Pointers

EECS 2031

Song Wang
wangsong@eecs.yorku.ca
eecs.yorku.ca/~wangsong/

Acknowledgement

- Some of the covered materials are based on previous EECS2031 offerings:
  - Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

Structures

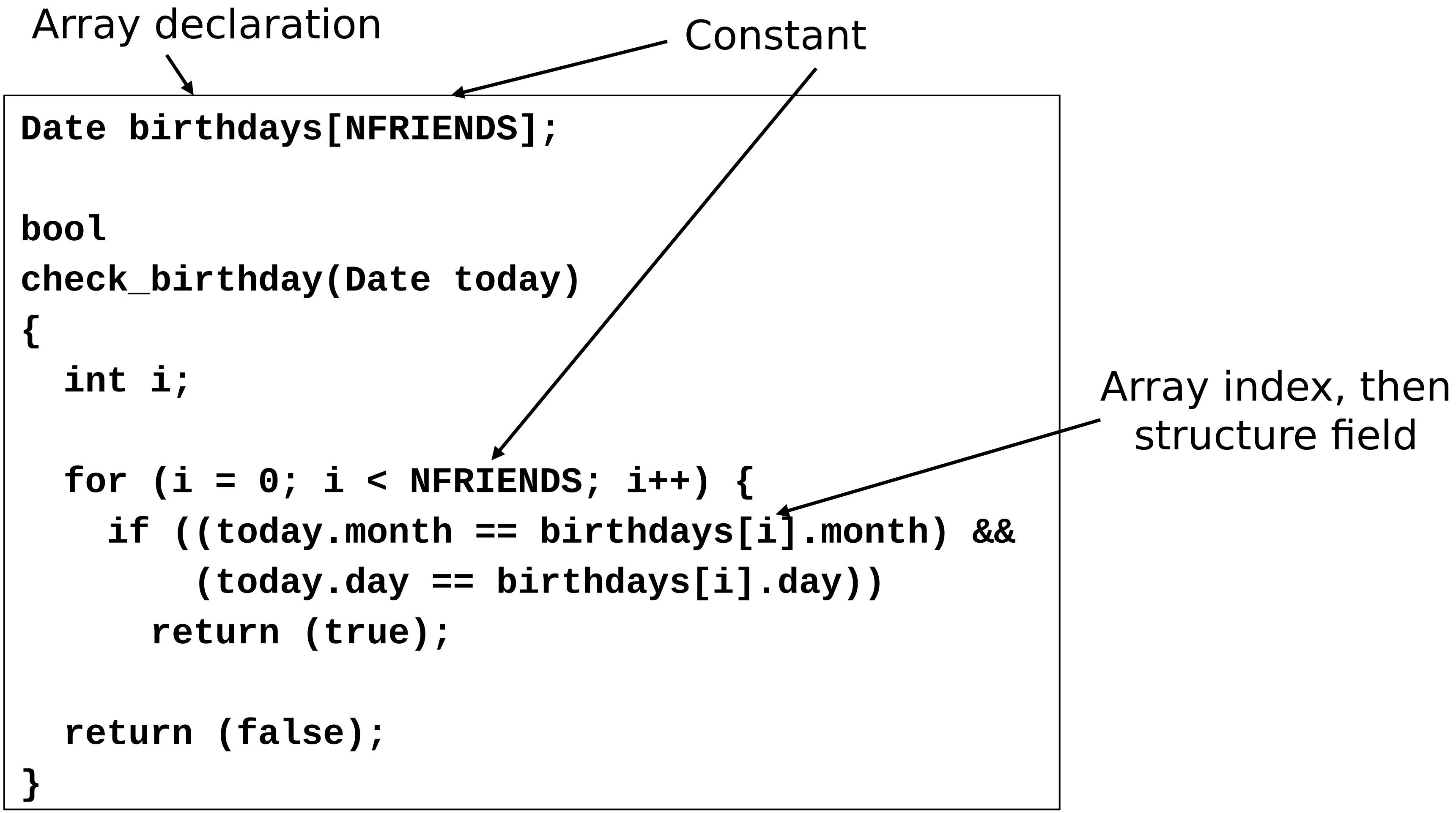
- Compound data:

- A date is
    - an `int month` <u>and</u>
    - an `int day` <u>and</u>
    - an `int year`

```
struct Date {
   int  month;
   int  day;
   int  year;
};

struct Date date;

date.month = 2;
date.day = 4;
date.year = 2021;
```

- Unlike Java, C doesn’t automatically define functions for initializing and printing …

Arrays of Structures



Array declaration

Constant

```
Date birthdays[NFRIENDS];

bool
check_birthday(Date today)
{
  int i;

  for (i = 0; i < NFRIENDS; i++) {
    if ((today.month == birthdays[i].month) &&
        (today.day == birthdays[i].day))
      return (true);

  return (false);
}
```

Array index, then
structure field

- You can pass structures as arguments to functions

```
main()  {
  struct shape  s = {1,2};
  do_sth(s)           /* s.width?  s.height?  */
}
```

```
void do_sth(struct shape d)
{
    d.width  += 100;
   d.height += 200;
}
```

call-by-value

```
d = s  // copy members
    d.width = s.width
    d.height = s.height
```

- This is call-by-value - a copy of the struct is made
  - Function cannot change the passed struct

- structs can be used as return values for functions as well

```
struct shape make_dim(int width, int height)
{
  struct shape d;   // in stack
  d.width = width;
  d.height = height;
  return d;
}
main(){
  struct shape myShape = make_dim(3,4);
}
```

```
// myShape = d;
Copy members, d is gone (deallocated) afterwards
```

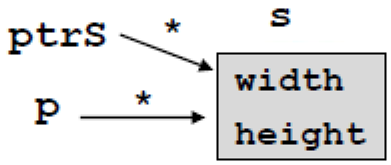
Structure and Functions --Structure Pointers

-- Structure Pointers

- call-by-value is inefficient for large structures: not decayed
  - use pointers (explicitly) !!!
- This also allows to change the passing struct

```
main(){
   struct shape s = {1,3};
   struct shape * ptrS = &s; // pointer to struct shape
   float f = get_area(ptrS); // float f = get_area(&s);
}
float get_area(struct shape *p)
{  struct shape tmp = *p;
   return tmp.width   *  tmp.height;
}
```

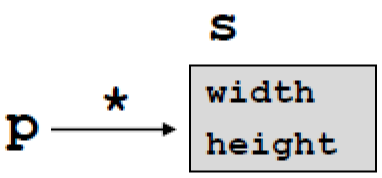
Expect a pointer to struct shape



Structure and Functions --Structure Pointers

- call-by-value is inefficient for large structures: not decayed
    - use pointers!!!
- This also allows to change the passing struct

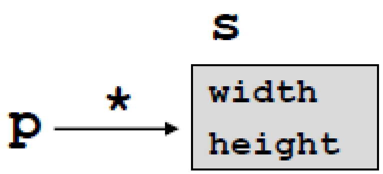
```
main(){
   struct shape  s = {1,2};
   do_sth(&s);
}
void do_sth(struct shape * p)
{
    (*p).width  += 100;
    (*p).height += 200;
}
```



Pointee s is modified !

Structure and Functions --Structure Pointers

```
void do_sth(struct shape *p){
    (*p).width += 100;
}
```



