Record the lecture!

GEOFF RECORD THE LECTURE!

C pointers

RECORD THE LECTURE

Pointers

Dynamic memory

Geoff's self-checklist:
- ~~❑ Start iClicker Cloud~~
- ❑ Record lecture

Record the lecture Geoff! ☹

Announcements

- Lab quizzes next week! Please attend your registered section
  - arrays
  - data types
  - loops
  - file input using `fgets` and `sscanf`

- Lab section capacity boosted slightly, STT seats released

- Lab section for DTS in talks with department
  - Uncertain status for next week

File input with `fgets` and `sscanf`

A quick aside – it's important for next week's lab quiz!

- The `fgets` function:
  - Reads a line from the specified stream and stores it into the string pointed to by `str`
  - Stops when either $n - 1$ characters are read, newline character is read, or end-of-file is reached, whichever comes first
  - Returns `str` on success, otherwise returns a null pointer

```
char* fgets(char* str, int n, FILE *stream)
```

- The `sscanf` function:
  - reads formatted input from a string
  - On success, returns the number of arguments, otherwise 0 or EOF

```
int sscanf(const char* str, const char* format, ...)
```

See `fgets_and_sscanf.c` code sample on course webpage – try it, modify it (and `data.txt`)!

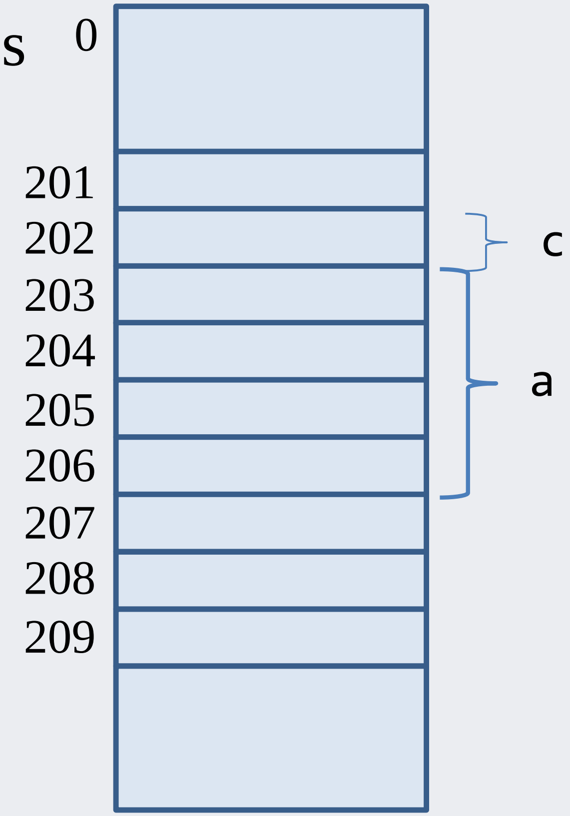
Addresses and pointers

- Every storage location in memory (RAM) has an *address* associated with it
  - The address is the location in memory where a given variable or identifier stores its data
- Can think of address in memory like a mailbox number
  - Use the address to find where the mailbox is
  - Look inside the mailbox to access the contents/value

Variable declaration

- Each byte of memory has a unique address

```
int a;
char c;
a = 5;
a++;
```



- At compile time, the compiler knows how much memory to allocate to each variable (e.g. 4 bytes for `int`, 1 byte for `char`, etc)

Addresses, &, and pointers

- You have already encountered addresses with the `scanf` function
  - `scanf` requires us to provide the address of a location using the "address of" operator, `&`
  - e.g. `scanf("%d", &a)`
  - This allows the `scanf` function to modify the value of the variable `a`, which is defined outside of `scanf`'s call stack

- A pointer is a data type that contains the address of the object in memory, but it is not the object itself

```
int a = 5;
int* p = &a;
```

  - `a` is an integer variable with the value 5
  - `p` is a pointer variable storing the address of `a`

Declaring pointers

- Pointer variables are declared as follows:

  ```
  datatype* identifier
  ```

  - e.g. `int* ptr;` or `int * ptr;` or `int *ptr;`
- Note that the type of a pointer is not the same as the type it points to
  - e.g. `ptr` is a pointer to an `int`, but is itself not an `int`

