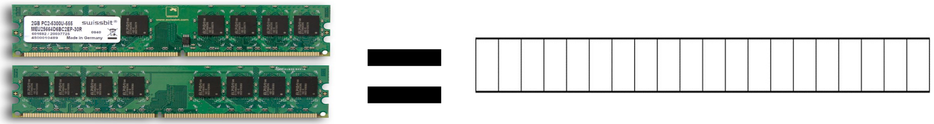
CSE 374 Programming concepts and tools

Winter 2024 Instructor: Alex McKinney

Review

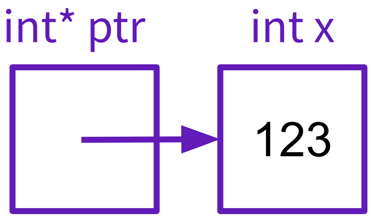
All data is binary - a list of 1's and 0's

- A single digit is called a **bit**
- Bits come in groups of 8, called a **byte**



Computers store data in files or in memory.

An address refers to a location in memory.



A **pointer** is a data object that holds an address.

- Address can point to *any* data, because they simply point to any space in memory

Review: Pointers

Storing in memory an address to another location in memory

```
int x = 4;        // Variable called ‘x’ of type ‘int’ given value ‘4’
int* xPtr = &x; // Variable called ‘xPtr’ of type ‘int pointer’ given value ‘location of x’
int xCopy = *xPtr;
      // Variable called ‘xCopy’ of type ‘int’ given value ‘value found at address xPtr’ (read)
*xPtr = 123;   // Assigning the value '123' to the ‘value found at address xPtr’  (write)
int* noPtr = NULL; // variable called ‘noPtr; of type ‘int pointer’ given value of ‘null’
```

Output Parameters

Lifetime and pointers

Review: Using Pointers as Output Parameters

C pointers offer a powerful mechanism for **returning multiple values** from a function.

- Pass the memory address of variables to be modified as function arguments.

```
void initialize(int *a, char *c);
int main(void) {
    int a = 0;
    char c;                 // c is undefined (don’t do this)
    initialize(&a, &c);     // a = 10, c = 'A'
}
void initialize(int *a, char *c) {
    *a = 10;
    *c = 'A';
}
```

Lifetimes and Pointers

```
void foo(int* x_ptr);
void main() {
  int x = 42;
  foo(&x);
}
void foo(int* x_ptr) {
  printf("x has %d\n", *x_ptr);
}
```

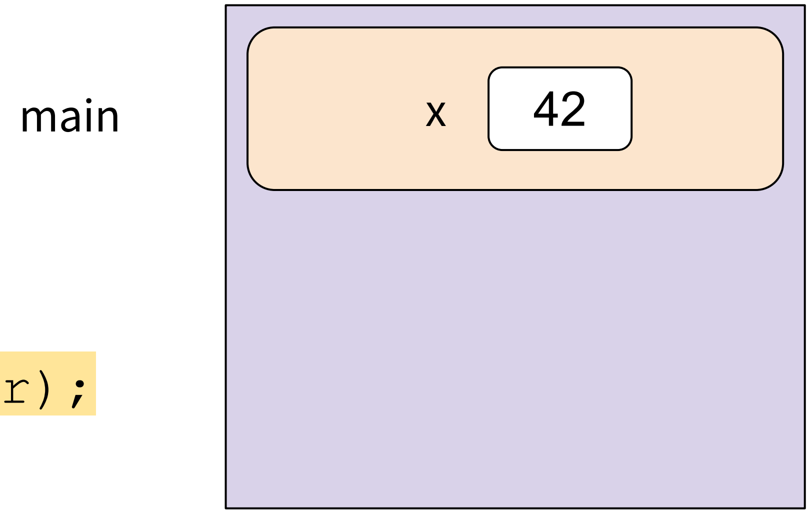
When we get to `printf()` ...

- Is `x_ptr` alive?
- Is `x` alive?

