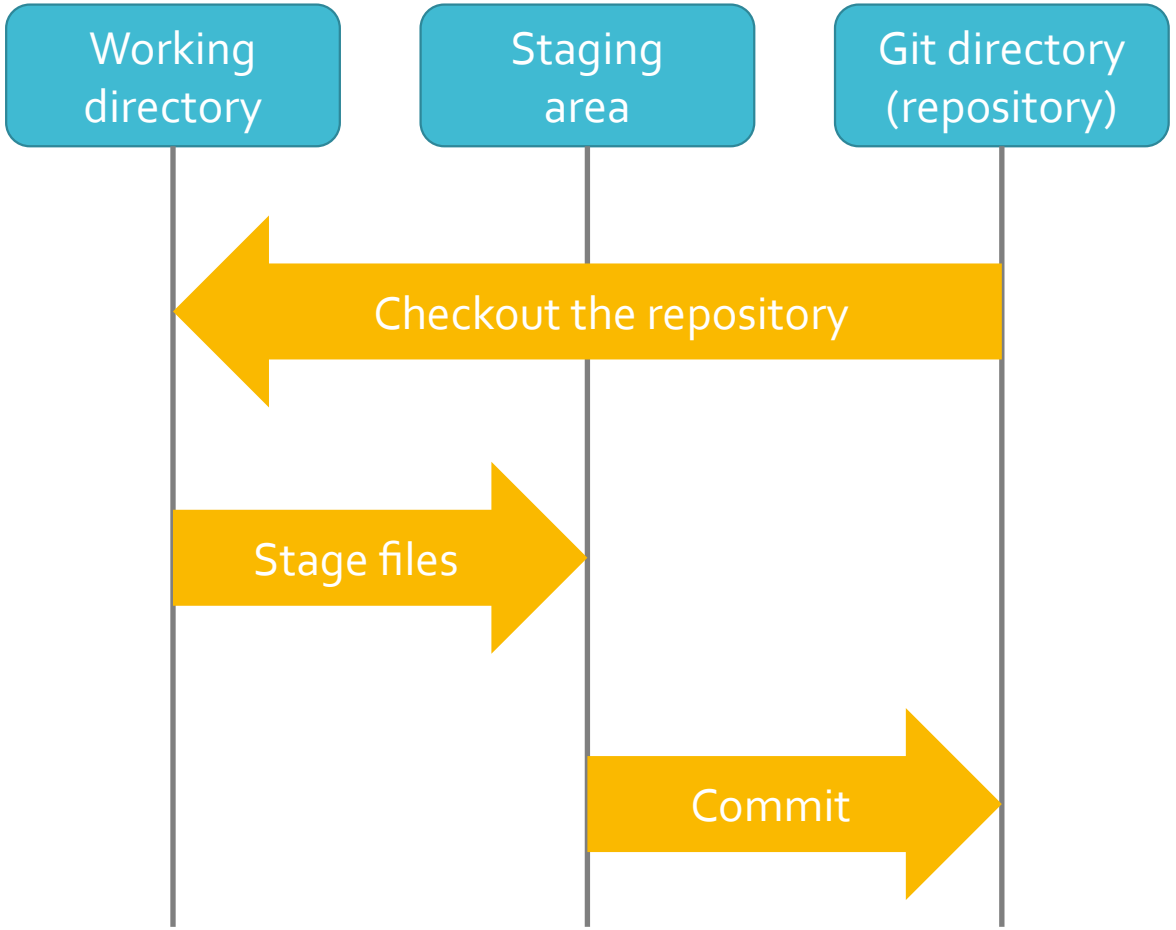
CIS 330
C++ and Unix

Lecture 3

Pointers

Last lecture - Git workflow



Example

```
> git init
Initialized empty Git repository in
/gpfs/projects/hpctensor/jeec/class/2019/S19/CIS330/.git/
> git add lecture02.c
> git commit -m "first lecture code"
> git push origin master
> git log
commit dfaa4e86109a427e32783b6ed72a4793b91f3d65
Author: Jee Choi <jeec@talapas-ln1.cm.cluster>
Date:   Fri Mar 15 18:34:16 2019 -0700

    first lecture code
```

Do not add binaries and object files to your repo - just the code and other files necessary to compiling your code

Do not add swap files (e.g., when you are using vim, it creates .swp files)

Always make sure to read all messages from git

- If you have a merge conflict, fix it and make sure the code compiles before you submit your assignment

Always keep committing your code at regular intervals and push it to Bitbucket

- If something goes wrong with your repo, you can always clone it and start from the last "save point."

Last Lecture - Control Flow

- if
- switch
- Repetition
  - while
  - do
  - for
- Branching
  - break
  - continue
  - goto

Last Lecture - Ternary Operators

1. a ? b : c -> if a is true then the expression evaluates to b, otherwise to c
2. (i == j) **?** printf(“A\n”) **:** printf(“B\n”);

Last Lecture - Macros

- #define
- #define PI 3.14 /* Define a constant value */
- #define calcCircleArea(r) (3.14 * (r) * (r)) /* Calculate area */
- Macros "calls" are replaced in the source code before compilation (by the C preprocessor)
- Similarly for #include and header files
  - May include other #define and function declarations

Compound data structures

```
1.  struct human_struct {
2.      char name[200];
3.      int age;
4.  };
5.  struct human_struct human_a;
```

- Data type with multiple entries – sometimes it make sense to associate multiple data types to a particular type of entity

Accessing individual components of a struct

```
1. strcpy(human_a.name, "The Doctor");
2. human_a.age = 1000;

3. struct human_struct* human_b = &human_a;
4. human_b->age = 2000;
```

Copying Struct

- Makes a "shallow" copy – individual components are copied one at a time
- What happens if you have pointers?

Union

```
1.  typedef union number {
2.      int i;
3.      float f;
4.      char *str;
5.  } number_t;
```

- Data type that allows a variable to take different types of values
- The members all reside in the same memory location – **not** stored redundantly

Questions?