- Warning! The declaration

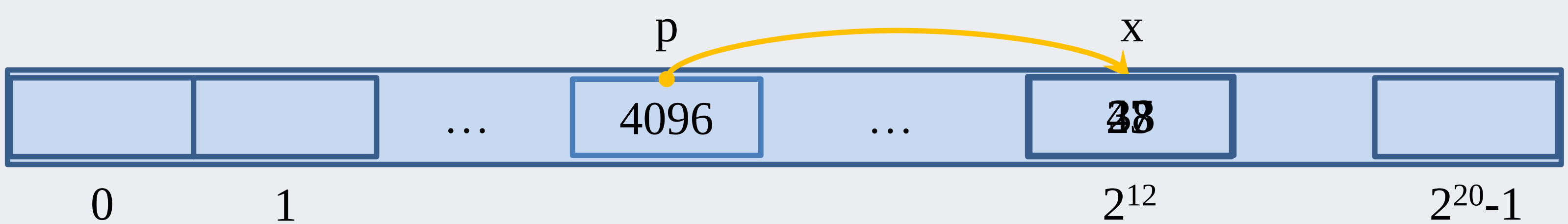
  ```
  int* var1, var2;
  ```

  - declares `var1` as a pointer, but `var2` as an integer!
- To declare both as pointers, either declare individually, or:

  ```
  int *var1, *var2;
  ```

Address operator and dereferencing

- Pointers can be assigned the address of an existing variable
  - Using the address operator, &
- The value which a pointer points to can be accessed by *dereferencing* the pointer
  - Using the * operator




```
int x = 23;
```

```
int* p = &x;
```

```
x = 47;
```

```
*p = 38;
```

Pointers as parameters

- Function parameters can be passed by reference using pointers

```
int getArraySum(int arr[], int size, int* pcount) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    if (arr[i] > 0) (*pcount)++;
    sum += arr[i];
  }
  return sum;
}
```

```
int numpositive = 0;
int numbers[] = {3, 7, -9, 5, -4};
int result = getArraySum(numbers, 5, &numpositive);
printf("Array sum: %d\n", result);
printf("Number of positive elements: %d\n", numpositive);
```

```
Array sum: 2
Number of positive elements: 3
```

Pointers as parameters

- What is output after the code on the right is executed? What is on the call stack for each function call?

```c
void f1(int arg)
{
  arg = 22;
  printf("f1 arg: %d\n", arg);
}
```

```c
void f2(int* arg)
{
  *arg = 410;
  printf("f2 arg: %d\n", arg);
}
```

```c
int x = 45;

f1(x);
printf("x after f1: %d\n", x);

f2(&x);
printf("x after f2: %d\n", x);
```

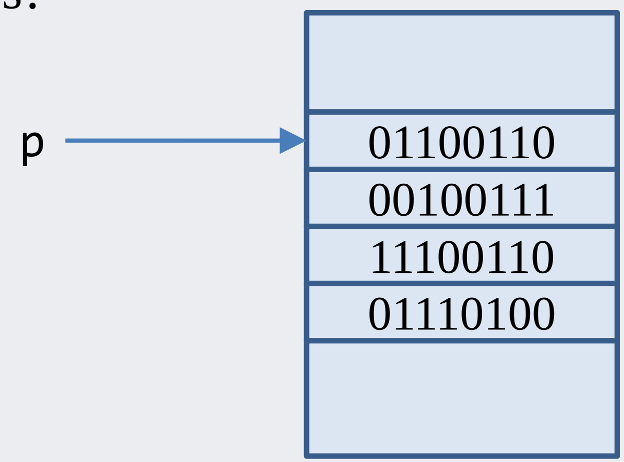
Modifying and dereferencing

- What is output by the following code?

```c
int main() {
  int a = 5;
  int* p = &a; // assume 0x5fbff8cc

  printf("value of p = %p\n", p);
  printf("dereferenced p = %d\n", *p);

  p++;

  printf("value of p+1 = %p\n", p);
  printf("dereferenced p+1 = %d\n", *p);
}
```

Pointer types

- `int*` `p` is a pointer to an integer (at some 32-bit address)
- `char*` `p` is a pointer to a character (also at some 32-bit address)
  - Can we use a generic type for pointers?



- Can we dereference `p` without knowing the type of the variable that it is pointing to?
  - `int`: 4 bytes
  - `char`: 1 byte

Generic pointers

- Generic pointers can be declared, but must be cast before they can be dereferenced

```c
int main() {
  int x = 10;
  char ch = 'A';
  void* gp;

  gp = &x;
  printf("integer value = %d\n", *(int*)gp); // outputs 10

  gp = &ch;
  printf("now points to char %c\n", *(char*)gp); // outputs A

  return 0;
}
```

Pointer to a pointer?

```
int main() {
  int x = 5;
  int* p = &x;
  *p = 6;
  int** q = &p;
  int*** r = &q;

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

- "You can keep adding levels of pointers until your brain explodes or the compiler melts – whichever happens soonest"
  - stackoverflow user JeremyP

Back to call-by-reference

- Consider the following function that adds two parameters supplied by reference

```
int  add(int* num1, int* num2) {
  int sum = *num1 + *num2;
  return  sum;
}

int main() {
  int  a = 2;
  int  b = 4;
  int  c = add(&a, &b);
  printf("sum = %d",  c);
}
```

- Can we modify the add function so that it uses a pointer to return the answer?

Returning pointers

- Will it work if we just change the return type to pointer, and return the sum variable's address?