Lifetimes and Pointers

```c
void foo(int* x_ptr);
void main() {
   int x = 42;
   foo(&x);
}
void foo(int* x_ptr) {
   printf("x has %d\n", *x_ptr);
}
```

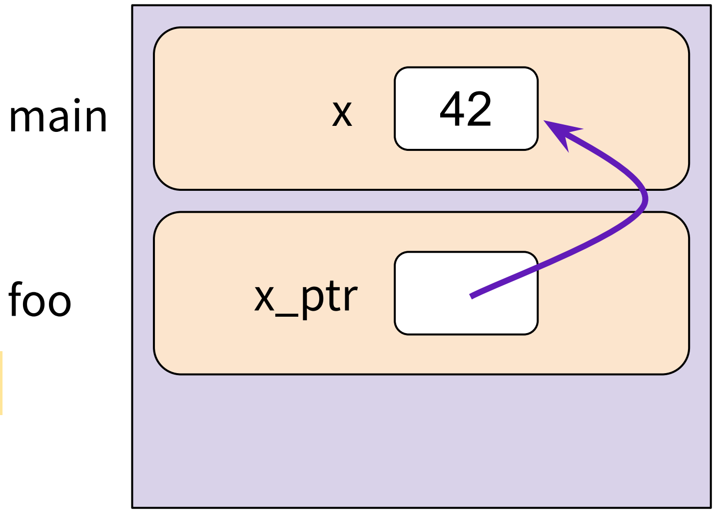
Is `x_ptr` alive? Is `x` alive?



Lifetimes and Pointers

```c
void foo(int* x_ptr);
void main() {
   int x = 42;
   foo(&x);
}
void foo(int* x_ptr) {
   printf("x has %d\n", *x_ptr);
}
```

Is x_ptr alive? Is x alive?



Yes, x_ptr and x are both alive.

Lifetimes and Pointers

```c
int* foo();
void main() {
   int* x_ptr = foo();
   printf("x has %d\n", *x_ptr);
}
int* foo() {
   int x = 42;
   return &x;
}
```

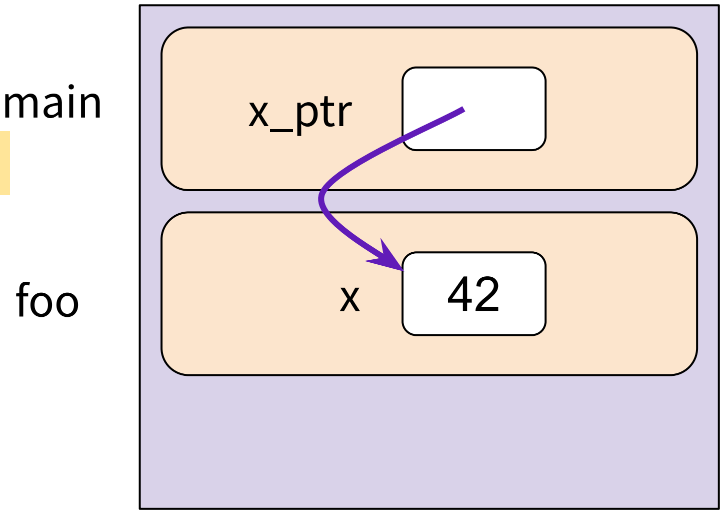
When we get to `printf()`...

- Is `x_ptr` alive?
- Is `x` alive?

Lifetimes and Pointers

```c
int* foo();
void main() {
  int* x_ptr = foo();
  printf("x has %d\n", *x_ptr);
}
int* foo() {
  int x = 42;
  return &x;
}
```

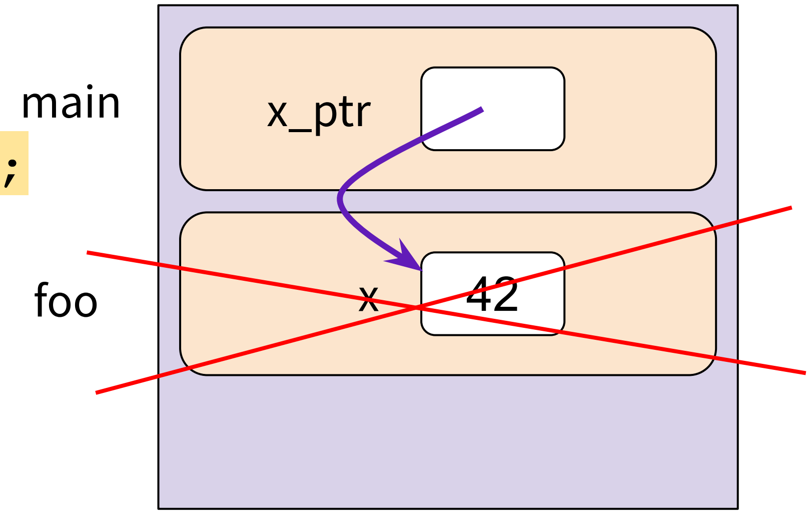
Is `x_ptr` alive? Is `x` alive?