Memory management

- In C, you have to manage your own memory
  - Accessing data is important in real applications – controlling it allows you to do "better," thereby allowing you to achieve better **performance**
- Examples
  - Arrange the data in such a way that it increases locality (spatial and temporal)
  - Allocating memory can be expensive – repeatedly allocating/deallocating small amounts of memory can degrade performance
- **Stack**
  - Short life span
  - e.g., live during the invocation of a function
- **Heap**
  - Long life span
  - e.g., lasts until you "delete" them (or the program ends)

Pointers

- "Points" to a location in memory (i.e., address)
- Denoted with * (e.g., int* a; => a pointer to an integer)
- You can "get" the address of a variable using &

```
1. int x;
2. int* y;
3. y = &x;
```

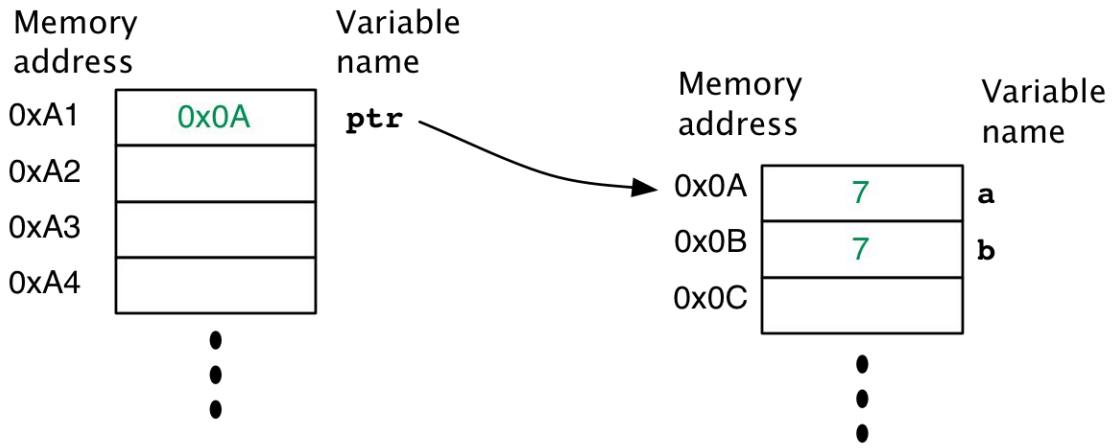
- You can "dereference" an address and access the value in that location using * or [ ]

```
4. fprintf(stdout, "%d\n", *y);
```

  - Or

```
5. fprintf(stdout, "%d\n", y[0]);
```

Pointers




```
int main() {
  int *ptr; // pointer to integer value
  int a, b; // integer variables

  a = 7;    // assign value to a
  ptr = &a; // assign address of a (0x0A in this example) to ptr
  b = *ptr; // assign value at the address contained in ptr to b

  // Result: a and be contain the same value (7) and
  //         ptr contains the address of a.
  return 0;
}
```

Memory address

- Represented by hexadecimal numbers

```
int* A = 0x8000;
```

Pointer arithmetic - arithmetic operations on pointers at done at data size granularity

```
A++; /* A+1 == 0x8004 since int is 4 Bytes */
```

```
double* A = 0x6000;
A++; /* A+1 == 0x6008 since double is 8 Bytes */
```

```
double* A = 0x6000;
A++; /* A+2 == ? */
```

Memory address

- Represented by hexadecimal numbers

```
int* A = 0x8000;
```

Pointer arithmetic - arithmetic operations on pointers at done at data size granularity

```
A++; /* A+1 == 0x8004 since int is 4 Bytes */
```

```
double* A = 0x6000;
A++; /* A+1 == 0x6008 since double is 8 Bytes */
```

```
double* A = 0x6000;
A++; /* A+2 == 0x6010 Hexadecimal is represented by 4 bits */
```

Pointer arithmetic

- Dereferencing a pointer
- int X = 10;
- int* A = &X;
- printf("%d\n", *A); /* this prints 10 */

- Another way to dereference memory – []
- A[0] <-> *A
- A[5] <-> *(A + 5)
- B = &(A[0]) <-> B = A
- B = &(A[5]) <-> B = A + 5

How do you "get" memory?

- malloc (memory allocate)
- free (frees up memory)

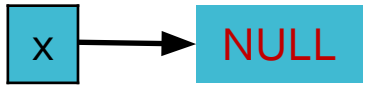
Examples of malloc and free

```
1. int* y = NULL;
2. y = malloc(sizeof(int) * 1);  /* 4 Bytes */
3. *y = 7;
4. fprintf(stdout, "y is %d\n", *y);
5. free(y); /* OS already knows # Bytes */
```

Examples of malloc and free

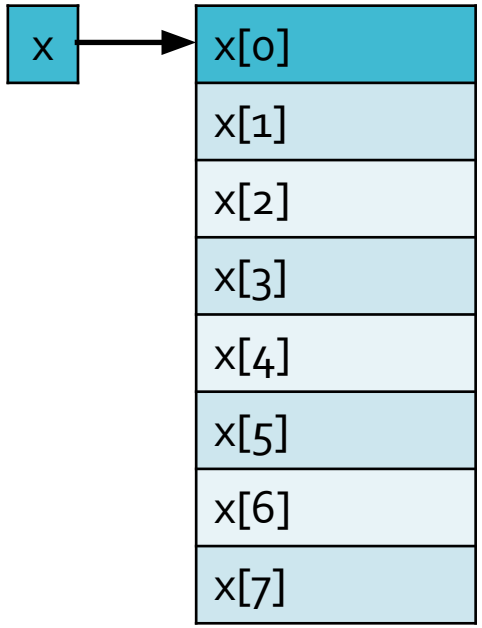
```
1. int* y = NULL;
2. y = malloc(sizeof(int) * 1);  /* 4 Bytes */
3. if(y == NULL) { exit(0); }
4. *y = 7;
5. fprintf(stdout, "y is %d\n", *y);
6. free(y); /* OS already knows # Bytes */
```