- Beware when accessing members a structure via its pointer

  `*p.width` ✗

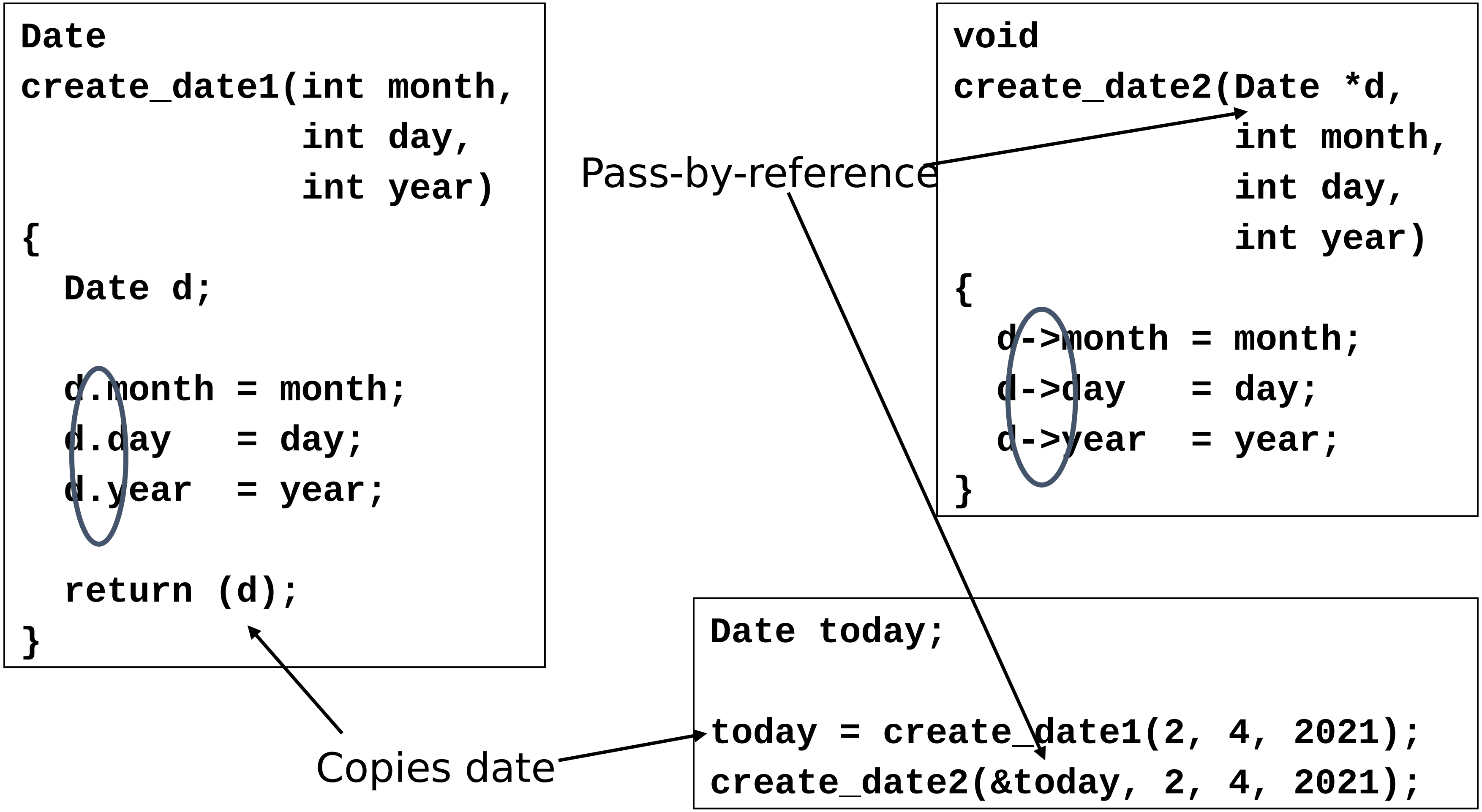
- Operator . takes higher precedence over operator *

  `(*p).width` ✓

- **Accessing member of a structure via its pointer** is so common that it has its own operator

  `p -> width`

Pointers to Structures



```
Date
create_date1(int month,
             int day,
             int year)
{
  Date d;

  d.month = month;
  d.day   = day;
  d.year  = year;

  return (d);
}
```

```
void
create_date2(Date *d,
             int month,
             int day,
             int year)
{
  d->month = month;
  d->day   = day;
  d->year  = year;
}
```

Pass-by-reference

Copies date

```
Date today;

today = create_date1(2, 4, 2021);
create_date2(&today, 2, 4, 2021);
```

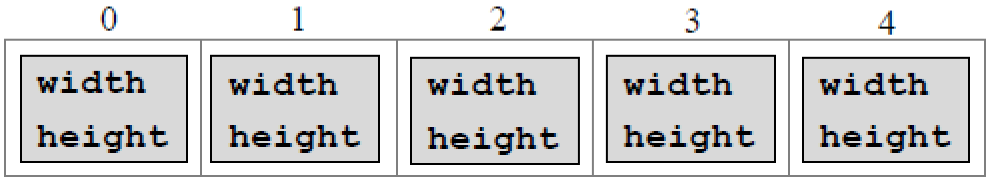
Arrays of structures --declaration

- Structures can be arrayed same as the other variables

```
struct shape {
   float width;
   float height;
 };
```

```
struct shape chairs[5];   // recall: int arr[5]
```

array of 5 (uninitialized) struct



0	1	2	3	4
width height	width height	width height	width height	width height

Not NULL

Dynamic memory allocation motivation

- When we define an array, we allocate memory for it

```
int arr[20];
```

sets aside space for 20 ints (80 bytes)