Lifetimes and Pointers

```c
int* foo();
void main() {
   int* x_ptr = foo();
   printf("x has %d\n", *x_ptr);
}
int* foo() {
   int x = 42;
   return &x;
}
```

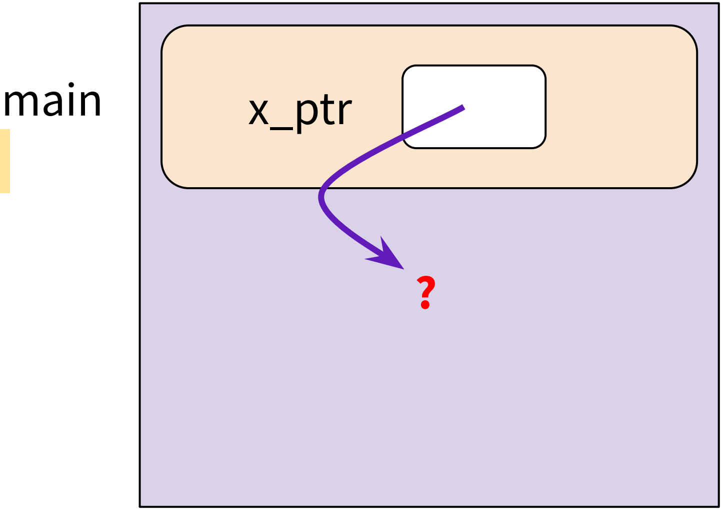
Is `x_ptr` alive? Is `x` alive?



Lifetimes and Pointers

```c
int* foo();
void main() {
  int* x_ptr = foo();
  printf("x has %d\n", *x_ptr);
}
int* foo() {
  int x = 42;
  return &x;
}
```

Is x_ptr alive? Is x alive?



x_ptr is alive, but x is not alive.

Dangling Pointer

When a pointer points to a dead local variable, it is "dangling"

Dereferencing a dangling pointer is undefined behavior (UB)

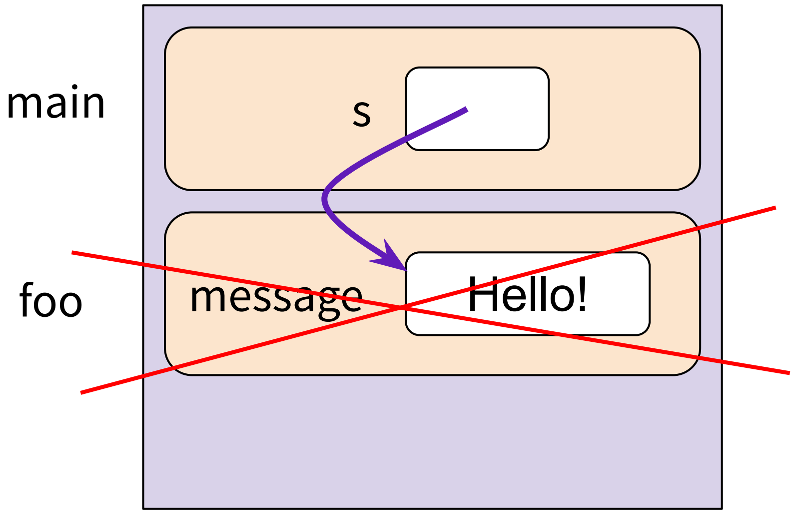
- UB means that anything *could* happen
- Your program could crash (best case) or silently fail (worse case)

