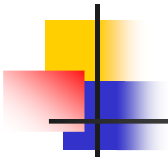
```
v[0]=10;      same as        *p=10;
v[10]=25;     same as        *(p+10)=25;
v[i]=0;       same as        *(p+i)=0;
*v=27;        same as        p[0]=27;
*(v+3)=0;     same as        p[3]=0;
```


Memory allocation

A C program uses memory spaces that are managed differently

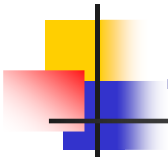
- Static memory
- Dynamic memory (or heap)
- Automatic memory (or stack)





Memory spaces and variables

- Static memory
  - For global variables
- Stack
  - For local variables
  - For parameters passed to / from functions
- Dynamic memory
  - For dynamic variables (not part of language but provided through library of functions)



Size of the memory spaces

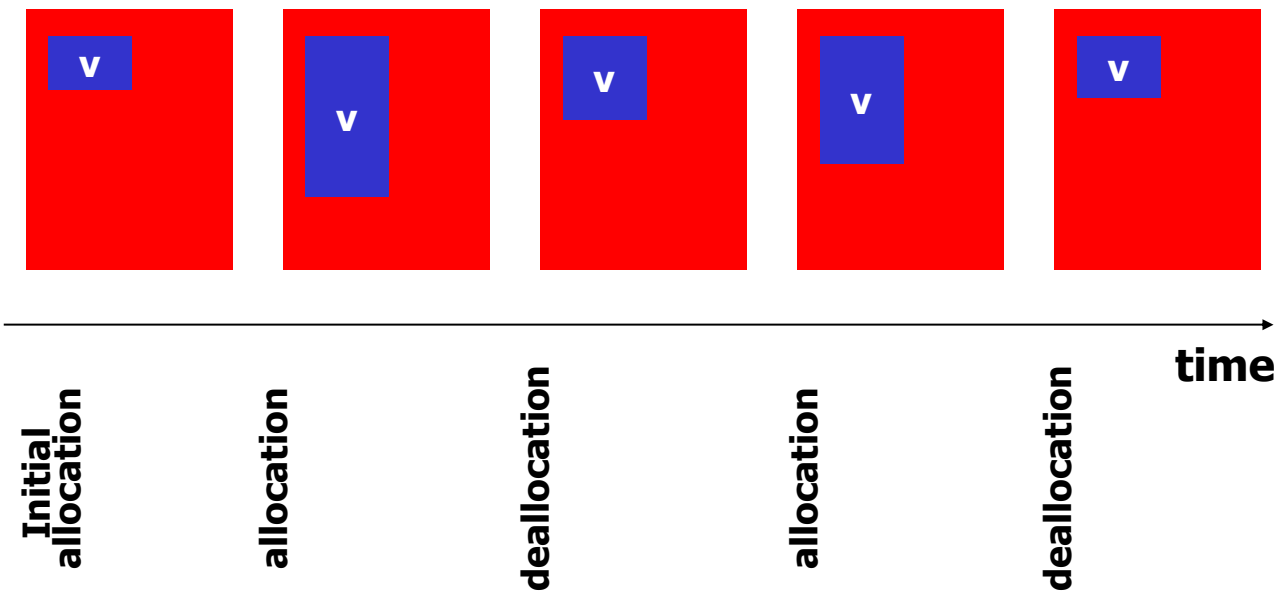
- Static memory has a fixed size, computed by the compiler, and is always used in full
- Heap and stack have a maximum size, are initially empty and then filled and released as needed. It is therefore possible to exceed the space available (stack overflow, memory overflow system errors)

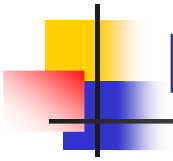
Static memory

```
#define MAX 1000
struct student v[MAX];
```

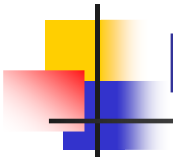
Student dimension is 42
v dimension is 42000 bytes.