- This space is allocated at compile-time (i.e. when the program is compiled)

```
#define SIZE 20

int arr[SIZE];          20*4 bytes
char arr[20][30];       20*30*1 bytes
int arr[] = {3,5,6}; 3*4 bytes
char arr[] = "Hello" 6*1 bytes
```

Dynamic memory allocation motivation

- What if we do not know how large our array should be?
- length is determined at runtime rather than compile time
- In other words, we need to be able to allocate memory at run-time (i.e. while the program is running)

- How?

```
int n;
printf("How many elements in int array? ");
scanf("%d", &n);
int my_array[n];   /* but not allowed in ANSI-C */
```

```
gcc -ansi -pedantic varArray.c
gcc -ansi -pedantic-errors varArray.c
   ISO C90 forbids variable length array 'my_array'
```

- Fortunately, C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution.

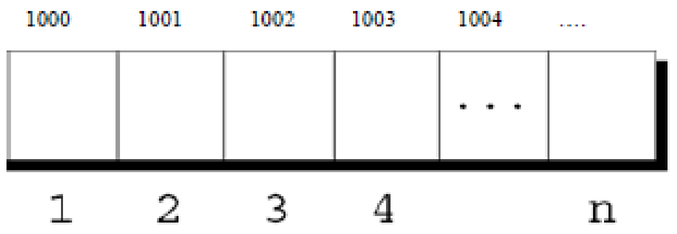
- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

- The `<stdlib.h>` header declares three memory allocation functions:

`malloc`	Allocates a block of memory but doesn’t initialize it.
`calloc`	Allocates a block of memory and clears it.
`realloc`	Resizes a previously allocated block of memory.

- These functions return a value of type `void *` (a “generic” pointer).
  - function has no idea what type of data to store in the block.

malloc()



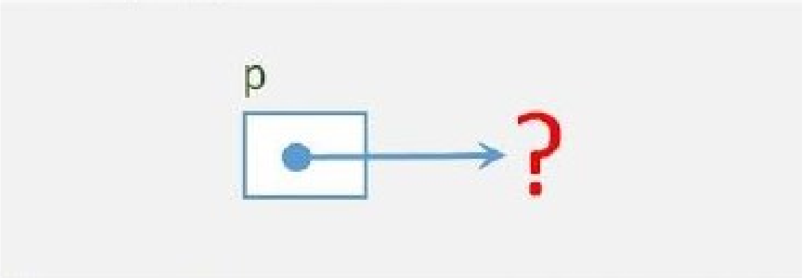
- `"stdlib.h"` defines:

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

- allocates memory at run-time
- returns a void pointer to the memory that has at least n bytes available (just allocated for you).
  - Address of first byte e.g., 1000
  - Can be <u>casted</u> to any type

malloc()

Dangling Pointers

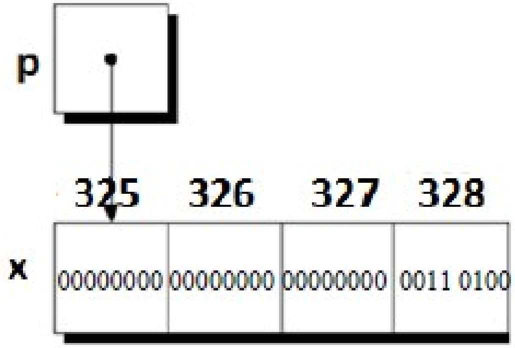


```
#include <stdlib.h>

int main() {
  int *p;  // uninitialized, not point to anywhere

  *p = 52;
  printf("%d\n", *p);
}
```

