More on Pointers

Null pointers

- In Java we have the keyword **null**, which is the value of an uninitialized 'reference type'
- In C we sometimes use NULL, but its just a macro for the integer 0
  - Pointers are initialized to 0 to indicate 'address 0' which indicates that the pointer points nowhere useful.
- Dereferencing a NULL pointer will segmentation fault.
- See pointers/null_pointers.c

Dangling pointers

- Dangling pointers (aka wild pointers, dangling references) are pointers that do not point to a *valid object* of the appropriate *type*.

- You can create these in a number of ways..
  - Returning a pointer to an automatic variables from a functions
  - Faulty pointer arithmetic
  - Casting a pointer to an unrelated pointer type.
  - More here https://en.wikipedia.org/wiki/Dangling_pointer

- See pointers/wild_pointers.c and pointers/casting_pointers.c

Void pointers

- Void pointers (void *) point to objects of unspecified type, and can therefore be used as "generic" data pointers.
- Moreover, void pointers represent addresses without any type information, just a location in memory.
  - Void pointers cannot be dereferenced.
  - Pointer arithmetic on them is not allowed.
- They can easily be (and in many contexts implicitly are) converted to and from any other object pointer type.
- See pointers/void_pointers.c

Double pointers

- Since we can have pointers to int, and pointers to char, and pointers to any structures we've defined, it shouldn't come as too much of a surprise that we can have *pointers to other pointers*.

- Or even pointers to pointers to pointers!

- For example, if you want..
  - a list of characters (a word), you can use **char* word**
  - a list of words (a sentence), you can use **char** sentence**
  - a list of sentences (a paragraph), you can use **char*** paragraph**
  - … and so on.

Double pointers *con't*

- Consider…

```
int     a =  3;
int*    b = &a;
int**   c = &b;
int***  d = &c;
```

- Here are how the values of these pointers equate to each other…

```
*d == c && *c == b;

// therefore…
**d == *c == b

// then clearly…
***d == **c == *b == a == 3;
```

- We'll see a practical example of this in a linked list implementation we'll look at later this lecture (or maybe next).

+

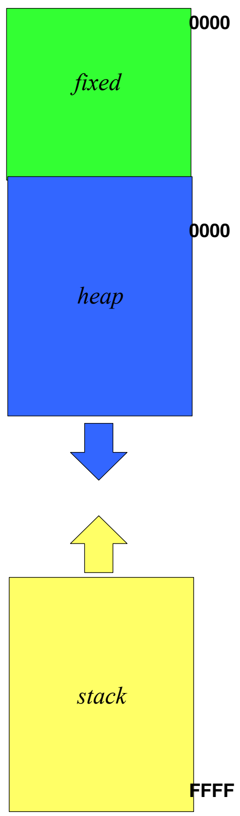
Types of Memory

Memory management in C

- The C programming language manages memory ***statically***, ***automatically*** or ***dynamically***.
- Depending on the *how and where a variable is declared in your source code*, the memory associated will be managed in one of these three ways.
- Each ***strategy*** for memory management corresponds to a particular ***region*** in memory.
- Each ***region*** and ***management strategy*** has its own characteristics and behaviors that you must understand.

Three memory regions

- When you run a a program, space is allocated from one of several ***memory regions*** depending on the the thing being allocated for.
- One region of memory is reserved for data that is never created or destroyed as the program runs. This is called ***fixed or static memory***.
- One region is reserved for data that needs to be allocated *dynamically*. This is called ***heap memory***.
- On region is reserved for automatic (local variables) defined inside a function. This is called ***stack memory*.**



Static memory

- Things allocated in static memory…
  - Executable code
  - Global variables
  - Constant structures (constant arrays, strings, structs etc.)
  - Static variables
- Location decided at compilation time.
- (This is a bit of hand-waving. We'll talk more about this later)

Stack memory

- Things allocated in stack memory…
  - Local variables for functions whose…
    - size can be determined at call time.
    - lifecycle is tied to execution of function itself.
- There is a limit on the size of variables that can be stored on the stack.
  - (C99 relaxed this constraint somewhat.)