Pointers to pointer (to pointer…)



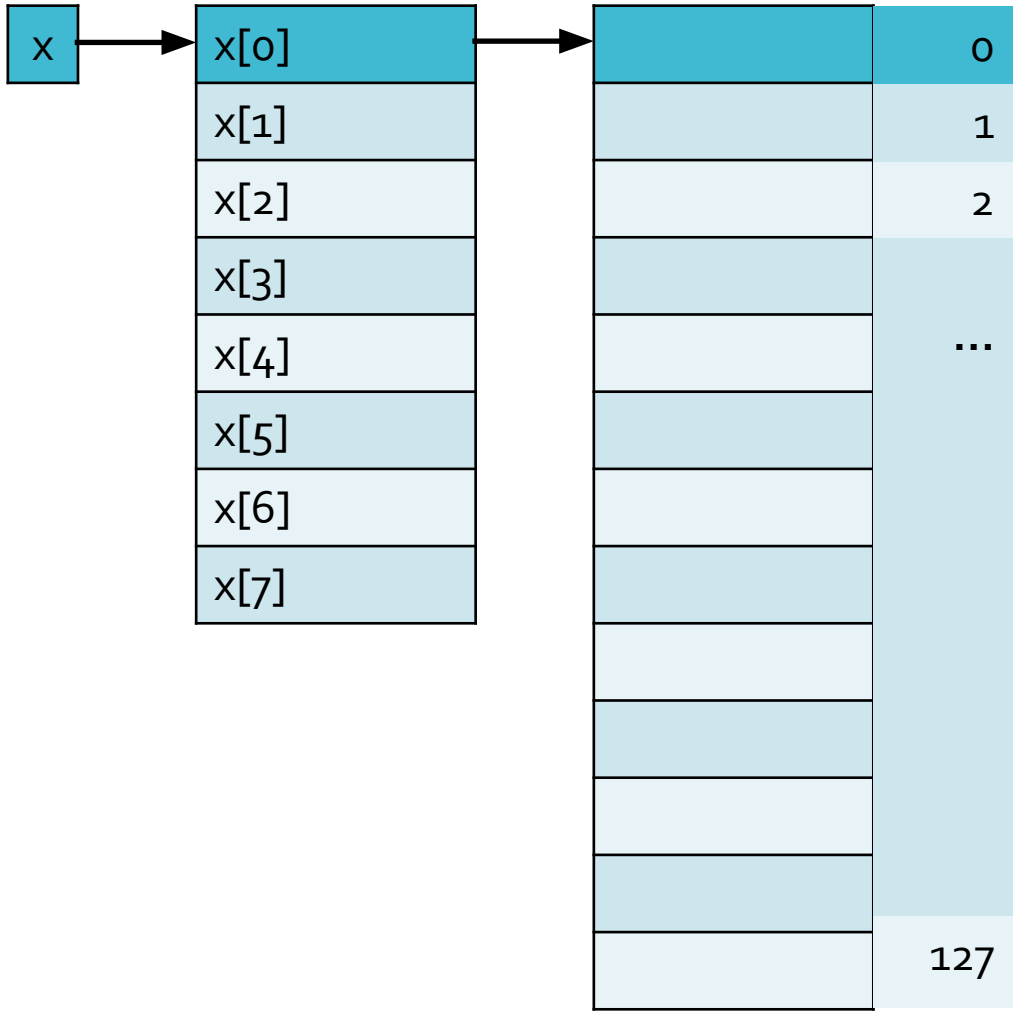

```
int** x = NULL;
```

Pointers to pointer (to pointer...)




```
x = malloc(sizeof(int*) * 8);
```

Pointers to pointer (to pointer…)




```
x[0] = malloc(sizeof(int) * 128);
```

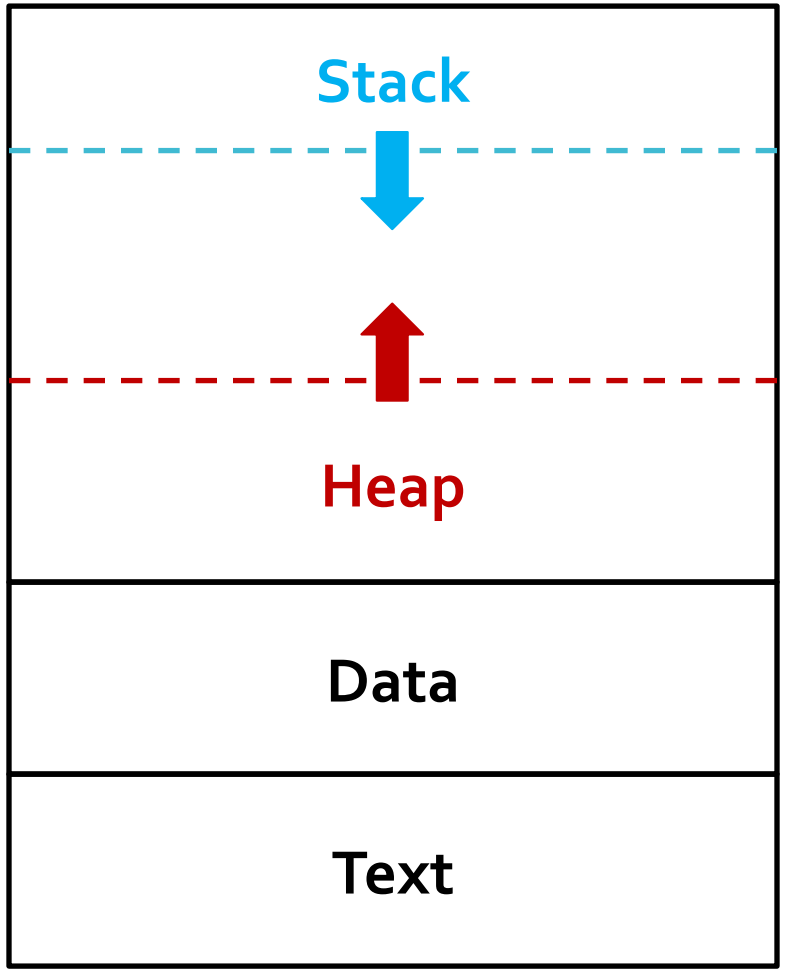
Pointers to pointer (to pointer…)

