Dynamic Memory Allocation

Dynamic 1D Arrays

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Local arrays with variable size

- Array dimensions in C **traditionally** had to be compile-time constants
  - It **was impossible** to declare local arrays of a size matching a variable value
  - In other words, it was **impossible** to write code such

```
scanf ("%d", &n);
...
int v[n];
```

A local variable is used to define the size of a local array

```
void f (int n) {
  int v[n];
  ...
}
```

A formal parameter is used to define the size of a local array

Local arrays with variable size

- The C standard ISO/IEC 9899 1999 (C9X) introduced Variable-Length Arrays (VLA's)
  - They allow the previous definitions, where local arrays may have sizes set by variables or other expressions, perhaps involving function parameters
  - In other words, it is now **possible** to write code such

```
scanf ("%d", &n);
...
int v[n];
```

A local variable is used to define the size of a local array

```
void f (int n) {
  int v[n];
  ...
}
```

A formal parameter is used to define the size of a local array

Local arrays with variable size

- However, we **will not** use this sort of constructs for many reasons
  - VLAs are a **subset** of what we can obtain with dynamic memory allocation
  - Run-time allocation is **unsafe**, as the object size is defined at run-time, and there is no proper checking strategy
  - VLAs are **local** objects, and, as such, they cannot be exported
    - In other words, they are automatically deallocated once the environment in which they have been created is abandoned, and they cannot be used outside that environment

Problem definition

- Dynamic memory allocation can be used to allocate arrays of the desired size at run-time
- We focus on 1D and 2D arrays
  - Multi-dimensional generalization are possible and somehow straightforward
- The target is the following
  - How can we define and use and array whose size is known **only** at run-time?
  - We can use the **duality** array ←→ pointers !

Example

Allocate an array to store N integer values

```
int n, *v;

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

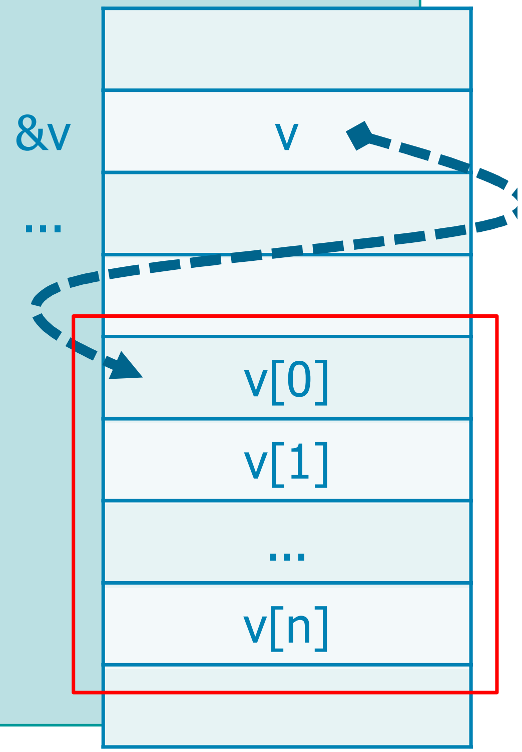
Example

```
int n, *v;

fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);
v = (int *) malloc (n * sizeof (int));
```

At this time, n must be known

As before, fnction **malloc** is normally used with the **sizeof** operator to allocate a proper quantity of memory. All objects stored in the chunk of memory reserved are usually of the same type.



Example

```
int n, *v;

fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);
v = (int *) malloc (n * sizeof (int));
if (v == NULL) {
  fprintf (stderr, "Memory allocation error.\n");
  exit (1);
}
for (i=0; i<n; i++) {
  fprintf (stdout, "v[%d]: ", i);
  scanf ("%d", &v[i]);
}
for (i=n-1; i>=0; i--) {
  fprintf (stdout, "v[%d]=%d\n", i, v[i]);
}
free (v);
```

Please, always check the malloc result

After allocation we can use the array the standard way, using the **Array** or the Pointer notation

(v+i)

*(v+i)

Example

Same solution, using pointer arithmetic

```
int n, *v, *p;

fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);
v = (int *) malloc (n * sizeof (int));
if (v == NULL) {
  fprintf (stderr, "Memory allocation error.\n");
  exit (1);
}
for (i=0, p=v; i<n; i++, p++) {
  fprintf (stdout, "v[%d]: ", i);
  scanf ("%d", p);
}
for (i=0, p--; i>=0; i--, p--) {
  fprintf (stdout, "v[%d]: ", i, *p);
}
free (v);
```

After allocation we can use the array the standard way, using the Array or the **Pointer** notation

Example

Same solution, using calloc

```
int n, *v;

fprintf (stdout, "Introduce n: ");
scanf ("%d", &n);
v = calloc (n, sizeof (int));
if (v == NULL) {
  fprintf (stderr, "Memory allocation error.\n");
  exit (1);
}
for (i=0; i<n; i++) {
  fprintf (stdout, "v[%d]: ", i);
  scanf ("%d", &v[i]);
}
for (i=n-1; i>=0; i--) {
  fprintf (stdout, "v[%d]=%d\n", i, v[i]);
}
free (v);
```

We can also use calloc

If we do not waste time (initializing the array twice)

Observations

- The typical application reads data from file
  - Example
    - A file include a list of integers
    - Read the list and save it in another file in reverse order
    - Input file : 2 4 6 8 10 12
    - Output file: 12 10 8 6 4 2

Observations

- The size of the file must be indicated on top of the file to properly allocate the array
  - If is not
    - Avoid starting with a malloc of size “1“ and then reallocate of size “+1“ when reading a new value
      - This is tremendously inefficient
  - Either
    - Allocate of size N and double the size when it is no more sufficient OR
    - Read the file a first time to count-up the number of values inside OR
    - Do not use dynamic arrays but some other data structures (e.g., lists)

Example