Luckily, `gcc` can often catch this kind of error with a warning

- But not always, you shouldn't trust the C compiler much, in general

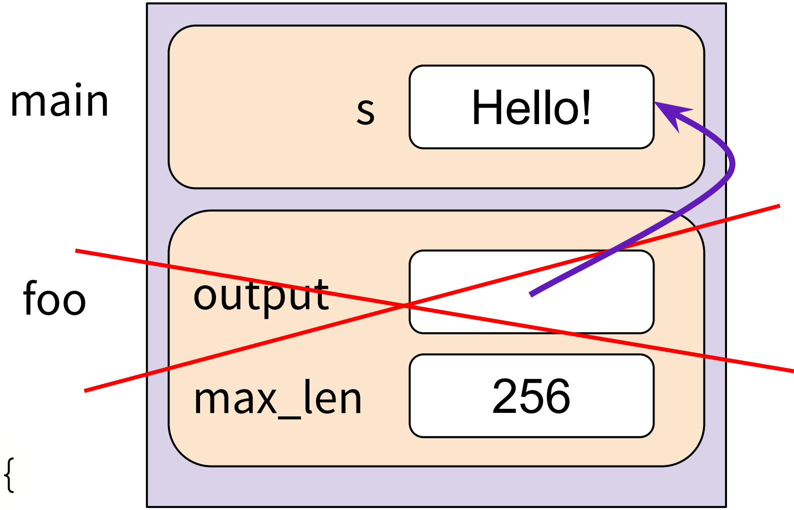
Example: Returning a String

```
char* foo();
int main() {
  char* s = foo();
  printf("String: %s\n", s);
}
char* foo() {
  char message[256] = "Hello!";
  return message;
}
```



Fix: Output Parameter

```c
#include<stdio.h>
#include<string.h>
void foo(int max_len, char* output);
int main() {
  char s[256];
  foo(256, s);
  printf("String: %s\n", s);
}
void foo(int max_len, char* output) {
  strncpy(output, "Hello!", max_len);
}
```



Output Parameters

We can't *return* a pointer to a local variable

Instead, take in a pointer as a parameter, used for output

- Remember, pointers can be used for reading and writing

```
void foo(int max_len, char* output);
```

Avoid returning/using dangling pointers pointing at data that has gone out of scope…

```
int* bad() {
    int num = 8;
    return &num;
}
```

Questions?

Dynamic Memory Allocation

Memory Allocation

Allocation refers to any way of asking for the operating system to set aside space in memory.

How much space? Based on variable type & your system.

- to get specific sizes for your system use `sizeof(<datatype>)` function in `stdlib.h`

Type	Storage Size	Value Range
char	1 byte	-128 to 127 or 0 to 255
int	2 or 4 bytes	-32,786 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
long	8 bytes	-9223372036854775808 to 9223372036854775807
float	4 bytes	1.2E-38 to 3.4E+38
double	8 bytes	2.3E-308 to 1.7E+308
long double	10 bytes	3.4E-4932 to 1.1E+4932

* pointers require space needed for an address – dependent on your system - 4 bytes for 32-bit, 8 bytes for 64-bit

Memory Allocation So Far

So far, we have seen two kinds of memory allocation:

- Global Variables – static memory allocation
  - space for global variables is set aside at compile time, stored in RAM next to program data, not stack
  - space set aside for global variables is determined by C based on data type
  - space is preserved for entire lifetime of program, never freed

```c
int counter = 0;      // global var
int main(int argc, char** argv) {
  counter++;
  printf("count = %d\n",counter);
  return EXIT_SUCCESS;
}
```

Memory Allocation So Far

- Local variables – automatic memory allocation
  - space for local variables is set aside at start of function, stored in stack
  - space set aside for local variables is determined by C based on data type
  - space is deallocated on return

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main(int argc, char** argv) {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return EXIT_SUCCESS;
}
```

Dynamic Allocation

There are some situations where static and automatic allocation aren’t sufficient:

- Need memory that persists across multiple function calls
- Memory size is not known in advance to the caller
- Something like…

```
char* ReadFile(char* filename) {
  int size = GetFileSize(filename);
  char* buffer = AllocateMem(size);