- Remember – pointers store memory addresses
  - Both "pointer to int" and "pointer to pointer to int" store memory addresses
  - Same goes for any "pointer to X"

Memory Segments

- In C, a program is stored in memory as "segments"
  - **Text** segment (code)
  - **Data** segment
  - **Stack** segment
  - **Heap** segment

Memory Segments



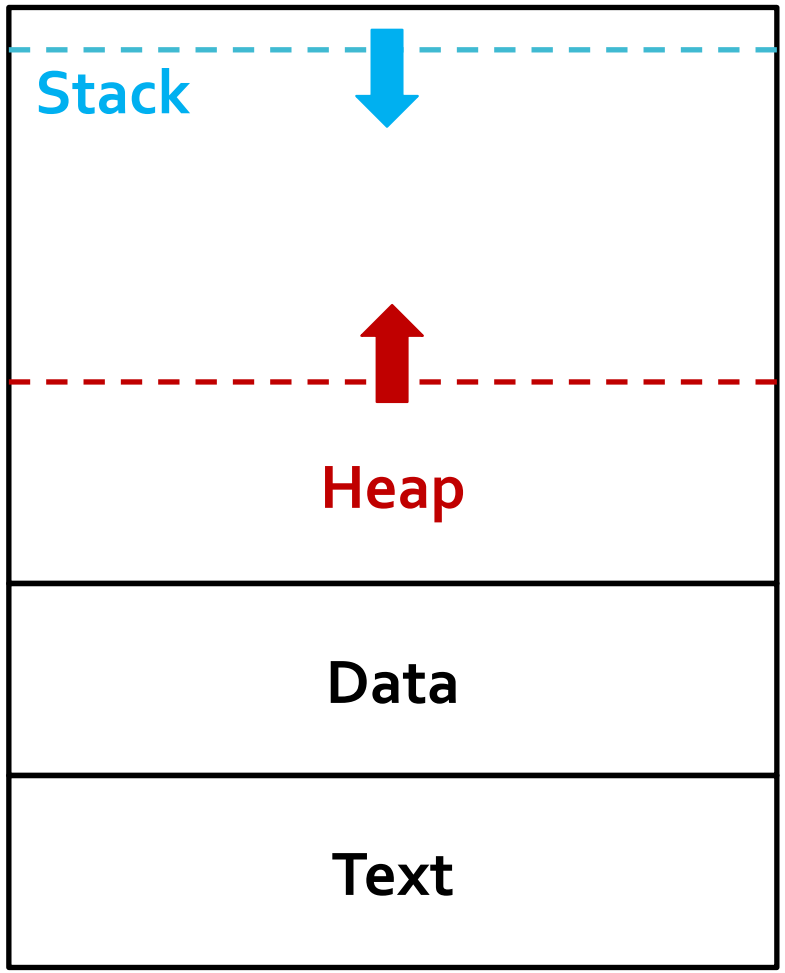
What happens when you use too much memory?

- Overflow occurs
- Unexpected/unpredictable behavior
- Stack overflow attack – intentionally create an overflow to overwrite the function's return address and execute user supplied data (e.g., some command shell instructions)
- Always `free()` your memory!
  - Fragmentation

Variable scope and types

- Block scope – scope within {}
  - Can be nested
- Function scope
  - Variables declared within a function
- Program scope
  - Global variables
  - Variables declared within the main function
- static variables
  - Variable lives outside its scope
- const variable
  - Value cannot be changed after initialization
- volatile variable
  - Tell the compiler not to optimize this variable
- extern variable
  - Extend scope across files
- register variable
  - Use certain registers to hold the value – just a suggestion, not guaranteed to work every time
  - Do not try to use it with & (address) – it may not have one

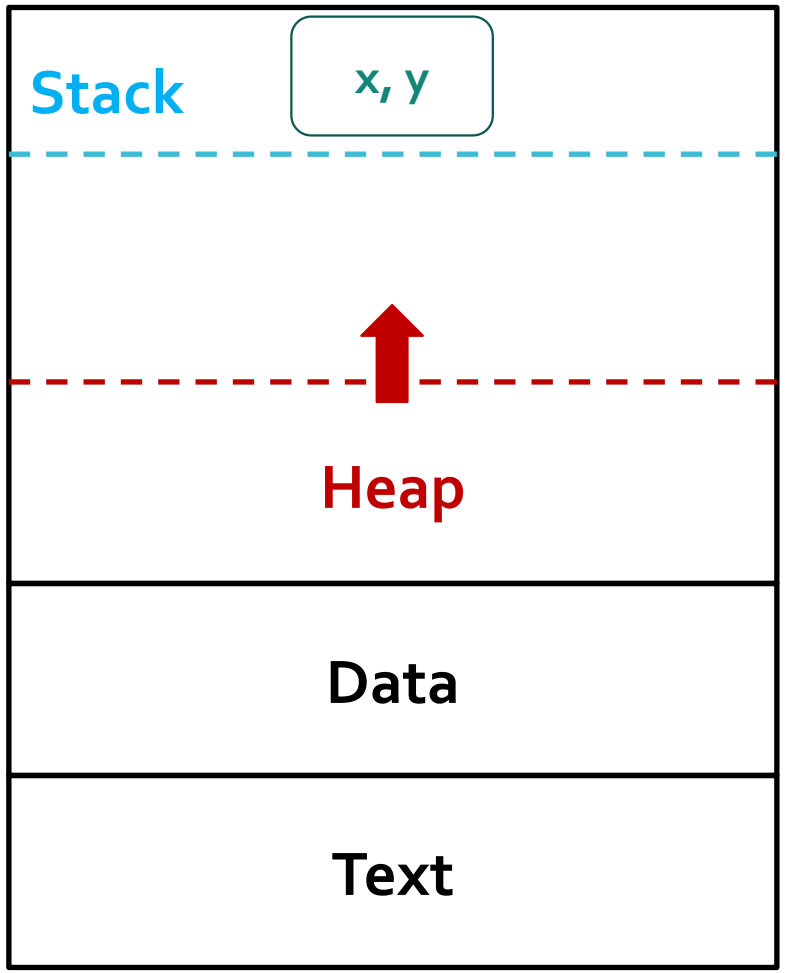
Memory Segments




```
1.  int main()
2.  {
3.      int x = 7;
4.      int y = 12;
5.      int z = return_add_numbers(x, y);
6.  }
```

```
1.   int return_add_numbers(int a, int b)
2.   {
3.       return (a + b);
4.   }
```

