CS 31 Discussion

ABDULLAH-AL-ZUBAER IMRAN

WEEK 9: DYNAMIC MEMORY ALLOCATION

Recap

- Memory management
- Pointers
  - Pointer and Arrays
  - Pointer Arithmetic
  - Pointer to Pointer
  - Reference Pointer

Discussion Objectives

Review and practice things covered during lectures

- Dynamic Memory Allocation
- Memory Leak
- Class Destructor

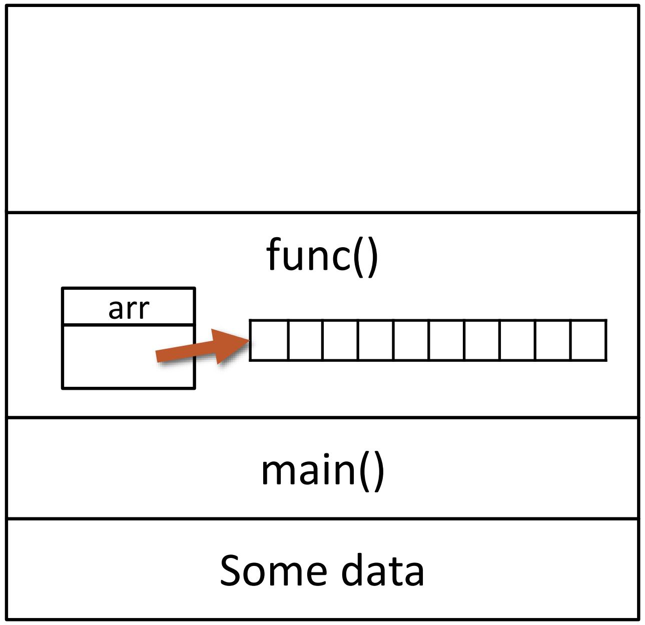
- Coding examples

Programming Challenge

Time for you to ask questions!

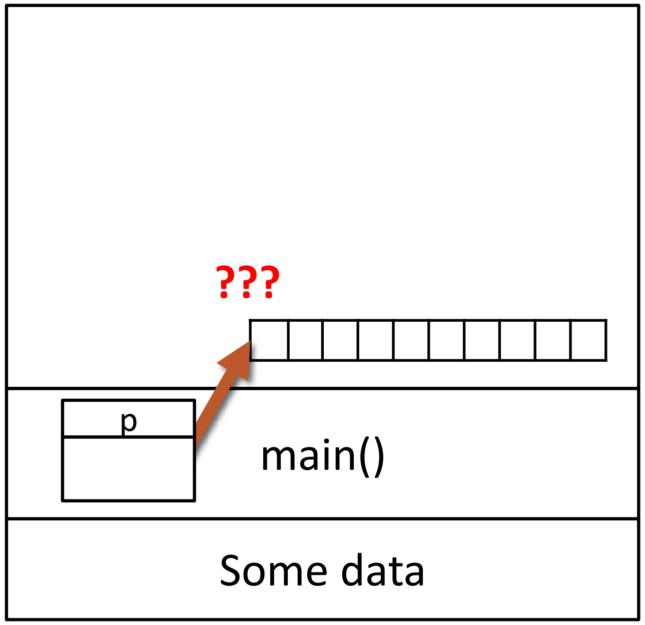
Dynamic Allocation of Memory

```
int *func();
int main()
{
 int *p = func();
 return 0;
}
int *func() {
 int arr[10];
 return arr;
}
```