```
int* add(int* num1, int* num2) {
  int sum = *num1 + *num2;
  return &sum;
}

int main() {
  int  a = 2;
  int  b = 4;
  int* c = add(&a, &b);
  printf("sum = %d", *c);
}
```

This will have problems!
Think about what is (or was) on the call stack

Passing array elements as parameters

- Arrays are passed by reference by default

```
double getMaximum(double data[], int size); // prototype
double getMaximum(double* data, int size); // equivalent prototype

double answer = getMaximum(myarr, length); // function call
```

- Note that we do not need to provide "&" when specifying the address of the entire array (i.e. the address of the first element)
- If we want to specify the address of an individual element of the array, we would need the address operator
  - e.g. `&data[4]`

Pointer arithmetic

- If we know the address of the first element of an array, we can compute the addresses of the other array elements
  - (or whatever comes after, if it is meaningful)

```
int A[5];
int* q = &A[0];
printf("q address: %p\n", q);
A[0] = 2;
A[1] = 4;

printf("value of q: %d\n", *q);
printf("value of q+1: %d\n", *(q + 1));
```

```
int x = 5;
int* p = &x;
printf("p address: %p\n", p);
printf("value of p: %d\n", *p);
printf("value of p+1: %d\n", *(p + 1));
```

Dynamic memory

Dynamic memory

- Arrays declared as local variables must have a known size at compile time
  - but sometimes we don't how much space we need until runtime
- Suppose we expect users to only need to store up to 1000 values, so we hard-code "1000" as an array size
  - What if user needs more? Change code and recompile
  - What if user only needs 5? Wastes memory
- If the value 1000 is hard-coded, this is hard to find and change
  - especially if used in multiple locations
- If hardcoded as a symbolic constant, still cannot change without recompiling

Memory management in C

- We have already seen how locally-declared variables are placed on the function call stack
  - allocation and release are managed automatically
- The available stack space is extremely limited
  - placing many large variables or data structures on the stack can lead to stack overflow
- Stack variables only exist as long as the function that declared them is running

Dynamic memory allocation

- At run-time, we can request extra space on-the-fly, from the *memory heap*

- Request memory from the heap – "allocation"
- Return allocated memory to the heap (when we no longer need it) – "deallocation"

- Unlike stack memory, items allocated on the heap must be explicitly freed by the programmer

Dynamic memory allocation

- Function `malloc` returns a pointer to a memory block of at least `size` bytes:

```
ptr = (cast-type*) malloc(byte-size);
```

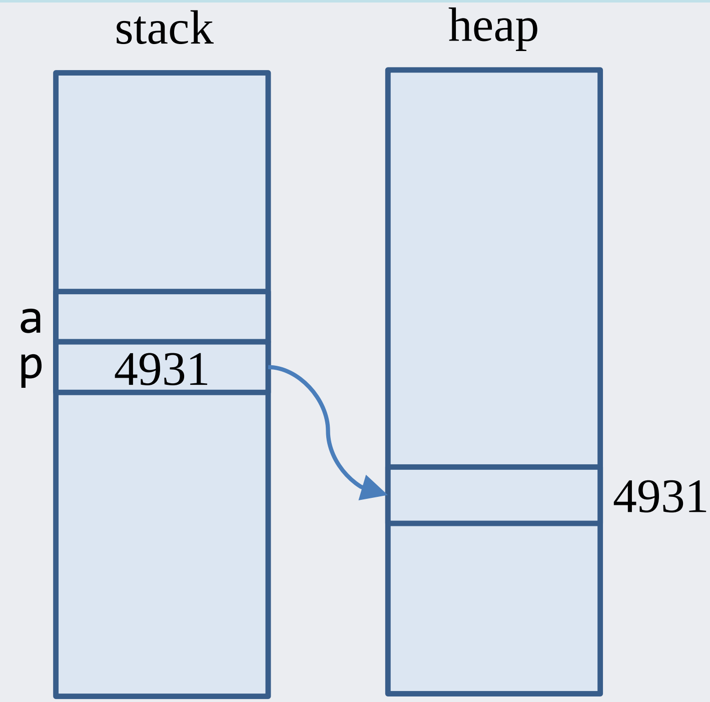
- Function `free` returns the memory block (previously allocated with `malloc`) and pointed to by `ptr` to the memory heap:

```
free(ptr);
```

- The system knows how many bytes need to be freed, provided we supply the correct address `ptr`

Heap example

```
int main() {
  int a;
  int *p = (int*) malloc(sizeof(int));
  *p = 10;
}
```



- If there is no free memory left on the heap, `malloc` will return a null pointer
- Note: `malloc` only allocates the space but does not initialize the contents.
  - Use `calloc` to allocate and clear the space to binary zeros

Allocating dynamic arrays

- Suppose we want to allocate space for exactly 10 integers in an array

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int* i;
  i = (int*) malloc(10*sizeof(int));
  if (i == NULL) {
    printf("Error: can't get memory...\n");
    exit(1); // terminate processing
  }

  i[0] = 3;  // equivalent: *(i+0) = 3;
  i[1] = 16; // *(i+1) = 16;
  printf("%d", *i);
  ...
}
```

Allocating dynamic arrays

From user input, variable array size

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int employees, index;
  double* wages;
  printf("Number of employees? ");
  scanf("%d", &employees);

  wages = (double*) malloc(employees * sizeof(double))
  if (!wages) { // equivalent: if (wages == NULL)
    printf("Error: can't get memory...\n");
  }

  printf("Everything is OK\n");
  ...
}
```

See `dma_examples.c`

Dangling pointers

- When we are done with an allocated object, we free it so that the system can reclaim (and later reuse) the memory

```
int main() {
  int* i = (int*) malloc(sizeof(int));
  *i = 5;
  free(i);


  printf("%d", *i);
}
```

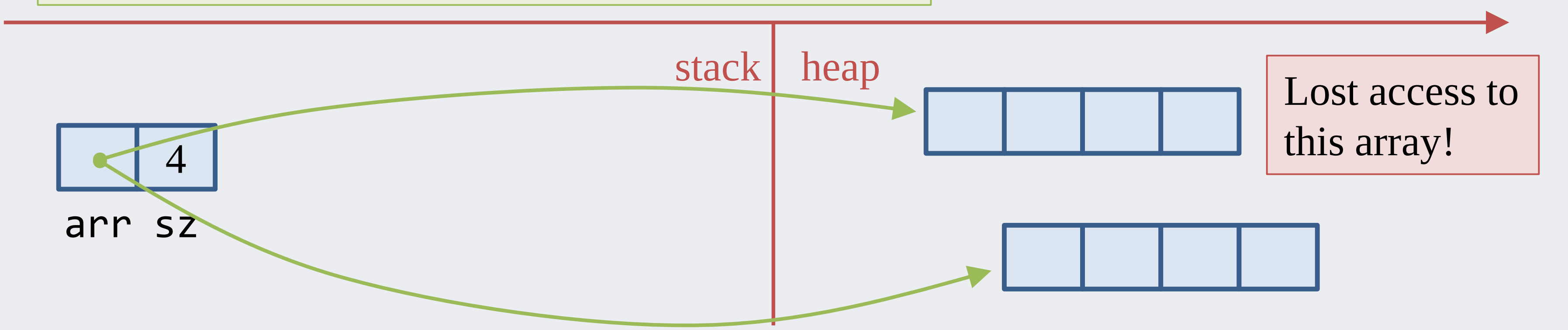
The space is marked as free, but the value remains until it is overwritten

- If the pointer continues to refer to the deallocated memory, it will behave unpredictably when dereferenced (and the memory is reallocated) – a dangling pointer
  - Leads to bugs that can be subtle and brutally difficult to find
  - So, set the pointer to `NULL` after freeing `i = NULL;`

Memory leaks

- If you lose access to allocated space (e.g. by reassigning a pointer), that space can no longer be referenced, or freed
  - And remains marked as allocated for the lifetime of the program

```
int* arr;
int sz = 4;
arr = (int*) malloc(sz*sizeof(int));
arr[2] = 5;
arr = (int*) malloc(sz*sizeof(int));
arr[2] = 7;
```



See `memory_leak.c`

Exercise

- What is printed to the screen?
  - Also clearly identify memory leaks and dangling pointers

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x=%d, *y=%d, z=%d, w=%d\n", *x, *y, z, w);
```

Readings for this lesson

- Thareja
  - Appendices A, B, E

- Next class:
  - Thareja, Chapters 4 – 5