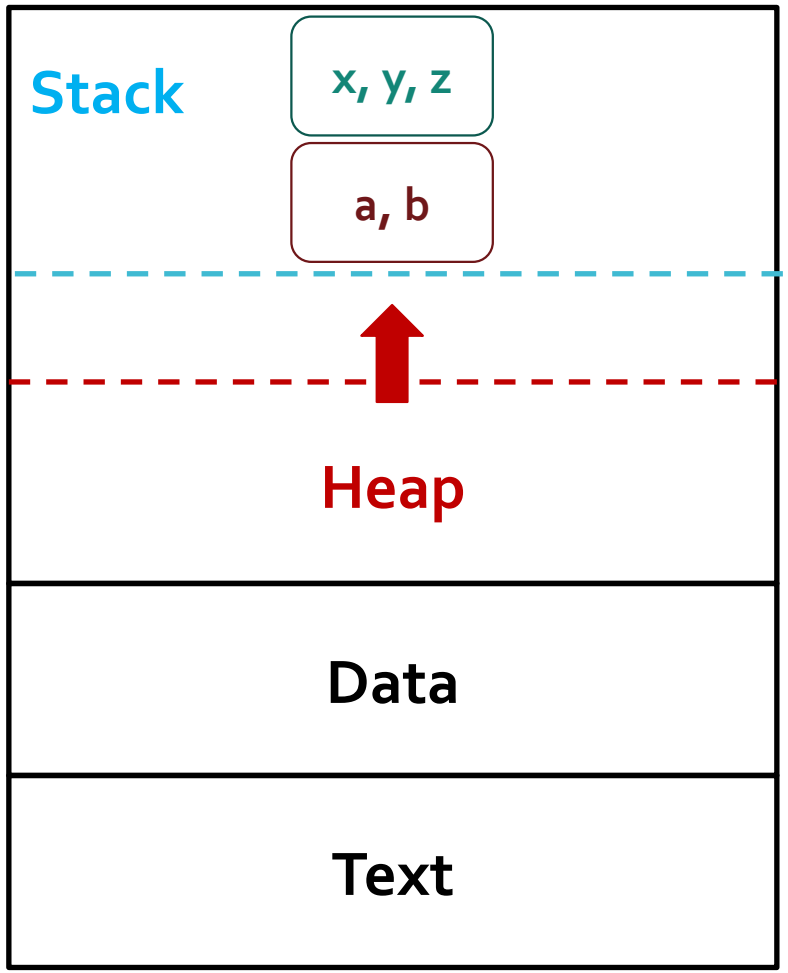
Memory Segments




```
1.  int main()
2.  {
3.      int x = 7;
4.      int y = 12;
5.      int z = return_add_numbers(x, y);
6.  }

1.    int return_add_numbers(int a, int b)
2.    {
3.        return (a + b);
4.    }
```

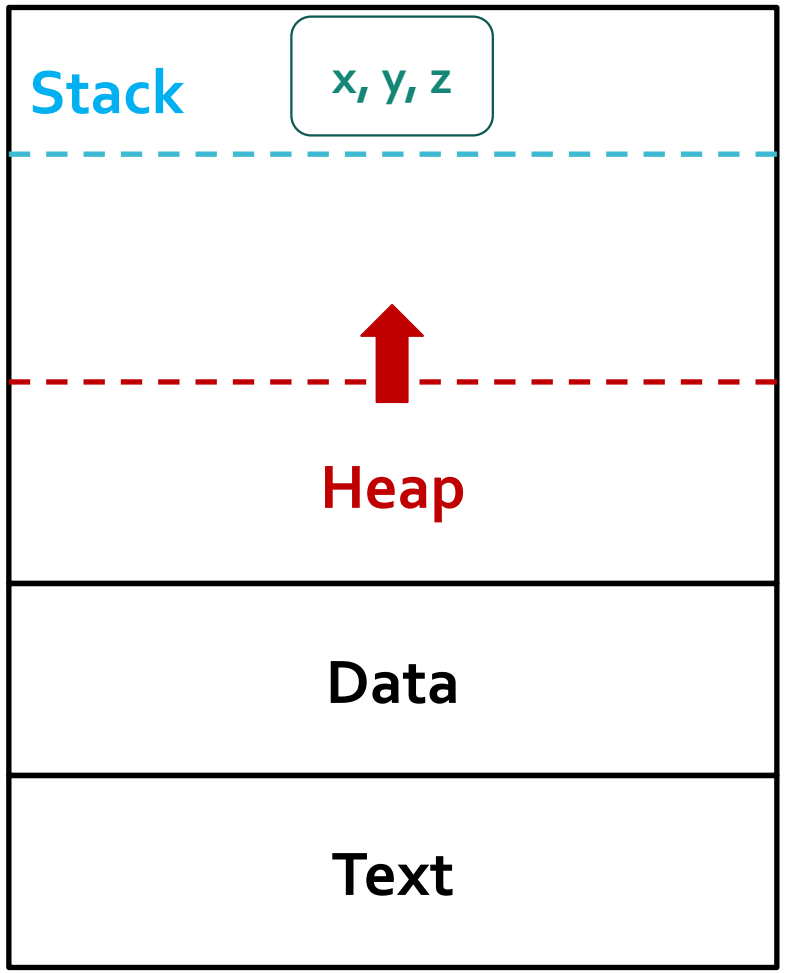
Memory Segments




```
1.  int main()
2.  {
3.      int x = 7;
4.      int y = 12;
5.      int z = return_add_numbers(x, y);
6.  }

1.    int return_add_numbers(int a, int b)
2.    {
3.        return (a + b);
4.    }
```