Heap memory

- Things allocated in heap memory…
  - Structures whose size varies dynamically
    - e.g. length of arrays or strings decided/modified at runtime.
  - Structures that are allocated dynamically
    - e.g. records in a linked list.
  - Structures created by a function that must *survive after the call returns*.

Stack Region

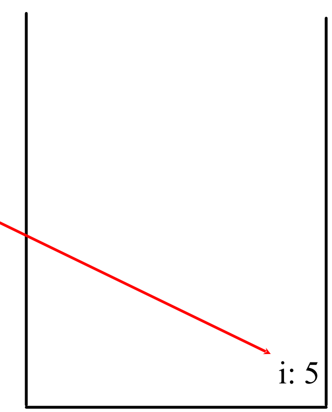
+ Basics

- The stack memory region works like the stack data structure.
  - What gets pushed and popped from it are “stack frames”.
- Every time a function is called a “stack frame” is pushed.
  - You can think of a “stack frame” as a memory ‘chunk’ for all the automatic variables in a function.
- When the method returns, the “stack frame” gets popped all the memory associated with that function call is effectively deallocated.
  - That region of memory becomes available for other use.

Trace the Call Stack

i is declared and initialized

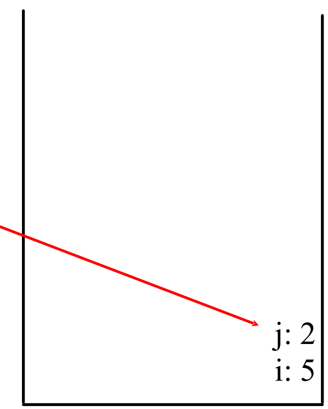
```
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



+ Trace the Call Stack

j is declared and initialized

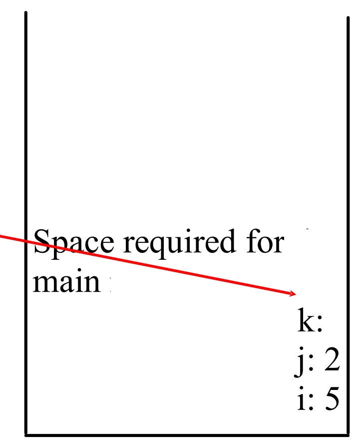
```c
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace the Call Stack

Declare k

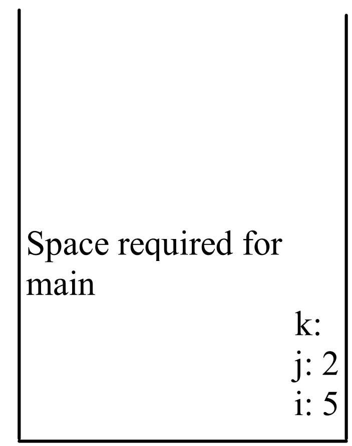
```c
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



+ Trace the Call Stack

Invoke max(i, j)

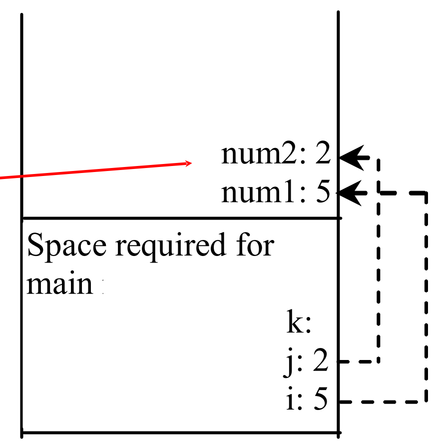
```c
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace the Call Stack