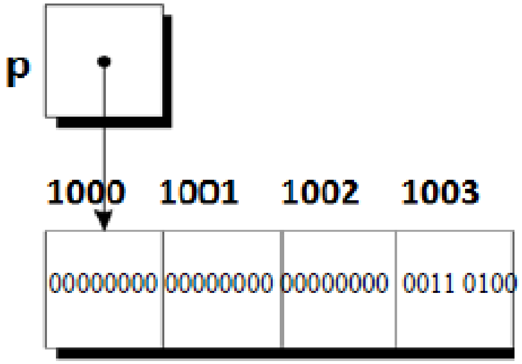
```
segmentation fault
core dump
```

malloc()

```
#include <stdlib.h>

int main() {
  int *p, x;
  p = &x;
  *p = 52;  // x=52
  printf("%d\n", *p);
}
```



malloc()

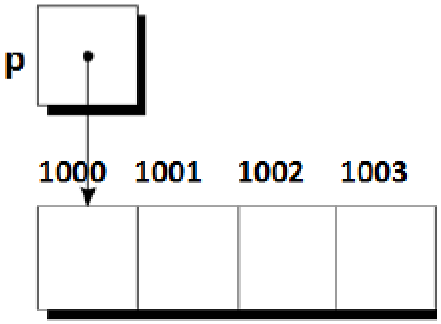


```
#include <stdlib.h>

int main() {
  int *p;
  p = (int *) malloc(4);
  *p = 52;
  printf("%d\n", *p);
}
```

- Note: type conversion (cast) on result of malloc

  `p = malloc(4);` also works. Will convert

malloc()



- A better approach to ensure portability

```
int *p;
p = (int *) malloc(4);
             ⬇
p = (int *) malloc( sizeof(int) );
*p = 52;
```

malloc()

- Allocation not always successful

- malloc() returns **`NULL`** when it cannot fulfill the request, i.e., memory allocation fails (e.g. no enough space)

```
int *p;
p = (int *)malloc(100000000);// malloc returns NULL
p = (int *)malloc(-10);        // malloc returns NULL
```

NULL

- <stdlib.h> <stdio.h> <string.h> …defines macro
  NULL a special pointer constant with value 0
- 0 (zero) is never a valid address

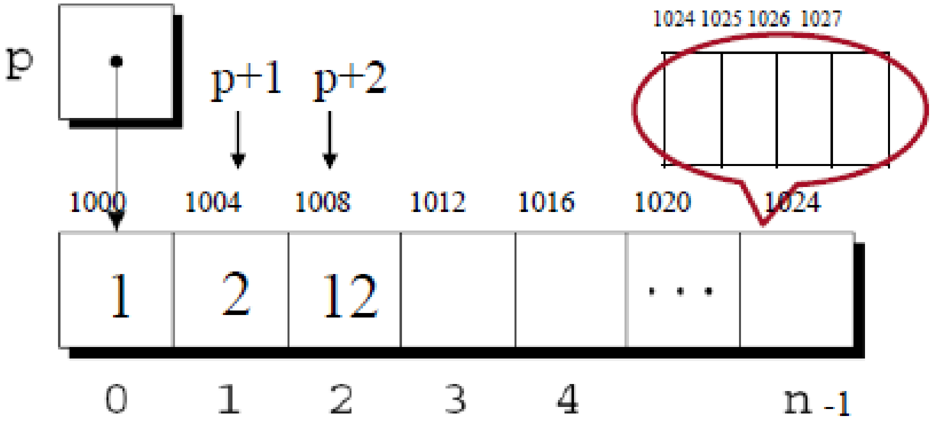
- **NULL** == "0 as a pointer" == "points to nothing"
  - `int * p;   // p == NULL?  Not really`
  - `p == 0 ?   // better use NULL    like EOF`

```
p = malloc(10000000);
if (p == NULL) {  // an "exception"
   exit(0) /* allocation failed; take appropriate action
}
else …
```

⬇

```
if ( (p = malloc(10000000)) == NULL) {
   exit(0) /* allocation failed; take appropriate action
}else ….
```

malloc()



4n bytes allocated.
n=7   28 bytes   1000~1027 allocated

```
#include <stdlib.h>

int main() {
   int n;
   printf("How many elements in int array? ");
   scanf("%d", &n);

   int * p = (int *)malloc(n * sizeof(int));
   if (p == NULL)
      exit(0);

   // else
   *p = 1;        // p[0] = 1     second +1 +4?
   *(p+1) = 2;    // p+1 = 1004   p[1]= 2
   *(p+2) = 12;   // p+2 = 1008   p[2] = 12
                     pointer arithmetic!!!
 }
```

calloc()

- What if we want to allocate arrays of **`n`** element?