Memory Segments

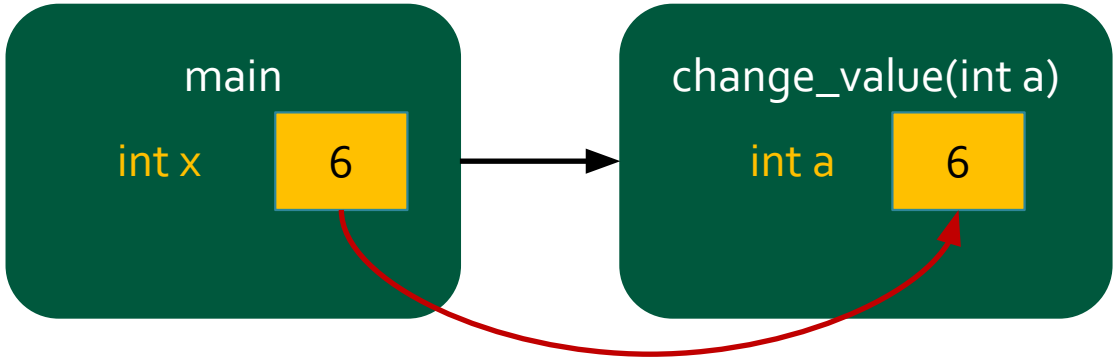



```
1.  int main()
2.  {
3.      int x = 7;
4.      int y = 12;
5.      int z = return_add_numbers(x, y);
6.  }
```

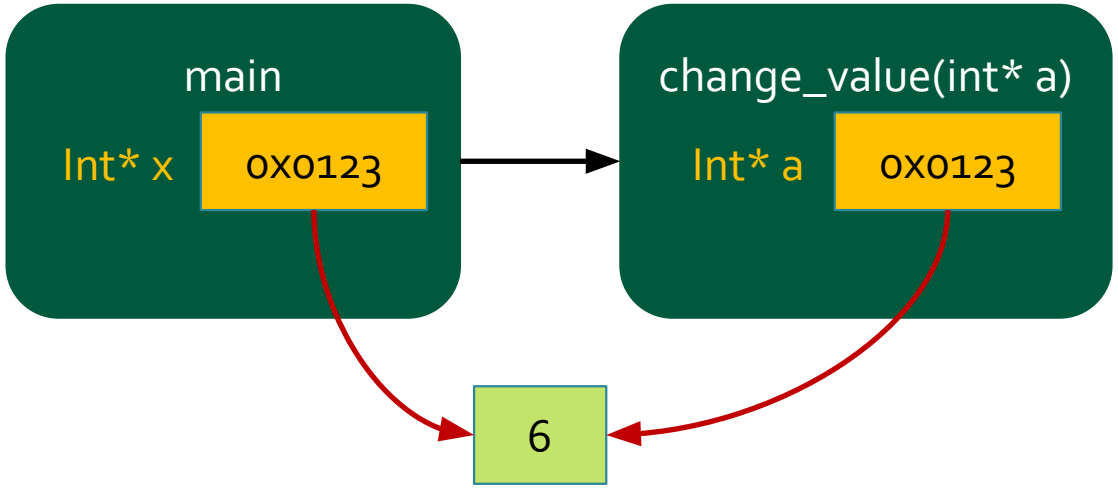
```
1.  int return_add_numbers(int a, int b)
2.  {
3.      return (a + b);
4.  }
```

Pass by Value/ Reference

- Pass by value



- Pass by reference



Pass by Value/ Reference

- Technically, everything is pass by value in C (the pointer values are being copied), but the idea of pass by reference is achieved via pointers

Function Pointers

- Yes, you can have a pointer to a function too
- Useful when you want a certain function to be called when a predetermined event occurs

Function Pointer Example

```
1.  double (*funcptr) (double x[], int cnt);
2.  funcptr = &computeAverage;
3.  double x[] = {1.0, 2.0, 3.0, 4.0};
4.  int cnt = 4;
5.  double avg = funcptr(x, cnt);
```

where,

```
1.  double computeAverage(double y[], int cnt) {
2.      double run = 0.0;
3.      for(int i = 0; i < cnt; i++) {
4.          run += y[i];
5.      }
6.      run = run / cnt;
7.      return run;
8.  }
```

What are arrays?