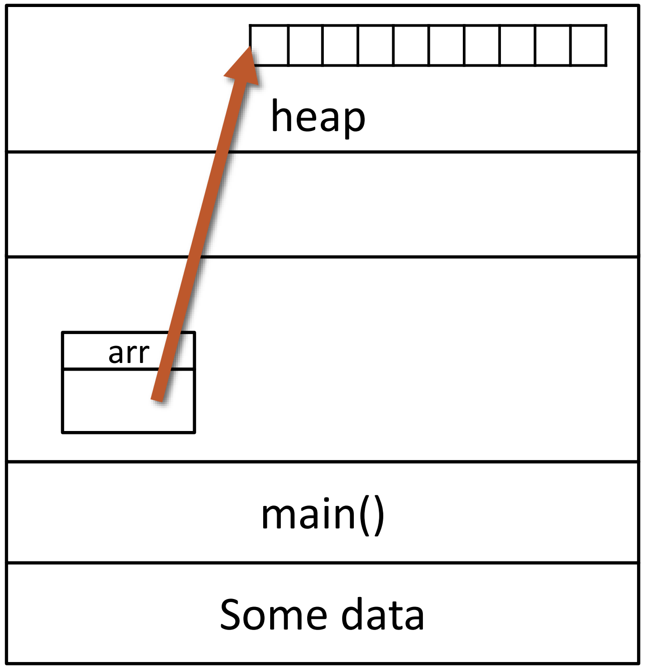
Dynamic Allocation of Memory

```
int *func();
int main()
{
 int *p = func();
 return 0;
}
int *func() {
 int arr[10];
 return arr;
}
```



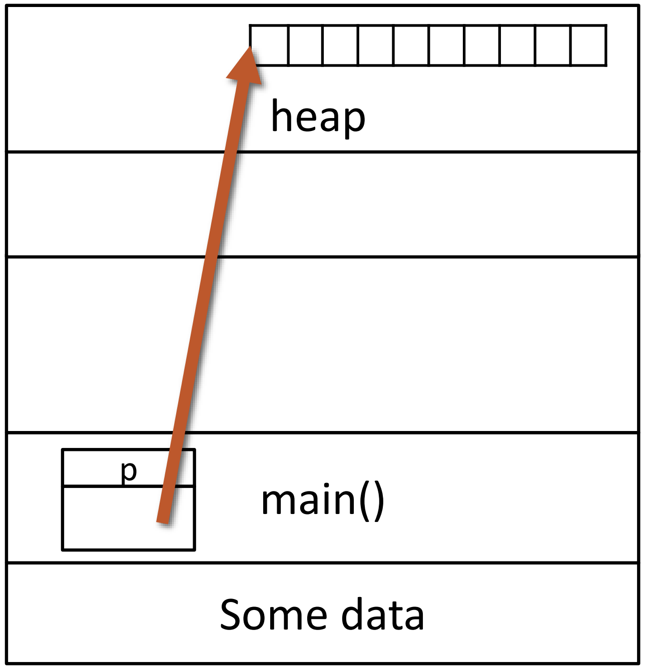
Dynamic Allocation of Memory

```
int main() {
 int *p = func();
 delete[] p;
 return 0;
}
int* func() {
 int* arr = new int[10];
 return arr;
}
```



Dynamic Allocation of Memory

```
int main() {
 int *p = func();
 delete[] p;
 return 0;
}
int* func() {
 int* arr = new int[10];
 return arr;
}
```