Dynamic memory Scenario





Dynamic memory



Dynamic Allocation

Two basic functions

- allocation of a memory area
  - Malloc, calloc, realloc
- release of a memory area
  - free

malloc

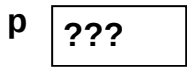
C language provides a system function for dynamic memory allocation

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

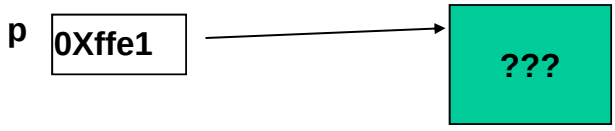
This requires the allocation of a memory area of n bytes and it returns the pointer to the beginning address of the allocated memory area, or NULL if no more memory is available.

Example

- The following declaration allocates memory space for the 'pointer' p
  - `int * p;`

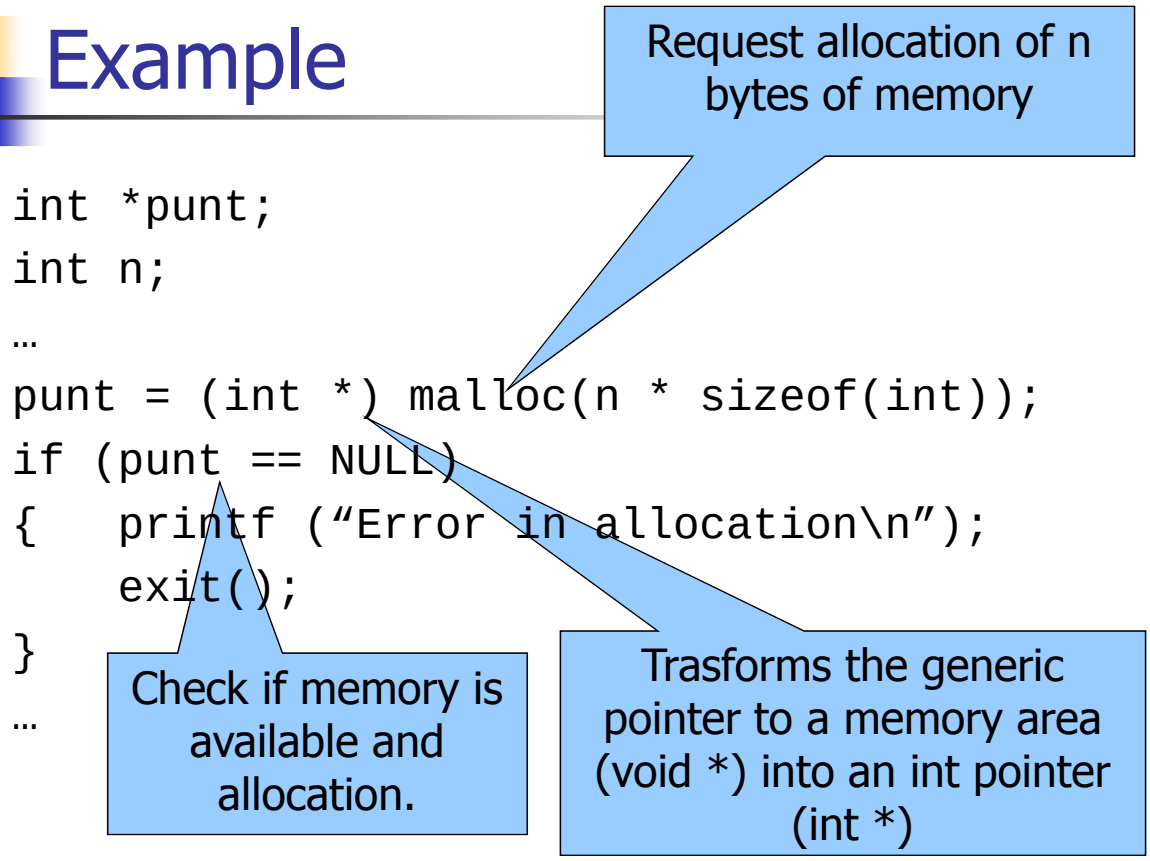


- Allocation of memory is made later
  - `p = (int *) malloc(sizeof(int));`



- Then the variable is initialized
  - *p = 20;

Example



Example

Write the procedure `allocate`

- Reads from keyboard an int `n`
- Allocates an array of `n` elements of type `struct student`
- Initializes each element of the array.