```
#define N 1000

int *v1, *v2;

v1 = malloc (N * sizeof (int));
if (v1 == NULL) { ... }
...
v2 = realloc (v1, 2 * N * sizeof (int));
if (v2 == NULL) {
  fprintf (stderr, "Memory allocation error.\n");
  free (v1);
  exit (1);
}
...
free (v2);
```

Allocation strategy:Double the number of elements at each new allocation

Common errors

```
char v[10];
char *p = malloc (10 * sizeof (char));
```

- sizeof (v)
  - The size of the array (in bytes), i.e., a set of 10 characters each one of 1 byte, that is, 10
- sizeof (p)
  - The size of the pointer p, i.e., 4 or 8 bytes on modern hardware architectures (at 32 or 64 bits)

Modularity

- One of the main problem with dynamic memory allocation is how to export objects
  - How can we make dynamically allocated variable visible from outside the environment in which they have been allocated?

Example

Allocation function

```
void array_create (int *ptr, int n) {
  ptr = (int *) malloc (n * sizeof (int));
  if (ptr == NULL)  { ... }
  return;
}
```

Here I want to allocate the array (and maybe read it from stdin)

Caller (user or client)

Here I want to use it

```
int n, *v=NULL;

scanf ("%d", &n);
array_create (v, n);
```

Unfortunately, v is **NULL** here

Modular Allocation

- To rectify this problem there are at least three possible solutions
  - Define variables, i.e., pointers, as **global** objects
    - Global variables must be avoided as long as possible
    - We will discuss this option (advantages and disadvantages) in the modulary session
  - **Return** variables, i.e., pointers, from the function using the return statement
  - Pass the variables, i.e., pointers, to the function as a **parameter by reference**
  - ..

Modular Allocation

- To rectify this problem there are at least three possible solutions
  - Define variables, i.e., pointers, as **global** objects
    - This is the simplest solution, but …
    - Global variables must be avoided as long as possible
      - We will discuss advantages and disadvantages in the modulary session
      - We will avoid this approach as long as possible

Modular Allocation

- Use the **return statement** to return the variables, i.e., pointers, from the function
  - This is simple enough, but …
  - Unfortunately in C only one value can be returned
    - Even if we can return a C structure including more pointers this can be seen as an awkward solution to solve easy cases

Example

Allocation function

```
int *array_create (int n) {
  int *ptr;
  ptr = (int *) malloc (n * sizeof (int));
  if (ptr == NULL)  { ... }
  return ptr;
}
```

Here I want to allocate the array (and maybe read it from stdin)

Caller
(user or client)

Here I want to use it

```
int n, *v=NULL;

scanf ("%d", &n);
v = array_create (n);
```

V is generally not **NULL** here

Modular Allocation

- Pass the variables, i.e., pointers, to the function as a **parameter by reference**
  - This is the most complex solution, but …
  - It is also the most general one

Example

Allocation function

```
void array_create (int **ptr, int n) {
  *ptr = (int *) malloc (n * sizeof (int));
  if (*ptr == NULL)  { ... }
  return;
}
```

Here I want to allocate the array (and maybe read it from stdin)

Here I want to use it

Caller
(user or client)

```
int n, *v=NULL;

scanf ("%d", &n);
v = array_create (&v, n);
```

V is generally not **NULL** here

Example

Allocation function

```
void array_create (int **ptr, int n) {
  int *lptr;
  lptr = (int *) malloc (n * sizeof (int));
  if (lptr == NULL)  { ... }
  *ptr = lptr;
  return;
}
```

Here I want to allocate the array (and maybe read it from stdin)

Here I want to use it

Caller
(user or client)

```
int n, *v=NULL;

scanf ("%d", &n);
v = array_create (&v, n);
```

V is generally not **NULL** here

String allocation

- Dynamic strings can be allocated as other dynamic arrays
- However, it is necessary to remind that a string has a termination character '\0'
  - Therefore, it is necessary to **always** reserve space for that character
- Algernatively, we can use the **strdup** function

Example

```
char str[100+1];
char *v;

scanf ("%s", str);
v = malloc ((strlen (str) + 1) * sizeof (char));
if (v == NULL) { ... }
strcpy (v, str);
...
free (v);
```

This +1 may worth several hours of useless debugging effort

Notice that **str** may/must have more elements than required, **v** has the tightest possible size

```
char str[100+1];
char *v;

scanf ("%s", str);
v = strdup (str);
...
free (v);
```

With **strdup**

General array allocation

- The previous code snippets can be generalized to any array
  - Arrays of structures including
    - Static fields
    - Dynamic fields
    - Etc.

Example

```
struct student {
  char last_name[DIM], first_name[DIM];
  int register_number;
  float average;
};
...
int n;
struct student *v;
...
v = (struct student *)
    malloc (n * sizeof (struct student));
if (v == NULL) { ... }
...
free (v);
```

We can allocate dynamic arrays with static arrays inside

Example

```
struct student {
  char *last_name, *first_name;
  int register_number;
  float average;
};
...
int n;
struct student *v;
...
v = (struct student *)
    malloc (n * sizeof (struct student));
if (v == NULL) { ... }
...
```

We can allocate dynamic arrays with dynamic arrays inside

But these dynamic array **must** be allocated … We need to allocate the last_name and first_name fields of all elements of v … This is the topic of the **next section**