Lecture 14. Dynamic Memory Allocation

- **The number of variables and their sizes are determined at *compile-time* — *before* a program runs**

```
/*
Read up to 1000 integers from
standard input and sort them using Quicksort.
*/
#include <stdio.h>
#include "quicksort.h"

int main(void) {
	int i, n = 0, array[1000];

	while (n < 1000 && scanf("%d", &array[n]) == 1)
		n++;
	quicksort(array, 0, n - 1);
	for (i = 0; i < n; i++)
		printf("%d\n", array[i]);
	return 0;
}
```

Suppose you want to sort 1001 integers? An unknown number of integers?

Size of the input is unknown at compile-time; it's known only at runtime

Need a way for the program to *adapt* to the size of the input

Solution: *allocate* the array at runtime, not at compile time

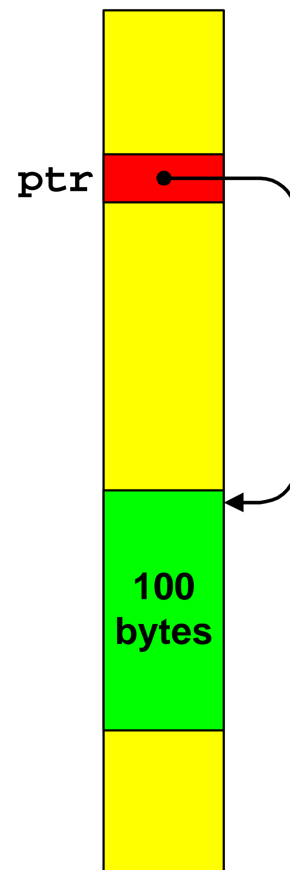
Copyright ©1996 David R. Hanson

Allocating Memory at Runtime

- **To allocate 100 bytes of memory**

```
char *ptr;

ptr = malloc(100);
if (ptr == NULL) {
    printf("Cannot allocate memory\n");
    exit(1);
}
```



`malloc` allocates a contiguous block of memory at least `100` bytes long and returns the address of the first byte

If `malloc` cannot allocate the memory requested, it returns `NULL` — *always* check! Better yet, use `emalloc` in `libmisc.a`

`malloc` returns a *generic pointer*, which can be assigned to any pointer type

```
strcpy(ptr, "Hello World!\n");
printf("%s", ptr);
```

```
Hello World!
```

- **The memory block returned by `malloc` can be accessed *only* through a pointer; no variable labels that block**

Copyright ©1996 David R. Hanson

Deallocating Memory

- To deallocate the memory pointed to by `ptr`

  `free(ptr);`

  `free` deallocates the block of memory pointed to by `ptr`

  After calling `free`, `ptr` is ***uninitialized***; using this uninitialized value is an ***error***

- Memory blocks are allocated/deallocated by ***explicit calls*** to `malloc`/`free`

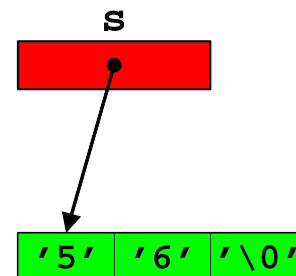
  A block allocated by `malloc` exists until a call to `free` deallocates it

  `malloc` 'creates' a block of memory, `free` 'destroys' it

- The ***lifetime*** of an allocated block is determined only by `malloc`/`free`; other function calls have ***no*** effect on its existence

```
char *itoa(int n) {
    char buf[100], *ptr;

    sprintf(buf, "%d", n);
    ptr = emalloc(strlen(buf) + 1);
    strcpy(ptr, buf);
    return ptr;
}
```




```
char *s;
s = itoa(56);
printf("%s\n", s);
```

`ptr` no longer exists,
but the memory pointed to by `s` ***does*** exist!

Copyright ©1996 David R. Hanson

Sizeof

```
int *SumPtr(int a, int b) {
    int *ptr, sum = a + b;

    ptr = emalloc(          );
    *ptr = sum;
    return ptr;
}

int *p = SumPtr(2, 5);
printf("%d\n", *p);
free(p);
```

how big is an `int`?

- `sizeof (`*type*`)` is a ***constant*** that gives the size of values of *type* in bytes

```
    ptr = emalloc(sizeof (int));
```

allocate space for an `int`

- Values given by `sizeof` are machine-dependent

	Sparc	Alpha	PCs
`sizeof (int)`	4 bytes	4 or 8	2 or 4
`sizeof (int *)`	4	8	2, 4, or 8
`sizeof (float)`	4	4	4
`sizeof (int *)`	4	8	2, 4, or 8
`sizeof (void *)`	4	8	8

These values are only typical, not exhaustive

Copyright ©1996 David R. Hanson

Sizeof, cont'd

- **The size of a structure type may *not* be the sum of the sizes of its fields**

```
struct date {
    int day, month, year;
    char monthname[4];
};
```

```
struct student {
    char name[30];
    float gpa;
    struct date birthday;
};
```

`sizeof (struct date)`	**10–32 bytes**
`sizeof (struct student)`	**54–72**

- **Use `sizeof` and `malloc/emalloc` to allocate *instances* of structure types**

```
char *months[] = { "Jan", "Feb", "Mar", "Apr", "May", Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

struct date *mkdate(int day, int month, int year) {
    struct date *ptr = emalloc(sizeof (struct date));

    ptr->day = day; ptr->month = month; ptr->year = year;
    strcpy(ptr->monthname, months[month-1]);
    return ptr;
}
```