- Arrays
  - **Collection** of data items of **same type**
  - Think “vectors” in linear algebra, or a simply a “list”
  - It can be multidimensional

Declaration

```
1. int array_a[ARR_SIZE];
2. int array_b[ARR_SIZE][ARR_SIZE];
3. int* array_c = NULL;
```

Initialization

```
1. int array_a[ARR_SIZE] = {100, 200, 300, 400, 500};
2. int array_b[] = {10, 20, 30};
3. int array_c[3] = {1, 2, 3, 4, 5}; /* ILLEGAL */
4. int array_d[n]; /* n is a variable */
```

N-D Arrays

```
1.  int array_a[ARR_SIZE][ARR_SIZE] = {{1,2}, {3,4}};
2.  int array_b[ARR_SIZE][ARR_SIZE][ARR_SIZE] =
3.      {
4.          { /* array_b[0][][] */
5.              {1, 2}, /* array_b[0][0][] */
6.              {3, 4}  /* array_b[0][1][] */
7.          },
8.          {/* array_b[1][][] */
9.              {5, 6}, /* array_b[1][0][] */
10.             {7, 8}  /* array_b[1][1][] */
11.         }
12.     };
```

And so on…

Pointer vs. [] declaration

- Pointers allow more flexibility in size and shape
- You can always access the pointer array using [] to deference
- **Use pointers whenever you can (please)**
- Except when you need a **small** n-D array for storing constant values (could be **faster** to access)

Pointer vs. [] declaration

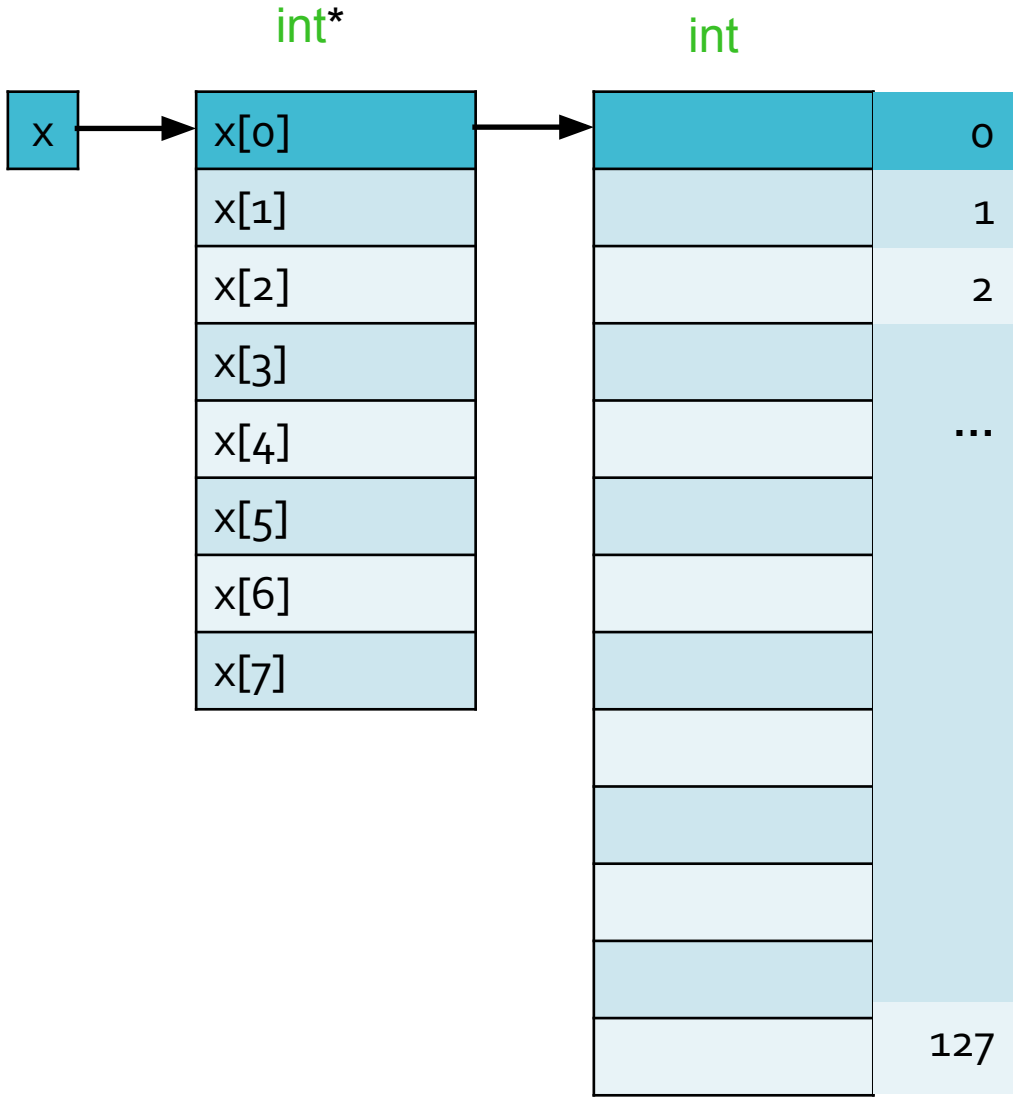
- Local arrays (e.g., int arr[100];) are created at compile time (i.e., they have a fixed size), created on the stack (i.e., managed automatically - no need to free())
- Because they have a fixed size, sizeof() will return a different value

```
int arr[100];
int* arrptr;

printf("%lu\n", sizeof(arr));
printf("%lu\n", sizeof(arrptr));
```

