

CS 31 Discussion

ABDULLAH-AL-ZUBAER IMRAN

WEEK 9: DYNAMIC MEMORY ALLOCATION

Discussion Objectives

Review and practice things covered during lectures

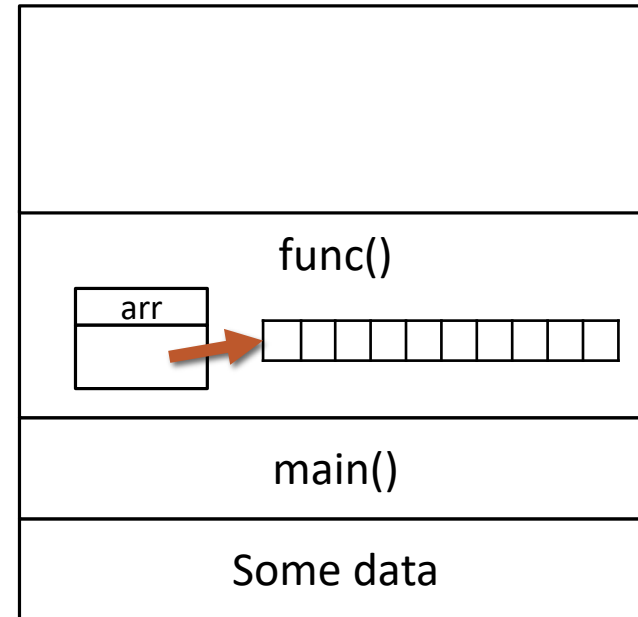
- Dynamic Memory Allocation
- Memory Leak
- Class Destructor

- Coding examples
- Project7

Time for you to ask questions!