copy the values of i and j to num1 and num2

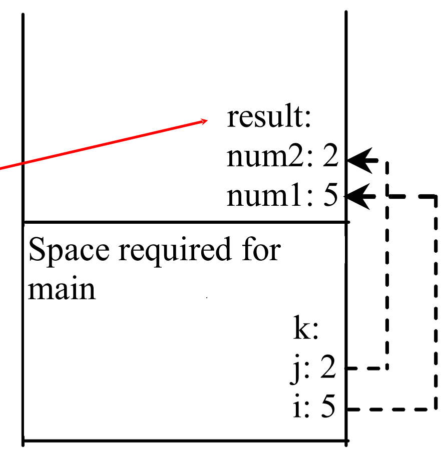
```c
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace the Call Stack

Declare result

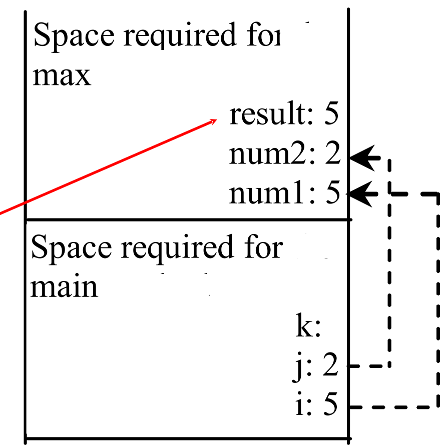
```
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace the Call Stack

```c
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

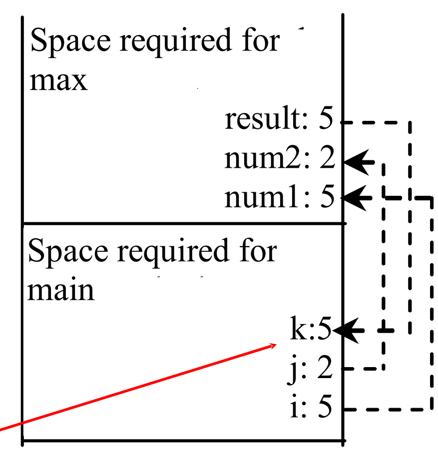
Assign num1 to result



Trace the Call Stack

```
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

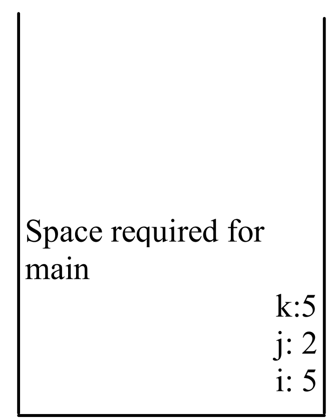
Return a **copy** of result and assign



+ Trace the Call Stack

Execute print statement

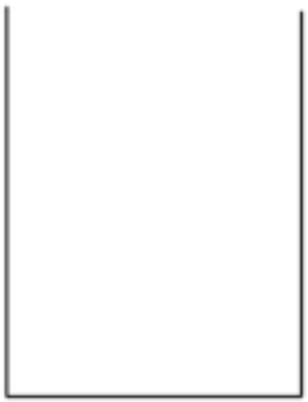
```
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



+ Trace the Call Stack

Complete the main function

```c
int max(int num1, int num2);

int main() {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  printf("The max is %d", k);
}

int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Stack in summary

- The stack grows and shrinks as functions push and pop local variables.
- Stack variables only exist while the function that created them is running
- There is no need to manage the memory yourself, variables are allocated and freed automatically.
- The stack has size limits.
- A common bug in C is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function after the declaring function has exited.

Heap Region

+ Heap

- For static and stack variables, the size of the allocation must be compile-time constant (except in C99, which allowed variable-length automatic arrays)
- The heap region gives us more freedom on how to utilize memory.
- Why?
  - Lifetime of data may be longer than a function call but shorter than the lifetime of the program.
  - Size of data may not be known in advance
    - e.g. May depend on result of calculation
  - Size may change over time
    - e.g., Increase canvas size or number of pages

Heap *con't*

- Unlike the stack, the heap does not have size restrictions on variable size (apart from the physical limitations of your computer).
- To allocate memory on the heap, you must use **malloc**() or **calloc**(), which are built-in C functions.
- Once you have allocated memory on the heap, you are responsible for using **free**() to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a ***memory leak***.

