Dynamic Memory Allocation

ITCS 2116: C Programming

College of Computing and Informatics

Department of Computer Science

Why Dynamic Memory Allocation?

- Don't know how much data will need to be stored until runtime; choices?

  Choice 1: Declare **static array** of maximum size that could possibly occur

```
#define MAXCLASSSIZE 500
struct student { …definition here… };
struct student students[MAXCLASSSIZE];

int i = 0;
while (more_students  && (i < MAXCLASSSIZE))
   readstudents (students[i++]);
```

Why Dynamic … (cont'd)

Choice 2: Declare **dynamic (auto) array** of specific size needed, at run time

```
int main (void) {
   int maxnum;
   printf("Number of students in class? \n");
   scanf("%d", &maxnum);
   struct student students[maxnum];

   int i = 0;
   while (more_students  && (i < maxnum))
      readstudents (students[i++]);
}
```

Why Dynamic… (cont'd)

Choice 3: Allocate memory **dynamically** using a standard library function (**`malloc`** or **`calloc`**)

```
#include <stdio.h>
#include <stdlib.h>
…
int main(void) {
    struct student *sp;
    while (more_students) {
        sp = (struct student *)
             calloc (num, sizeof(struct student));
        if (sp != NULL)
            readstudents (sp);
    }
}
```

Dynamic Storage Allocation

- Dynamic storage allocation is used most often for strings, arrays, and structures.
- Dynamically allocated structures can be linked together to form lists, trees, and other data structures.
- Dynamic storage allocation is done by calling a memory allocation function.

Memory Allocation Functions

- 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).

Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer.***
- A null pointer is a special value that can be distinguished from all valid pointers.
- After we have stored the function's return value in a pointer variable, we must test to see if it is a null pointer.

Null Pointers

- An example of testing **malloc**'s return value:

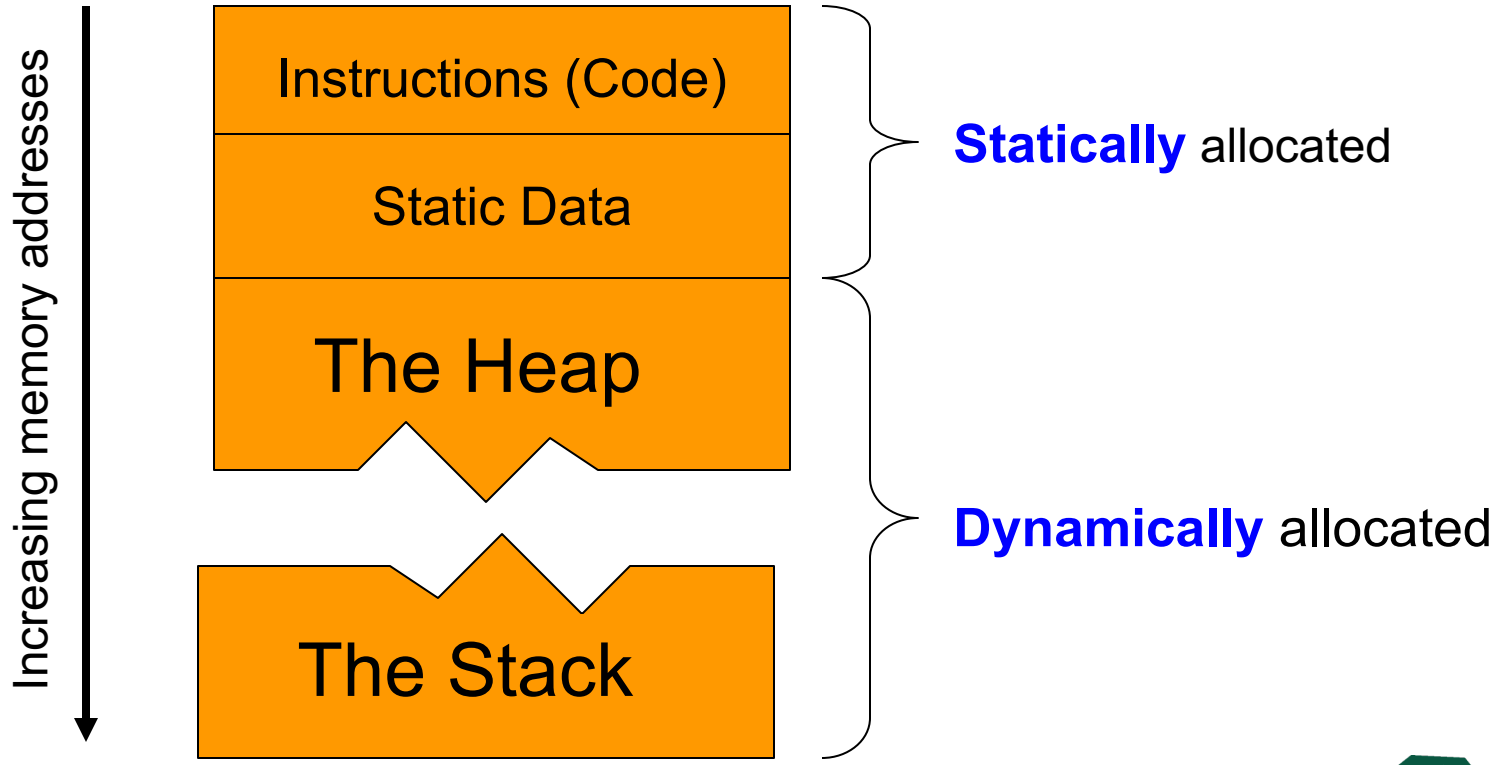
```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed; take appropriate action */
}
```