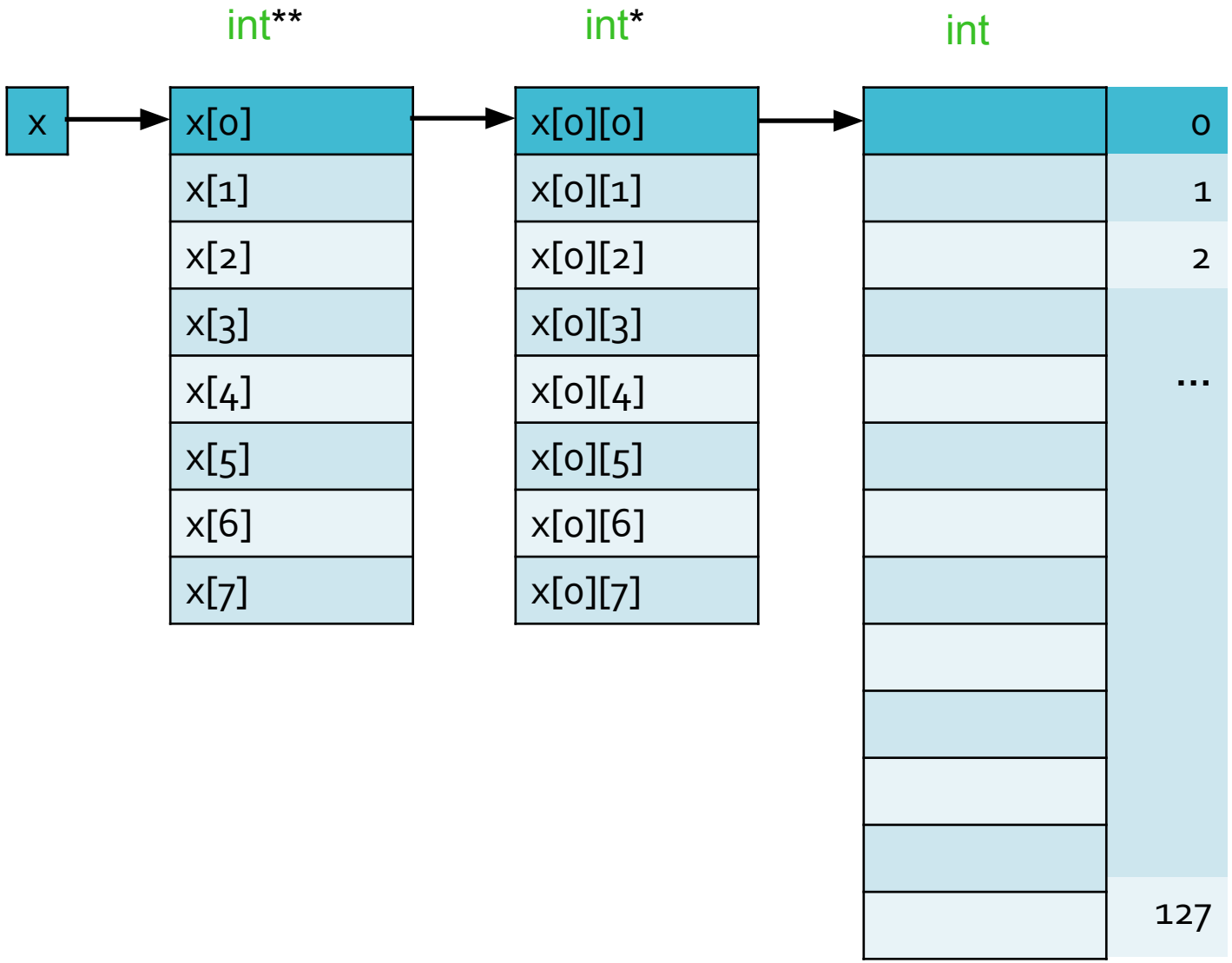
What will be printed?

2D Arrays




```
int** x = NULL;
```

3D Arrays



int*** x = NULL;

Allocating a Pointer Array

- 2D Array

```
1.  d2_array = malloc(sizeof(int*) * ARR_SIZE);
2.  for(int i = 0; i < ARR_SIZE; i++) {
3.      d2_array[i] = malloc(sizeof(int) * ARR_SIZE);
4.  }
```

Allocating a Pointer Array

- 3D Array

```
1.  d3_array = malloc(sizeof(int**) * ARR_SIZE);
2.  for(int i = 0; i < ARR_SIZE; i++) {
3.      d3_array[i] = malloc(sizeof(int*) * ARR_SIZE);
4.      for(int j = 0; j < ARR_SIZE; j++) {
5.          d3_array[i][j] = malloc(sizeof(int) * ARR_SIZE);
6.      }
7.  }
```

Freeing a Pointer Array

```
1. for(int i = 0; i < ARR_SIZE; i++) {
2.     for(int j = 0; j < ARR_SIZE; j++) {
3.         free(d3_array[i][j]);
4.     }
5.     free(d3_array[i]);
6. }
7. free(d3_array);
```

Strings

- Basically an array of char (or a char*)
- Terminated with a null character
- String is a data structure that uses array of char to implement
- Declaration

```
char array_str[STR_SIZE];
char* array_str = NULL; /* malloc */
```

- Initialization

```
char array_str[STR_SIZE] = "Hello World\n";
char* array_str = "Hello World\n";
```

String Functions

- char *strcpy(str1, str2)
  - Copy str2 to str1
- char *strcat(str2, str1)
  - Concatenate str1 to str2 and returns str2
  - Make sure there is enough space in str2 to append str1 to it
- int strcmp(str1, str2)
  - Compare str1 to str2 and return a value less than zero if str1 is lexicographically less than the second
- size_t strlen(str1)
  - Length of str1, not including the null character
- char *strchr(str1, c)
  - Find the first occurrence of c (int converted from char) in str1, and returns the location of the found character
- strncat
- strncmp
- strncpy
- strrchr

n-byte variation of the original functions

Static vs. dynamic allocation

```
int main()
{
  int array1[10]; // static allocation
  int* array2 = (int*) malloc(sizeof(int) * 10); // dynamic allocation
}
```

Static array

Size is determined at compile-time

Created on the stack memory (less memory space)

- Stack is a “temporary” space so limited capacity is by-design
- Stack memory is contiguous, prevents errors (e.g., infinite loop) from going too far

Dynamic array

Created at runtime

Created on the heap - can be very large (until you run out of memory)

Any large data (i.e., arrays) should be created on the heap

Example

```
int array1[5][5];
int* array2 = (int*) malloc(sizeof(int) * 25);

printf("%lu\n", sizeof(array1));
printf("%lu\n", sizeof(array2));
```

What would be printed?

Example

```
int array1[5][5];
int* array2 = (int*) malloc(sizeof(int) * 25);

printf("%lu\n", sizeof(array1));
printf("%lu\n", sizeof(array2));
```

What would be printed?

100

8

Coding example