Stack Vs Heap

- **Stack**
  - Fast access
  - Don't have to explicitly de-allocate
  - Space is managed efficiently, memory will not become fragmented
  - Local variables only
  - Limit on stack size (OS-dependent)
  - Variables cannot be resized

- **Heap**
  - Variables can be accessed globally
  - No limit on memory size
  - Slightly slower access due to pointer dereferencing
  - No guaranteed efficient use of space, memory may become fragmented over time.
  - You must manage memory.
  - Variables can be resized using realloc()

* More here http://www.inf.udec.cl/~leo/teoX.pdf

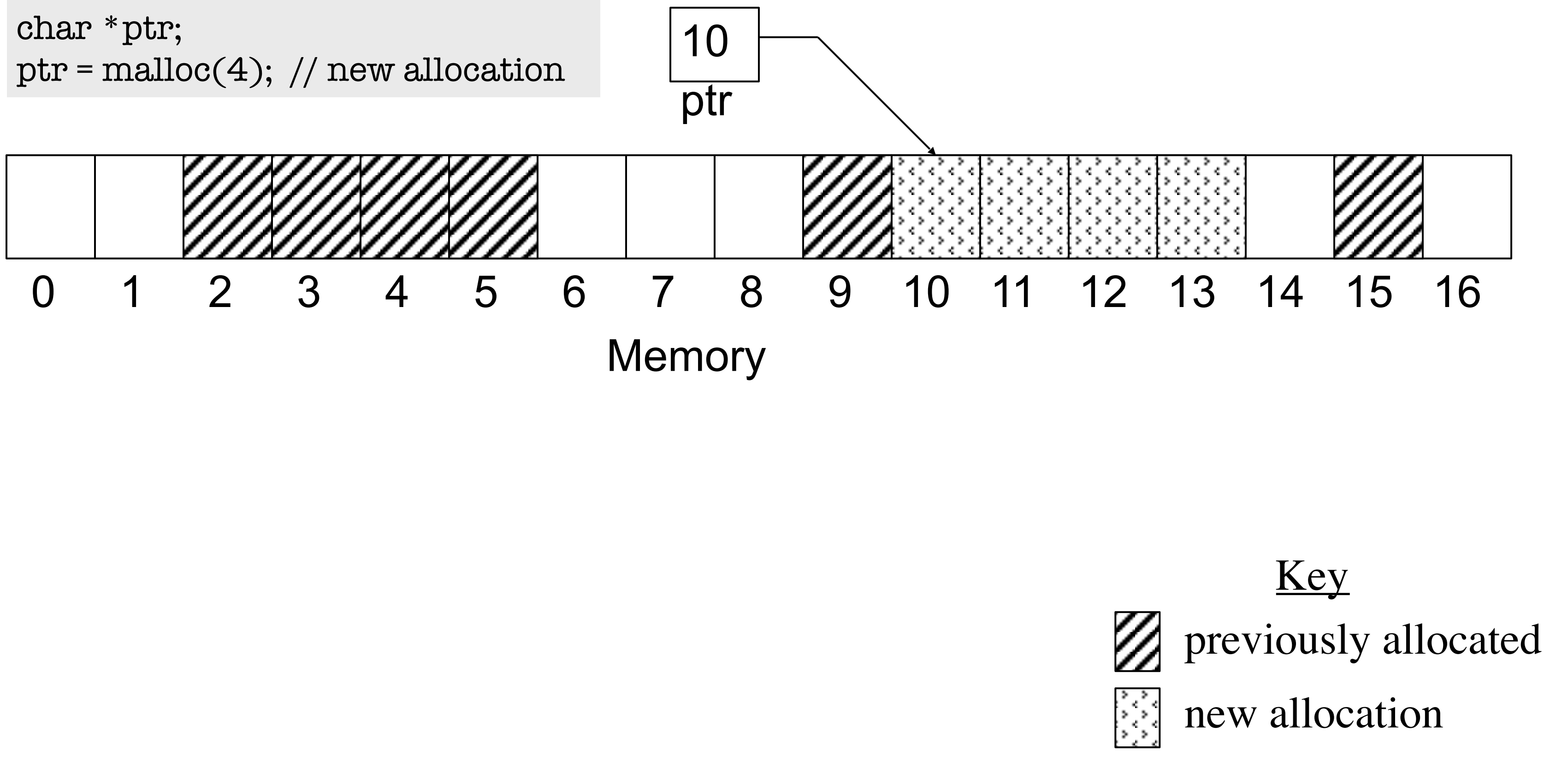
Dynamic Memory Allocation

+ malloc()