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

  alternatively,

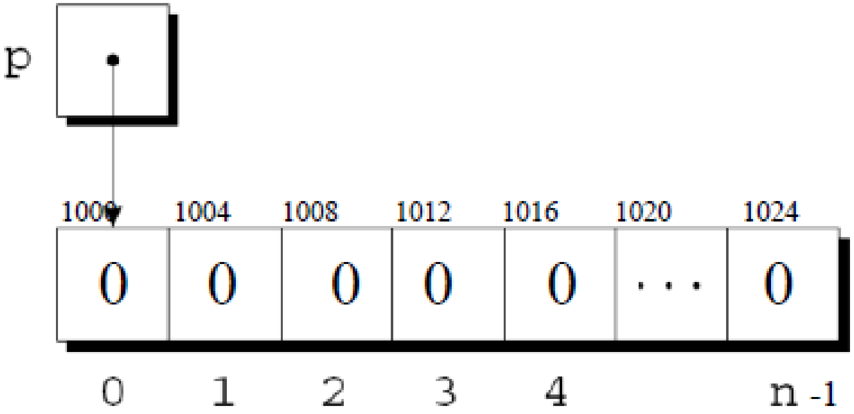
  ```
  void * calloc(int n, int size);
  ```

- **`calloc`** allocates an array of **`n`** elements where each element has size **`size`**
- e.g.

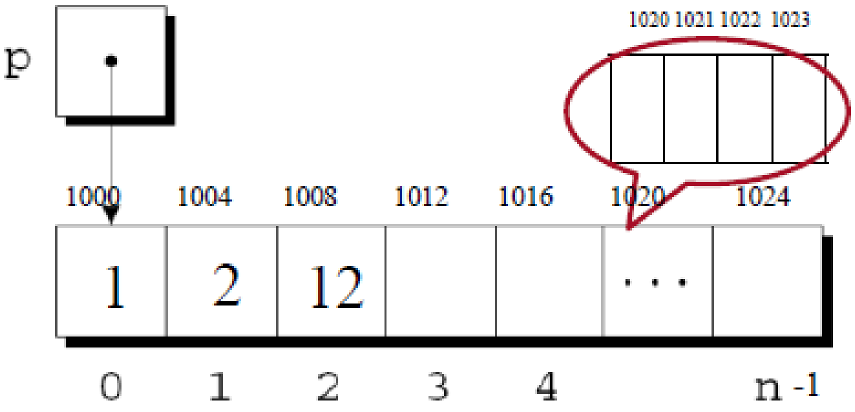
  ```
  int *p;
  p = (int *)calloc(6, sizeof(int));
  ```

calloc() vs. malloc()

- `calloc(x , y)` is pretty much the same as `malloc(x * y)`
- except
  - `malloc` does not initialize memory
  - `calloc` initializes memory content to 0 (zero)



calloc() malloc()



4n bytes allocated.
n=7 28 bytes 1000~1027 allocat

```
#include <stdlib.h>

int main() {
  int n;
  printf("How many elements in int array? ");
  scanf("%d", &n);

//int * p = (int *)malloc(n * sizeof(int));
  int * p = (int *)calloc(n , sizeof(int));
  if (p == NULL) exit(0);

  *p = 1;          // p[0] = 1
  *(p+1) = 2;      // p+1 = 1004  p[1]= 2
  *(p+2) = 12;     // P+2 = 1008  p[2] = 12;
```

free()

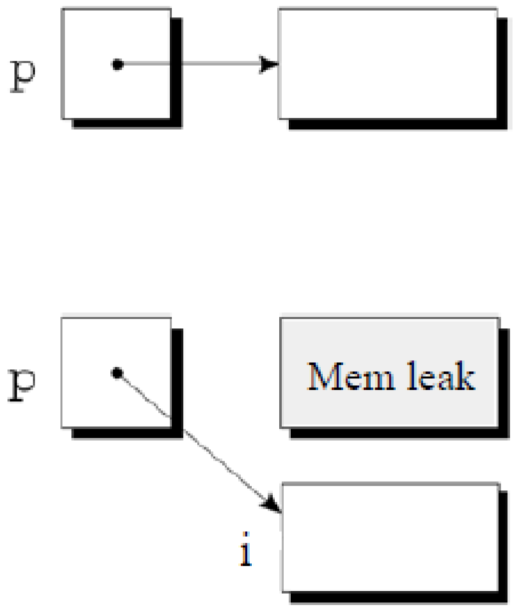
- memory allocation functions `malloc, calloc` obtain memory blocks from a storage pool known as the ***heap,*** where storage is persistent until the programmer explicitly requests that it be deallocated (or program terminates)