  ReadFileIntoBuffer(filename, buffer);
  return buffer;
}
```

You don’t know how big the file size is!

Implicit vs. Explicit Allocation

What we want is ***dynamically***-allocated memory. The memory persists until either:

- A garbage collector collects it (*automatic memory management*)
  - **Implicit** memory allocator: programmer only allocates space, doesn't free it
  - Example: `new` in Java
- Your code explicitly deallocated it (*manual memory management*)
  - **Explicit** allocator: programmer allocates space and frees it up when finished
  - C requires you to manually manage memory
    - Gives you more control, but also causes headaches 🤕
  - Example: `malloc` and `free` in C

Memory is allocated from the **heap**, not the stack!

Multiple Ways to Store Program Data

Static global data

- *Fixed size* at compile-time
- Entire *lifetime of the program*

Stack-allocated data

- Local/temporary variables
- *Known lifetime* (deallocated on `return`)

Dynamic (heap) data

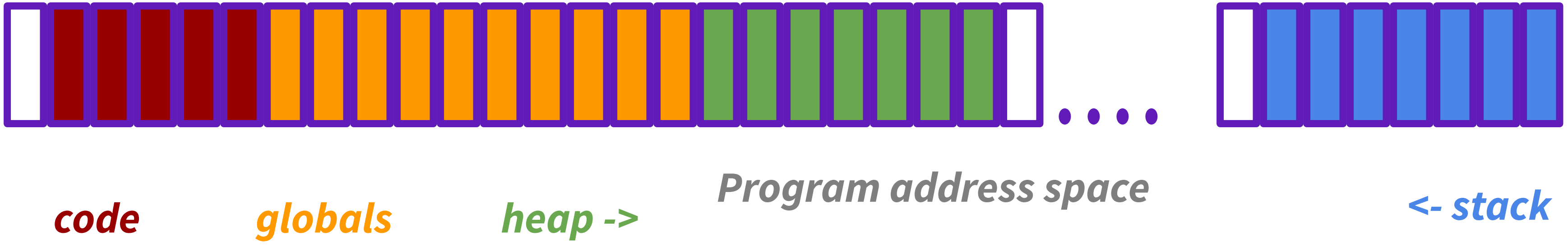
- Size known only at runtime (*i.e.* based on user-input)
- Lifetime known only at runtime (long-lived data structures)

```c
int count = 0;

void foo(int n) {
  int tmp;
  int local_array[n];