- **NULL** is a macro (defined in various library headers) that represents the null pointer.
- Some programmers combine the call of **malloc** with the **NULL** test:

```
if ((p = malloc(10000)) == NULL) {
  /* allocation failed; take appropriate action */
}
```

Memory Layout of a Program

- The **heap** is an area of *virtual memory* available for dynamic (runtime) memory allocation



C vs. Other Languages

- **C** requires you to **manually allocate and reclaim** memory.
- Other languages (e.g. Java, C#) automatically allocate and reclaim memory for you.

The `sizeof` Operator

- Not a function call; a **C operator**
  - returns **number of bytes** required by a data type
- Return value is of predefined type **`size_t`**

(see **`sizeof_example.c`** in *Code samples and Demonstrations* in *Canvas*)

```
#include <stdlib.h>
size_t tsz1, tsz2, tsz3;
int a;
float b[100];
struct student { …definition here… } st;

tsz1 = sizeof (a);  /* 4 */
tsz2 = sizeof (b);  /* ? */
tsz3 = sizeof (st); /* ? */
```

what are these sizes?

The `calloc()` Standard Library Function

Syntax: `void * calloc (size_t num, size_t sz)`

Generic pointer, must be cast to type of result

OS allocates (`num * sz`) **bytes** of contiguous storage (all bytes **initialized** to zeros)

```
struct student * students;
students = (struct student *)
      calloc (num, sizeof(struct student));
int * ip;
ip = (int *) calloc (1, sizeof (int));
char *cp;
cp = (char *) calloc (1000, sizeof (char));
```

`calloc()` (cont'd)

- Return value is starting address of the storage allocated
- If not enough memory available, returns **`NULL`**
  - Could also be a unique pointer that could be passed to free()
  - **ALWAYS check for this error**

☠ *common source of bugs* ☠
failure to check return value

```
cp = (char *) calloc (1000, sizeof (char));
if (cp == NULL) {
   printf("Cannot allocate memory; exiting\n");
   exit (-1);
}
```

The `malloc()` Std. Lib. Function

- Syntax: `void * malloc (size_t sz)`
- OS allocates `sz` bytes of contiguous storage
  - **Uninitialized**
- Returns starting address of storage
  - If size is 0, returns NULL or unique pointer that can be freed

☠ *common source of bugs* ☠
malloc() does not initialize memory

```
students = (struct student *)
       malloc ( num * sizeof(struct student));
ip = (int *) malloc (sizeof (int));
cp = (char *) malloc ( 1000 * sizeof (char));
```

(see `examples.c` in *Code samples and Demonstrations* in *Canvas*)

The `realloc()` Std. Lib. Function

- Syntax: `void * realloc(void * ptr, size_t sz)`
- Grows or shrinks allocated memory
  - `ptr` must be dynamically allocated
  - Growing memory doesn't initialize new bytes
  - If can't expand, returns `NULL`. Old memory is unchanged
  - If `ptr` is `NULL`, behaves like malloc
  - If `sz` is `NULL`, behaves like free
  - Memory shrinks in place
  - **Memory may NOT grow in place**
    - If not enough space, will move to new location and copy contents
    - Old memory is freed
    - Update all pointers!!!