Copyright ©1996 David R. Hanson

Dynamic Arrays

- **To sort an arbitrary number of integers**

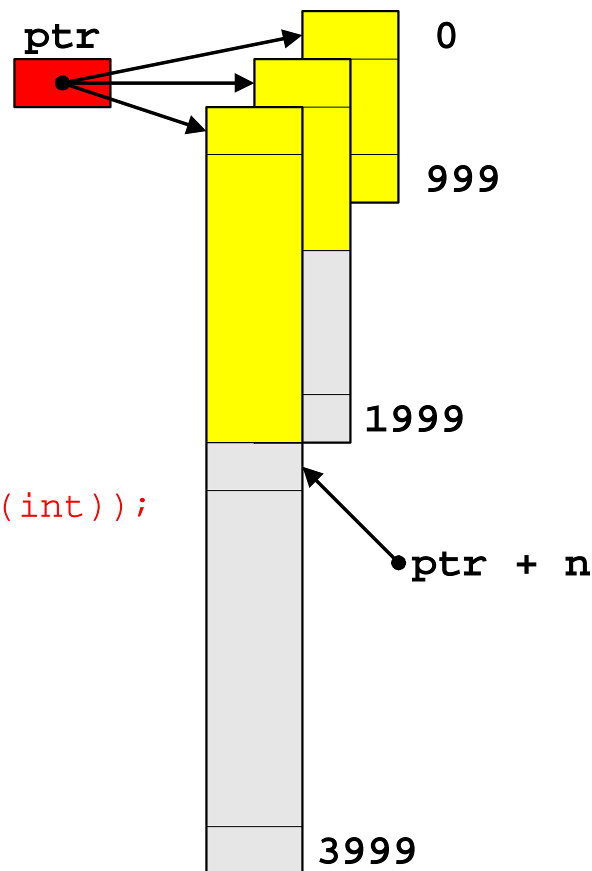
  Start with an array than can hold 1000 integers

  Double the size **of this array when more space is needed; 1000, 2000, 4000, …**

```
#include <stdio.h>
#include "quicksort.h"
#include "misc.h"

int main(void) {
    int i, n = 0, *ptr, x, size = 1000;

    ptr = emalloc(size*sizeof (int));
    while (scanf("%d", &x) == 1) {
        if (n >= size) {
            size *= 2;
            ptr = erealloc(ptr, size*sizeof (int));
        }
        ptr[n++] = x;
    }
    quicksort(ptr, 0, n - 1);
    for (i = 0; i < n; i++)
        printf("%d\n", ptr[i]);
    return 0;
}
```



Copyright ©1996 David R. Hanson

Dissecting sort2.c

```
#include "misc.h"
```

Includes the header file `misc.h`, which declares `emalloc` and `erealloc`

```
lcc -I/u/cs126/include sort2.c quicksort.c /u/cs126/lib/libmisc.a
```

Compiles `sort2.c` and `quicksort.c`, and searches `libmisc.a` to build `a.out`

```
int i, n = 0, *ptr, x, size = 1000;
```

Declares `ptr` and `size` (and `i`, `n`, and `x`), and initializes `size` to 1000

```
ptr = emalloc(size*sizeof (int));
```

Allocates space for `size` integers, and assigns the addess of this *array* to `ptr`

```
while (scanf("%d", &x) == 1) {
    ...
    ptr[n++] = x;
}
```

Reads each integer and assigns it to the next element in the array `ptr`

For any pointer `ptr`: `ptr[i]` is equivalent to `*(ptr + i)`

If `ptr` points to the first element of a dynamically allocated array:

> **`ptr + i` points to the `i`th element,**
> **so `ptr[i]` refers to the `i`th element, too**

Copyright ©1996 David R. Hanson

Dissecting sort2.c, cont'd

```
if (n >= size) {
    size *= 2;
    ptr = erealloc(ptr, size*sizeof (int));
}
```

Doubles the size of the array pointed to by `ptr`, if necessary

If `n` exceeds the current size of the array, `size` is doubled, and `erealloc` is called to expand the array accordingly

`erealloc` returns the address of the expanded array, which is assigned to `ptr`

`erealloc` is like `emalloc`: It calls the standard library function `realloc` and checks for errors

```
quicksort(ptr, 0, n - 1);
for (i = 0; i < n; i++)
    printf("%d\n", ptr[i]);
```

Sorts and prints the integers in `ptr[0..n-1]`

Common Errors

- **Failing to allocate memory**

```
int *p, i;
                                  p = emalloc(sizeof (int));
*p = i;
```

- **Failing to allocate *enough* memory**

```
p = emalloc(sizeof (int *));        p = emalloc(sizeof (int));
*p = i;

char *strsave(char *str) {
    return strcpy(emalloc(strlen(str)), str);
}                                         strlen(str) + 1
```

- **Deallocating memory that was *not* allocated by `malloc`**

```
char buf[100];
free(buf);                          free(buf);
```

- **Deallocating memory that has *already been deallocated***

```
p = emalloc(sizeof (int));
free(p);
...
free(p);                            free(p);
```

Common Errors, cont'd

- Changing the value of a pointer returned by `emalloc`, then passing it to `free`

```
char *itoa(int n) {
    char buf[100];

    sprintf(buf, "%d", n);
    return strsave(buf);
}
```

```
char *s = itoa(56);
while (*s != '\0')
    putchar(*s++);
free(s);
```

```
char *s = itoa(56), *p = s;


free(p);
```

- Thinking that `sizeof` is a ***runtime*** operation

```
int i, n, *p;

p = emalloc(sizeof (n));
for (i = 0; i < n; i++)
    p[i] = 0;
```

```
p = emalloc(n*sizeof (int));
```

- Failing to deallocate memory