  int* dyn =
    (int*)malloc(n*sizeof(int));
}
```

Review: Working memory



As a program executes it interacts with the computer's working memory:

- **Code**: space for the code compiled instructions
- **Globals**: space for global variables, static constants, string literals, etc.
- **Heap**: holds dynamically allocated variables (`new` or `malloc` variables)
- **Stack**: holds current instructions, each function in a frame

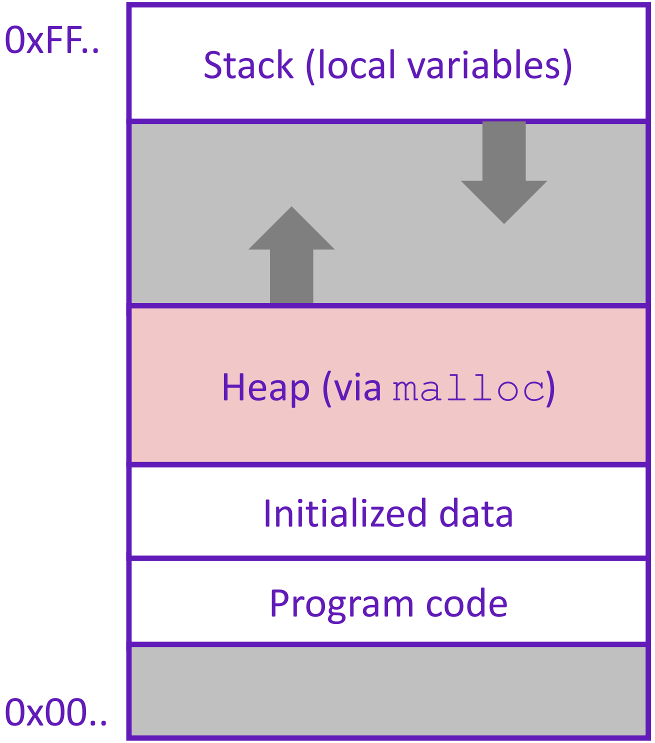
Address Space Visualization

Higher addresses on top, lower addresses on bottom

- Space is first set aside for program code
- Then for the initialized data (global variables, constants, string literals)

As program runs…

- Heap grows up as we dynamically allocate more memory
- Stack grows down as we call functions
- When the space between the stack and heap is full - crash (out of memory)

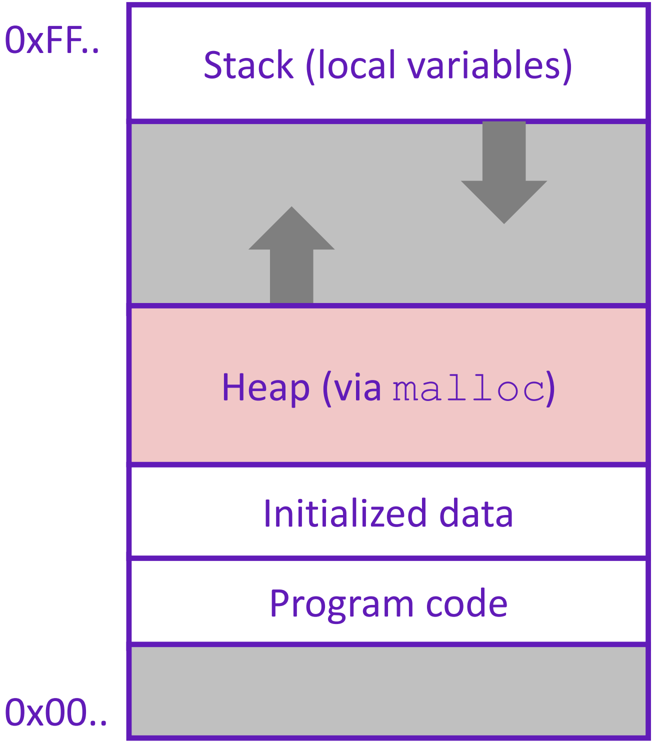


The Heap

The Heap is a large pool of available memory used to hold dynamically-allocated data

- **`malloc`** allocates chunks of data in the Heap; **`free`** deallocates those chunks
- **`malloc`** maintains bookkeeping data in the Heap to track allocated blocks
  - That's why bound checking is important! If we write beyond the allocated space, we might destroy the bookkeeping data.

Address Space Visualization



Questions?

Allocating Memory in C

Need to `#include <stdlib.h>`

```
void* malloc(size_t size)
```

- Allocates a continuous block of `size` bytes of **uninitialized** memory
- `void*` means a pointer to any type (e.g. `int`, `float`, `char`)
- Returns a pointer to the beginning of the allocated block
- Returns `NULL` if allocation failed or `size==0`
  - Allocation fails if out of memory, very rare but always check before using!
- Different blocks not necessarily adjacent
- Read/writing memory next to the returned block is **undefined behavior**

malloc()

General usage:

```
var = (type*) malloc(size in bytes)
```

Cast `void*` pointer to known type

- Put the type you want to convert to in parentheses before the variable

`sizeof()` makes code portable to different machines

```
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
  return 1;
}
...    // do stuff with arr
```

Special Case: Strings

When you allocate space for a string, you need to include space for the `\0`

This is a common mistake!

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

char* str = “Hello World!\n”
char* copy = (char*) malloc(sizeof(char) * (strlen(str) + 1));
if (copy == NULL) {
  return 1;
}
strcpy(copy, str);
...   // do stuff with copy
```

Casting

You can ask C to convert a variable into another type using casting

Put the type you want to convert to in parentheses before the variable

```
long x = 123;
int* ptr = (int*) x;
printf("%d\n", *ptr);
```

This will compile without warnings, but it is a really bad idea!

- `long` and `int*` have different sizes
- We would be dereferencing memory at location `123` (undefined behavior)

Aside: `errno` (must `#include <errno.h>`)

How do you know if an error has occurred in C?