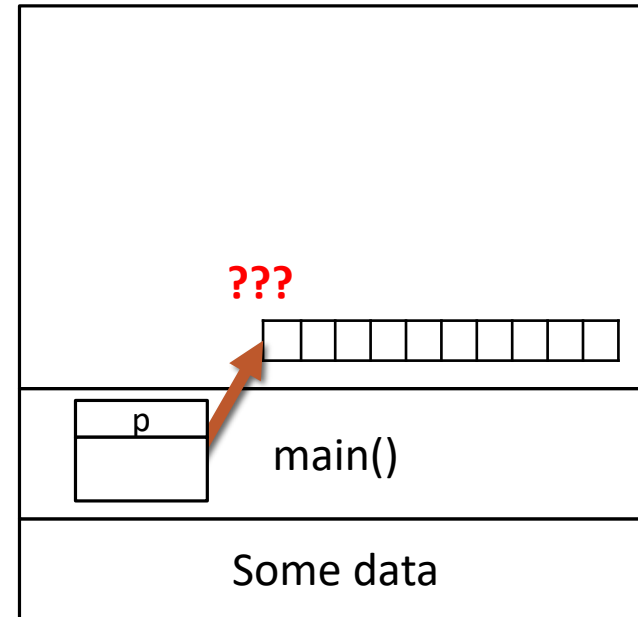
Dynamic Allocation of Memory

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



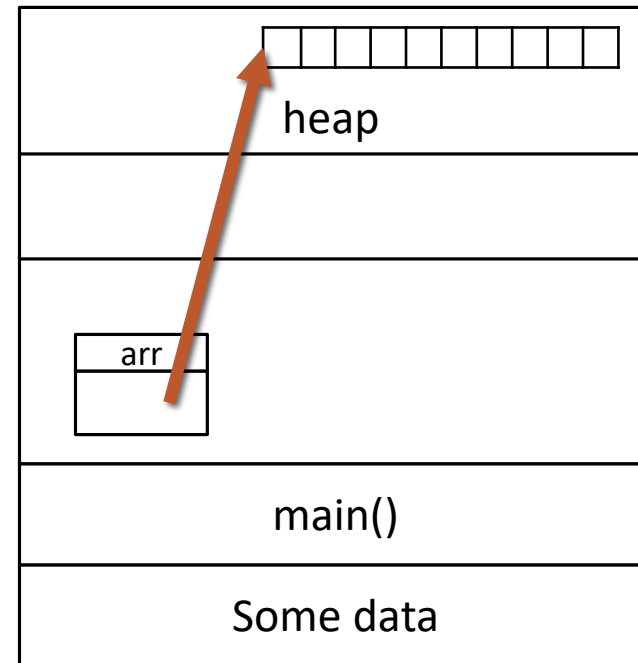
Dynamic Allocation of Memory

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



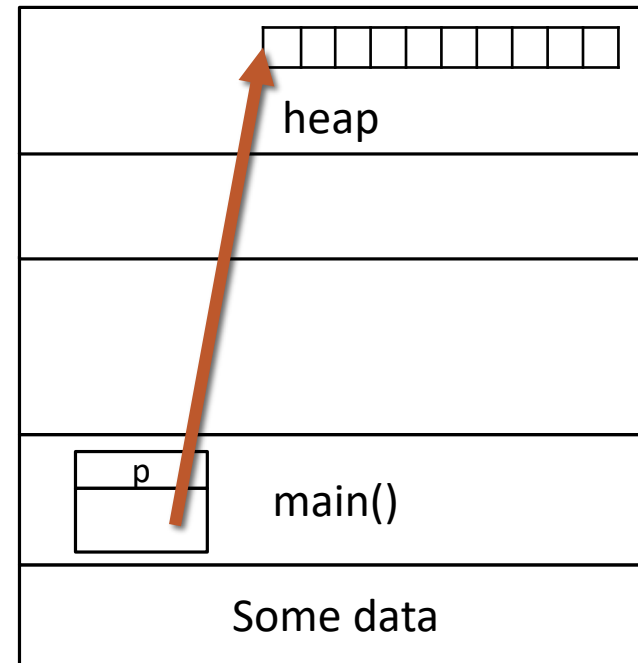
Dynamic Allocation of Memory

```
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

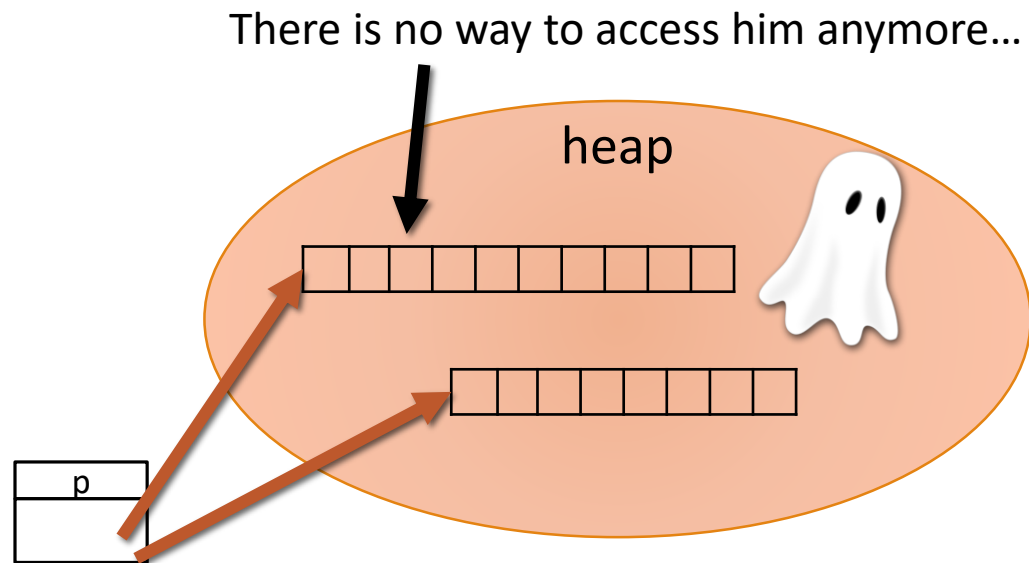
	Stack	Heap
What variables live here?	Local variables, functions, function arguments, etc.	Dynamically allocated memory reserved by the programmer
How can variables be accessed?	By any type of identifier defined in scope	Only through pointers!
Best for storing:	Local variables that are specific to limited scopes	Variables whose size is not known at compile-time
Memory is allocated:	Whenever a variable is declared in scope	Whenever the <code>new</code> 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 <code>delete</code> 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++)  
    pArr[i] = new int;
```

```
delete[] pArr; ❌
```

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++)  
    pArr[i] = new int;
```

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

Dynamic allocation for objects

Dynamically allocating memory to objects

```
Cat *pKitty = new Cat(); //default constructor  
Cat *pKitty2 = new Cat(10); //constructor with parameter
```

Accessing member function

```
Dereference with Dot: (*pKitty).meow()  
                    Arrow: pKitty->meow()
```

Equivalent

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 -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;
}
```


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 must be initialized through the initializer list.

Project7: Attack of the Yorps

- Carefully read the specs and all the rules
- Complete the implementation in accordance with the description
- Four classes that represent the four kinds of objects this program works with: Game, Arena, Yorp, and Player.
- Remember: Accessors and Mutators?
- Start early
- Hard part: How different parts of the program work together and what the responsibilities of each class are
- Time due: 11:00 PM Thursday, June 6
- Turn in: A zip containing yorps.cpp file

Project7: Do's and Don'ts

- You can make whatever changes you want to the private parts of the classes: You may add or remove private data members or private member functions, or change their types.
- You must not make any deletions, additions, or changes to the public interface of any of these classes — we're depending on them staying the same so that we can test your programs.
- You can and will, of course, make changes to the implementations of public member functions, since the callers of the function wouldn't have to change any of the code they write to call the function.
- You must not declare any public data members, nor use any global variables whose values may change during execution (except the global constants in the skeleton code).
- You may add additional functions that are not members of any class. The word `friend` must not appear in your program.
- Any member functions that you implement must never put an object into an invalid state, one that will cause a problem later on. (For example, bad things could come from placing a yorp outside the arena.)
- Any function that has a reasonable way of indicating failure through its return value, it should do so. Constructors pose a special difficulty because they can't return a value. If a constructor can't do its job, we have it write an error message and exit the program with failure by calling `exit(1)`.

Thanks!


Questions?

Complete online course evaluation!!

Please do it now on CCLE

Next Discussion: Final Review

Some of the materials presented have been taken from earlier TA discussions

A solid orange horizontal bar at the bottom of the slide.