Procedure allocate

```
int n;      /* global variabile */
…
struct student *allocate(void)
{  int i; struct student *p;
   scanf("%d", &n);
   p=(struct student *) malloc(n*sizeof(struct student));
   if (p==NULL)
      return (NULL);
   for (i=0; i<n; i++)
   {  p[i].code=0;
      strcpy(p[i].name, "");
      strcpy(p[i].surname, "");
    }
   return (p);
}
```


Dynamic Allocation of strings

In C strings are stored as char arrays, using '\0' as last character to represent the end of the string.

Two ways to store a string made of n chars:

- Use an array statically allocated of length N>n *or*
- Dynamic allocate an array of n+1 bytes.

Example

Write the function `read` which reads from keyboard date of `n` students, and it stores them in a previously allocated array.

```
struct student{
  int code;
  char *name;
  char *surname;
};
```

Fields `name` and `surname` become pointers.

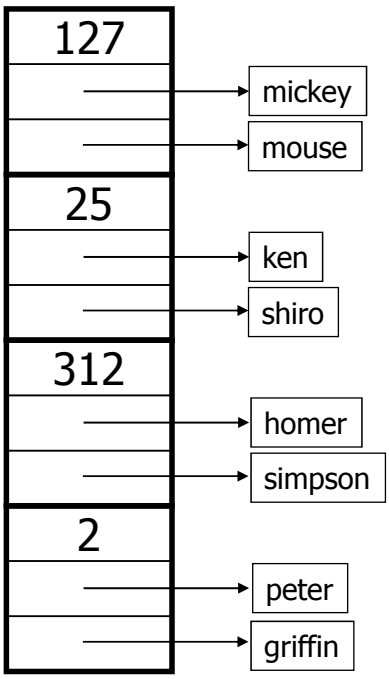
Procedure read

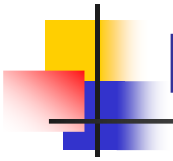
```
int read (struct student *p)
{  int i, val; char name[MAX], surname[MAX];
   for (i=0, i<n; i++)
   {  scanf ("%d %s %s\n", &val, name, surname);
      p[i].code=val;
      p[i].name=strdup(name);
      if (p[i].name == NULL)
         return (-1);
      p[i].surname=strdup(surname);
      if (p[i].surname == NULL)
         return (-1);
   }
   return (0);
}
```

Procedure strdup

```
char *strdup (char *str)
{  int len; char *p;
   len=strlen (str); /* the string str's length */
   p=(char *)malloc((len+1)*sizeof(char));
   if (p==NULL)
      return (NULL);
   strcpy (p, str);
   return (p);
}
```


Data Structure



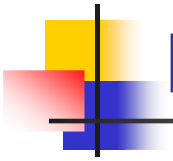


Release memory

The system function **`free`** is used to release a memory area:

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

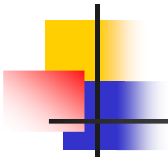
It frees the memory zone pointed by the parameter, which have been allocated with `malloc`.



Example

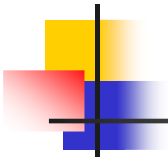
Write the procedure `freedom`, which deallocates the array of n structures passed as parameter.

It is also necessary to deallocate the memory used by strings within each struct element, BEFORE deallocating the array.



Procedure freedom

```
void freedom(struct student *p)
{  int i;
   for (i=0; i<n; i++)
   {  free (p[i].name);
      free (p[i].surname);
   }
   free (p);
}
```

Procedure freedom (vers. 2)

```
void freedom(struct student *p)
{  int i; struct student *q;
   q=p;
   for (i=0; i<n; i++)
   {  free (q->name);
      free (q->surname);
      q++;
   }
   free (p);
}
```

Note Well !

- What happens if we call free(p) immediately ?

```
void freedom(struct student *p) {
   free (p);
}
```

- All strings are no more accessible and cannot be deleted anymore because we deleted their pointers !!!
- Memory is wasted by these strings
- If this bad behavior happens many times, we will eventually run out of memory -> program crashes !!