- C does not have exceptions like Java

Usually, return a special error value (e.g. NULL)

`stdlib` functions set a global variable called `errno`

Can check error type by comparing to error code

- `if (errno == ENOMEM)  // Allocation failure`

Or call `perror(message)` to print to stderr

- `perror("malloc");  // "malloc: Cannot allocate memory"`

Reading Uninitialized Memory

Unlike Java, in C variables are un-initialized by default

- Meaning that they could hold any value before we first write to them
- Invalid read – reading from memory before you have written to it

```c
int i;
printf("%d\n", i);  // Invalid read!
i = 374;
printf("%d\n", i);  // OK, was initialized to 374

float* fptr = (float*) malloc(sizeof(float));
if (fptr == NULL) {
  // Handle error ...
}
printf("%f\n", *fptr);  // Invalid read!
```

`calloc()`

General usage:

```
var = (type*) calloc(numOfElements, bytesPerElement)
```

Like **`malloc`**, but also zeros out the block of memory (i.e. calloc == *clear* and *allocate*)

- i.e. **initializes** memory to all 0s
- Slightly slower; but useful for non-performance-critical code
- Also in `stdlib.h`

```c
// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
  // Handle error ...
}
printf("%f\n", arr[0]); // OK, will print 0.00000
```

`realloc()`

General usage: `var = (type*) realloc(p, newSizeInBytes)`

Creates a new allocation with given size, copies the contents of `p` into it and then frees `p`

- Saves a few lines of code
- Can sometimes be faster due to allocator optimizations
- Also in `stdlib.h`

```
// allocate a 10-int array
int* arr = (int*) calloc(10, sizeof(int));
if (arr == NULL) {
  // Handle error ...
}
// reallocate the memory
arr = (int*) realloc(arr, sizeof(int)*20);
```

Freeing Memory in C

Need to `#include <stdlib.h>`

`void free(void* p)`

- Releases whole block pointed to by `p` to the pool of available memory
- Pointer `p` must be the address *originally* returned by **`m/c/realloc`** (*i.e.* beginning of the block), otherwise system exception raised
- Pointer is unaffected by call to free
  - Defensive programming: can set pointer to `NULL` after freeing it
- Don't call `free` on a block that has already been released (double free)
  - Undefined behavior, best case program crashes

`free()`

Usage:

```
free(pointer);
```

Rule of thumb: for every call to `malloc` there should be one call to `free`

```
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  // Handle error ...
...                // do stuff with arr
free(arr);
arr = NULL;        // OPTIONAL
free(arr);         // Bad! Double free is undefined
```

Questions?

Poll Question (PollEv.com/cs374)

Which is a **correct** use of `malloc` and `free`?

```
int x[] = {1, 2, 3};
int sum = x[0]+x[1]+x[2];
free(x);
```

A.

```
long* make_array(void) {
 return
(long*)malloc(sizeof(long)*5);
}
// later...
long* val = make_array();
free(val);
```

B.

```
char* str =
   (char*) malloc(sizeof(char)*5);
free(str);
strcpy(str, ”Hello”);
```

C.

```
char** strings =
   (char**)malloc(sizeof(char)*5);
// use strings…
free(strings);
```

D.

Which is a correct use of malloc and free?

0

A — 0%

B — 0%

C — 0%

D — 0%

Memory Allocation Example in C

```
void foo(int n, int m) {
  int i, *p;                                // declare local variables
  p = (int*) malloc(n*sizeof(int));         // allocate block of n ints
  if (p == NULL) {                          // check for allocation error
    perror("malloc");                       // prints error message to stderr
    exit(1);
  }
  for (i=0; i<n; i++)                       // initialize int array
    p[i] = i;
                                            // add space for m ints to end of p block
  p = (int*) realloc(p,(n+m)*sizeof(int));
  if (p == NULL) {                          // check for allocation error
    perror("realloc");
    exit(1);
  }
  for (i=n; i < n+m; i++)                   // initialize new spaces
    p[i] = i;
  for (i=0; i<n+m; i++)                     // print new array
    printf("%d\n", p[i]);
  free(p);                                  // free p, pointer will be freed at end of
function
}
```

Ex9 due Wednesday, HW3 due Sunday!