- The malloc() function is used for allocating heap memory at runtime.
- **void* malloc(int size_in_bytes);**
  - searches heap for ‘size’ contiguous free bytes.
  - returns the address of the first byte, unless no memory available then returns the null pointer.
  - programmers responsibility to not lose the pointer.
  - programmers responsibility to respect bounds.
- You must check to make sure that malloc was successful after each allocation!

malloc() example

```
char *ptr;
ptr = malloc(4);  // new allocation
```



C Vs Java

- **malloc()** is a bit like '**new**' in Java.
  - They both allocate space on the heap.
  - They both return the address to the location in the heap where the space requested was allocated.
- There is an important difference though, you do not need to 'clean-up' after yourself in Java.
- In C, you must deallocate memory heap-allocated memory explicitly.

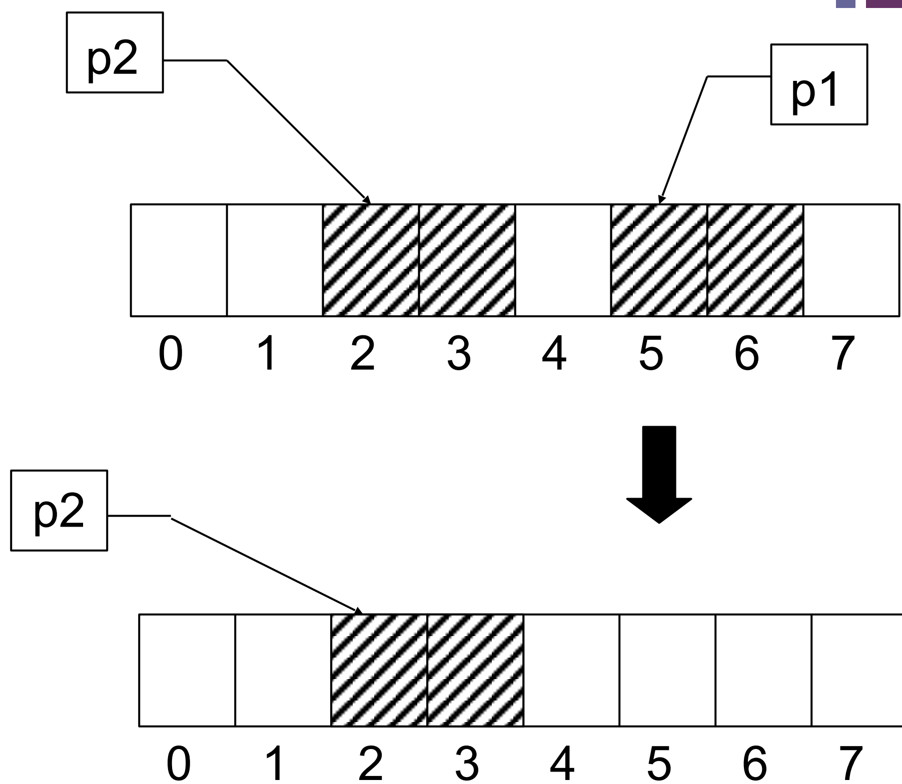
free()

- Any memory allocated with malloc() is reserved, in other words, it can't be used until it is deallocated with free().
- **void free(void* p);**
  - Releases the area pointed to by p.
  - ‘p’ must not be null.
  - System will know how much memory to deallocate.

free() example

```
char *p1;
p1 = malloc(2);
char *p2;
p1 = malloc(2);
```

```
free(p1);
```



Key

allocated memory

free allocation

sizeof()

- The sizeof() function is used to determine the size of any data type
- **int sizeof(type);**
  - returns how many bytes the data type needs
  - for example: sizeof(int) = 4, sizeof(char) = 1
  - works for standard data types and structs
    - after C99, works on variable-length arrays