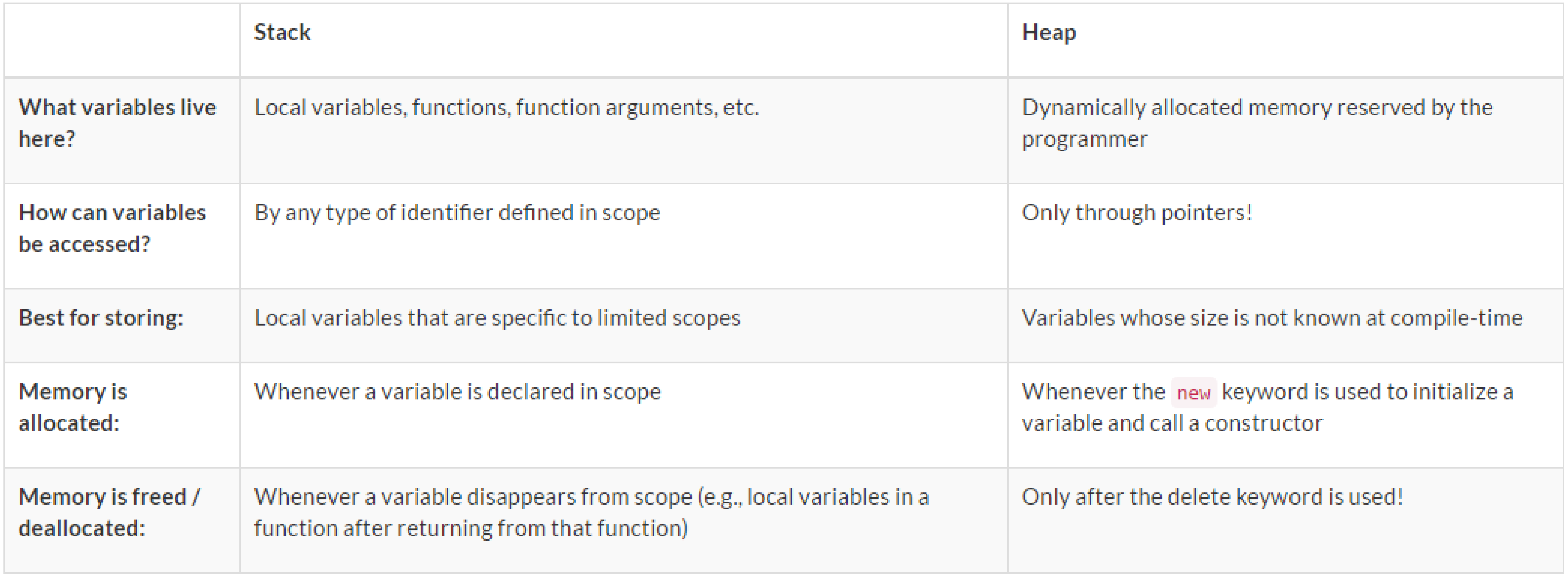
Stack vs. Heap

Stack

- Local variables, functions, function arguments, etc.
- Variables in the stack vanish when outside the scope.

- Heap
  - Dynamically allocated memory reserved by the programmer
  - Variables in the heap remain until you use delete to explicitly destroy them.

Stack vs. Heap



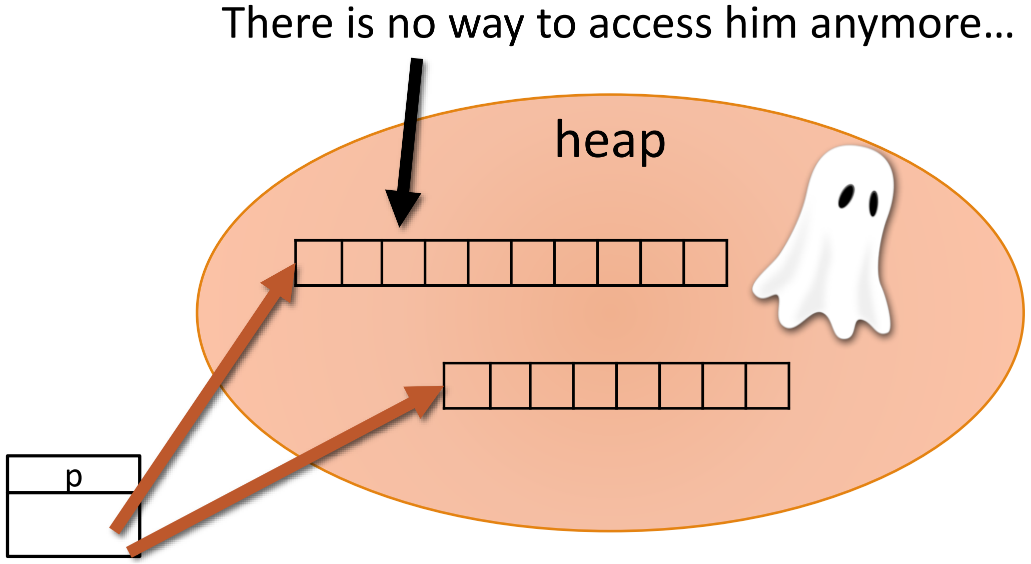
	Stack	Heap
What variables live here?	Local variables, functions, function arguments, etc.	Dynamically allocated memory reserved by the programmer
How can variables be accessed?	By any type of identifier defined in scope	Only through pointers!
Best for storing:	Local variables that are specific to limited scopes	Variables whose size is not known at compile-time
Memory is allocated:	Whenever a variable is declared in scope	Whenever the `new` keyword is used to initialize a variable and call a constructor
Memory is freed / deallocated:	Whenever a variable disappears from scope (e.g., local variables in a function after returning from that function)	Only after the delete keyword is used!