Ex9 due before next Wednesday's lecture

- **This exercise has a later due date because we will talk more about `malloc()` in the next lecture.**
- Link available on the website: https://courses.cs.washington.edu/courses/cse374/24wi/exercises/

HW3 due Sunday 11.59pm!

- Instructions on course website: https://courses.cs.washington.edu/courses/cse374/24wi/homeworks/hw3/

Bonus Slides 🎁

Wouldn’t it be nice…

If we never had to free memory?

Do you free objects in Java?

- Reminder: *implicit* allocator

There are two main kinds of implicit allocators

- Garbage collection
- Reference counting (C++ smart pointers)

Garbage Collection (GC)

(Automatic Memory Management)

Garbage collection: automatic reclamation of heap-allocated storage – application never explicitly frees memory

```
void foo() {
    int* p = (int*) malloc(128);
    return;  /* p block is now unreachable! */
}
```

Used in many modern languages:

- Lisp, Haskell, Java, C#, Ruby, Python, JavaScript, Go, MATLAB, and more…

Variants ("conservative" garbage collectors) exist for C and C++, but cannot collect all garbage

Garbage Collection

How does the memory allocator know when memory can be freed?

- In general, we cannot know what is going to be used in the future since it depends on conditionals
- But, we can tell that certain blocks cannot be used if they are *unreachable* (via pointers in registers/stack/globals)

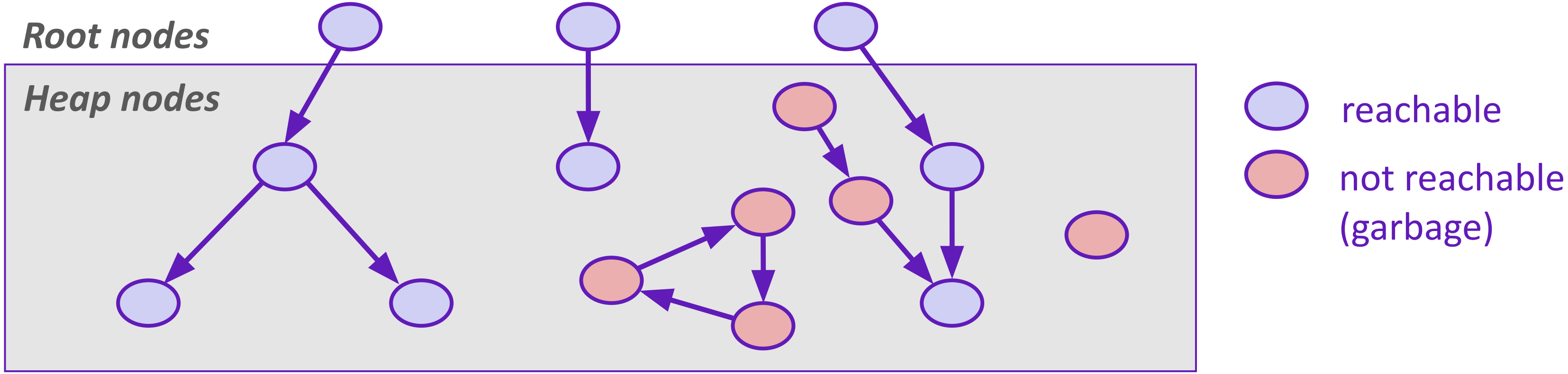
Memory allocator needs to know what is a pointer and what is not – how can it do this?

- Sometimes with help from the compiler

Memory as a Directed Graph

Each allocated heap block is a node in the graph

Each pointer is an edge in the graph



A node (block) is *reachable* if there is a path from any root to that node

- A root node is a global or local variable (or any other reachable object)

Non-reachable nodes are *garbage* (cannot be needed by the application)

Garbage Collection

Dynamic memory allocator can free blocks if there are <u>no pointers to them</u>

How can it know what is a pointer and what is not?

We'll make some *assumptions* about pointers:

- Memory allocator can distinguish pointers from non-pointers
- All pointers point to the start of a block in the heap
- Application cannot hide pointers
(*e.g.* by coercing them to a long, and then back again)

Mark and Sweep

Interrupt the program at a regular interval

- "Stop the world" pauses

Deallocate any unreachable nodes

Resume program execution until the next pause and repeat