Dynamic memory allocation

Introduction

- Fixed-size data structures
- Have to declare the size of arrays, and may end up going for too much
- Contradicts the savings of bytes we have been talking about
- Solution: Get only as much memory as needed – no more and no less – at run time

Dynamic memory allocation

- Enables a program to obtain more memory space at execution time and to release memory when it is no longer needed
- Limit for dynamic memory allocation can be as large as the amount of virtual memory available on the system
- The function `malloc` allocates storage for a variable and returns a pointer to it
- ANSI version is prototyped as

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

  - `void *` is used to return a generic pointer

- Assume the declarations

```
char *p;
int n = ...;
```

- The assignment statement

```
p = malloc(n);
```

  requests a block of memory space consisting of `n` bytes

  - If `n` consecutive bytes are available, the request is granted
  - The address of the first byte is assigned to `p`
  - If the call is unsuccessful, `p` is assigned `NULL`
  - It is a good idea to follow the `malloc()` by

```
if ( p == NULL ) ...
```

  - The allocated memory block can be treated the same as if it had been declared by using the statement

```
char p[n]
```

    if that were possible (`n` is not a constant)

- Any integer expression can be used as an argument to `malloc` instead of only a constant-expression in array declaration
- `malloc` gives us more freedom with respect to when and where we reserve the memory
- Recommended use of `malloc`

```
person_info_t * person;
person = ( person_info_t * ) malloc ( sizeof ( person_info_t ) );
```

- Memory is deallocated by using the `free` function
  - Memory is returned to the system to be reallocated in future
  - The memory allocated to the variable `person` above can be freed by using the statement

    ```
    free ( person );
    ```

  - Since the pointer is passed using call-by-value, it is not set to `NULL`
  - `free ( NULL );` is valid and results in no action
- Caution
  1. *If* `malloc` *is used to allocate anything other than a character string, it should be typecast to avoid the type-conflict errors*
  2. *A structure's size is not necessarily the sum of the sizes of its members; this is so because of various machine-dependent boundary alignment requirements;* `sizeof` *operator resolves this problem*
  3. *Not returning dynamically allocated memory when it is no longer needed can cause the system to run out of memory prematurely – a phenomenon known as "memory leak"*
  4. *Do not attempt to* `free` *memory not allocated through* `malloc`
  5. *Do not attempt to refer to memory that has been* `free`*d*
- The function `realloc(ptr, size)`
  - Used to increase or decrease the allocated space
  - Changes the size of the block referenced by `ptr` to `size` bytes and returns a pointer to the [possibly moved] block
  - The values returned by the function is `null` if the memory allocation fails
  - If successful, the function returns a pointer to the first byte
  - The contents of the block are left unchanged up to the lesser of the new and old sizes; contents may be copied if the block needs to be moved
  - The function `free(p);` can also be written as `realloc ( p, 0 );`