Programmers need to deallocate the memory by themselves!

Memory Leak

What happens here?

```
int *p;
p = new int[10];
p = new int[8];
```



A simple rule to remember:

For each new statement, there should be one delete statement.

Memory Leak

```
int *p = new int;                            delete p;

int *p = new int[2];                         delete[] p;

int *pArr[10];

for (int i = 0; i < 10; i++)                 delete[] pArr; ✗
  pArr[i] = new int;
```

Memory Leak

```
int *p = new int;                       delete p;

int *p = new int[2];                    delete[] p;

int *pArr[10];

for (int i = 0; i < 10; i++)   for (int i = 0; i < 10; i++)
  pArr[i] = new int;             delete pArr[i];
```

Class - Constructors

Dynamic allocation

```
 Cat *pKitty = new Cat();
 Cat *pKitty2 = new Cat(10);
 pKitty->meow()
(*pKitty).meow()
```

Class - Constructors

Can this compile? If so, what's the output?

```
#include<iostream>
using namespace std;
class Cat {
public:
  Cat(int initAge);
  int age();
  void setAge(int newAge);
private:
  int m_age;
};
Cat::Cat(int initAge) {
  setAge(initAge);
}
int Cat::age(){
  return m_age;
}
```

```
void Cat::setAge(int newAge){
  m_age = newAge;
}
class Person {
private:
  Cat pet;
};
int main(){
  Person Mary;
}
```

Class - Constructors

A fixed solution

```
#include<iostream>
using namespace std;
class Cat {
public:
  Cat(int initAge);
  int age();
  void setAge(int newAge);
private:
  int m_age;
};
Cat::Cat(int initAge) {
  setAge(initAge);
}
int Cat::age(){
  return m_age;
}
```

```
void Cat::setAge(int newAge){
  m_age = newAge;
}
class Person {
public:
  Person():pet(1){
    cout << "Person initialized" << endl;
  };
private:
  Cat pet;
};
int main(){
  Person Mary;
}
```

Class - Constructors

Order of construction

When we instantiate an object, we begin by initializing its member variables *then* by calling its constructor. (Destruction happens the other way round!)

The member variables are initialized by first consulting the initializer list. Otherwise, we use the default constructor for the member variable as a fallback.

For this reason, member variable without a default constructor <u>must</u> be initialized through the initializer list.

Class -Destructors

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

- The destructor should use `delete` to eliminate any dynamically allocated variables created by the object
- A destructor's name starts with ~, followed by the name of the class, with no return type or arguments.

```
class Cat {
public:
  Cat(int initAge);
  ~Cat();
  int age();
  void setAge(int newAge);
private:
  int m_age;
};
```

Class: Destructor

```
class String

{

private:

   char *s;

   int size;

public:

   String(char *); // constructor

   ~String();     // destructor

};
```

```
String::String(char *c)
{
   size = strlen(c);
   s = new char[size+1];
   strcpy(s,c);
}

String::~String()
{
   delete []s;
}
```

Thanks!

Questions?

Complete Evaluation for the class

Today’s discussion slides can be found at

http://web.cs.ucla.edu/~aimran/winter19_cs31_w8.pdf

Next Discussion: Final Review