The `free()` Standard Library Function

- Syntax: `void free (void * ptr)`
  - no way to check for errors!
  - `ptr` **must** have been previously allocated by `malloc()` or `calloc()`
  - no need to specify **amount** of memory to be freed.
  - Frees (for other uses) memory previously allocated

```
free(students);
free (ip);
free (cp);
```

☠ *common source of bugs* ☠
failure to free unused memory

Dynamic Memory Allocation

Common Mistakes

- These bugs can **really** be hard to find and fix
  - May run for hours before the bug pops up, and in a place that appears to have no relationship to the actual cause of the error

Mistake **M1**: Invalid Pointers

☠ *common source of bugs* ☠

- Problems?

```
int i, j, result;
result = scanf ("%d %d", i, &j);
```

```
char *ptr;
…
ptr = 'A';
…
*ptr = 'B';
```

(see **`invalid1.c`** and **`invalid2.c`** in *Code samples and Demonstrations* in *Canvas*)

Invalid Pointers (cont'd)

☠ *common source of bugs* ☠

- Problems?

```
int * f( void )
{
    int val;
    …
    return &val;
}
```

why is this a problem?

(see **`invalid3.c`** in *Code samples and Demonstrations* in *Canvas*)

Invalid Pointers (cont'd)

- Problems? Fix?

☠ *common source of bugs* ☠

```
…dynamically allocate and construct a linked list…
…
/* now list is no longer needed,
 * free memory
 */
for (p = head; p != NULL; p = p->next)
    free(p);
```

why is this a problem?

M2: Not Initializing Memory

☠ *common source of bugs* ☠

- Problems?

```
int * sumptr;
int ival[100] = { …initial values here… };
int i;

sumptr = (int *) malloc ( sizeof(int) );

for (i = 0; i < 10; i++)
   *sumptr += ival[i];
```

(see `no_initialization.c` in *Code samples and Demonstrations* in *Canvas*)

M3: Stack Buffer Overflows

```
void bufoverflow (void)
{
    char buf[64];

    (void) gets(buf);
    return;
}
```

☠ common source of bugs ☠

- Problems?
- One of the biggest sources of **security** problems

Are you sure the input will be no more than 64 characters long?

M4: Writing Past End of Dynamically Allocated Memory

```
int i, sz;
int *ip, *jp;

(void) scanf ("%d", &sz);
ip = (int *) calloc (sz, sizeof(int));
…check for errors here…

jp = ip;
for (i = 0; i <= sz; i++)
   (void) scanf ("%d", jp++);
```

☠ common source of bugs ☠

why is this a problem?

M5: Freeing Unallocated Memory

Problems?

```
int i;
int *ip;

ip = &i;
…
free(ip);
```

☠ *common source of bugs* ☠

why is this a problem?

(see `allocate.c` in *Code samples and Demonstrations* in *Canvas*)

Freeing Unallocated …(cont'd)

- Problems?

```
int *ip;

ip = (int *) calloc (1000, sizeof(int));
…
free(ip);
…
free(ip);
```

☠ *common source of bugs* ☠

(see **`double_free.c`** in *Code samples and Demonstrations* in *Canvas*)

M6: Memory Leaks

☠ *common source of bugs* ☠

- Allocated memory is referenced using pointer returned by allocation
- If you lose pointers (free them, change to another address), you can no longer reference or free allocated memory
- Common problem in large, long-running programs (think: servers)
  - over time, memory footprint of program gets bigger, bigger, …

M6: Memory Leaks

```
void leak (int n)
{
    int * xp;
    xp = (int *) malloc (n * sizeof(int));
    …memory is used and then no longer needed…
    return;
}
```

☠ common source of bugs ☠

why is this a problem?

No use of `free` to release memory.

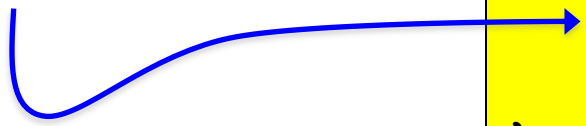
Automatic Garbage Collection?

C requires you to **manually** allocate and reclaim memory, e.g...

```
void addFirst (Object obj) {
    Node * newNode =
        (Node *) malloc (sizeof(Node));
    assert( newNode != NULL );
    newNode->data = ...;
    newNode->next = first;
    first = newNode;
}

Object removeFirst() {
    assert (first != NULL);
    Node * old = first;
    Object obj = first->data;
    first = first->next;
    free (old);
    return obj;
}
```

Programmer explicitly indicates there are no more references to the removed object



References

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.

- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.