- A block of memory that's no longer accessible to a program is said to be ***garbage.***
  - A program that leaves garbage behind has a ***memory leak.***

- Some languages (e.g., Java) provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.

Memory Leaks

```
int *p;
p = (int *) malloc( 20 );
...
p = &i; //now point to sth else
```



- The first memory block is lost “forever” (until program terminates).
- May cause problems (exhaust memory).

Memory Leaks

A program that forgets to deallocate a block is said to have a "memory leak" which may or may not be a serious problem. **The result will be that the heap gradually fills up as there continue to be allocation requests**, but no deallocation requests to return blocks for re-use.

For a program that runs, computes something, and exits immediately, memory leaks are not usually a concern. Such a “one shot” program could omit all of its deallocation requests and still mostly work.

Memory leaks are more of a problem for a program that runs for an indeterminate amount of time. In that case, the memory leaks can gradually fill the heap until allocation requests cannot be satisfied, and the program stops working or crashes.

free()

- Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

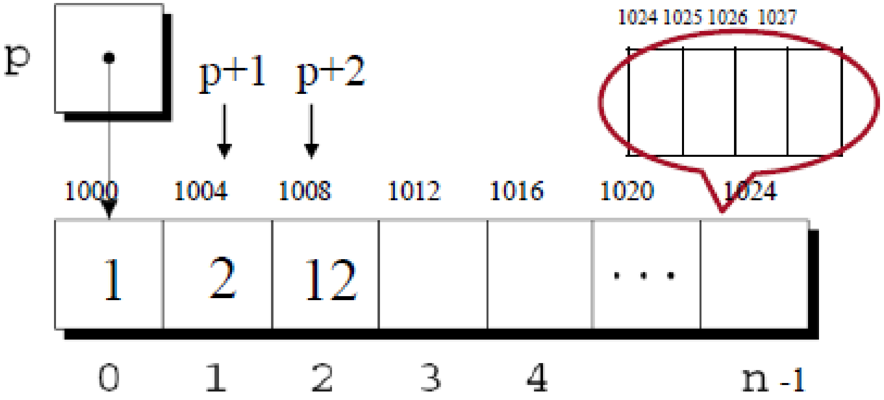
```
void free (void *ptr);
```

- "frees" memory we previously allocated, tells the system we no longer need this memory and that it can be reused

- address in "`ptr`" must have been returned from either `malloc`, `calloc` or `realloc`.

```
p = malloc(7*4);
...
free(p);
```

malloc() calloc()



4n bytes allocated.

n=7 28 bytes 1000~1027 allocated

```
#include <stdlib.h>

int main() {
  int n;  int *p;
  printf("How many elements in int array? ");
  scanf("%d", &n);

  p = (int *)malloc(n * sizeof(int)); //or
  p = (int *)calloc(n , sizeof(int));
  if (p == NULL)
     exit(0);

  *p = 1;          // store 1 at address 1000 (1000~1003)
  *(p+1) = 2;      // p+1 = 1004  store 2 at address 1004
  *(p+2) = 12;     // p+2 = 1008  store 12 at address 1008
```

pointer arithmetic!!!

```
  free (p);
  p=&i;
```

realloc()

```
char *ptr;
ptr = malloc(20);
…
ptr = realloc(ptr,50)
```

- resize a dynamically allocated array.

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

- `ptr` must point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`.
  - `ptr` is NULL, a new block is allocated

- `size` represents the new size of the block, which may be larger or smaller than the original size.

- `realloc(NULL, n)` behaves like `malloc(n)`.
- `realloc (ptr, 0)` behaves like `free(prt)`, as it frees the memory block.