CS 31 Discussion 1J

ABDULLAH-AL-ZUBAER IMRAN

WEEK 9: DYNAMIC MEMORY ALLOCATION & REST

Recap

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

Discussion Objectives

Review and practice things covered during lectures

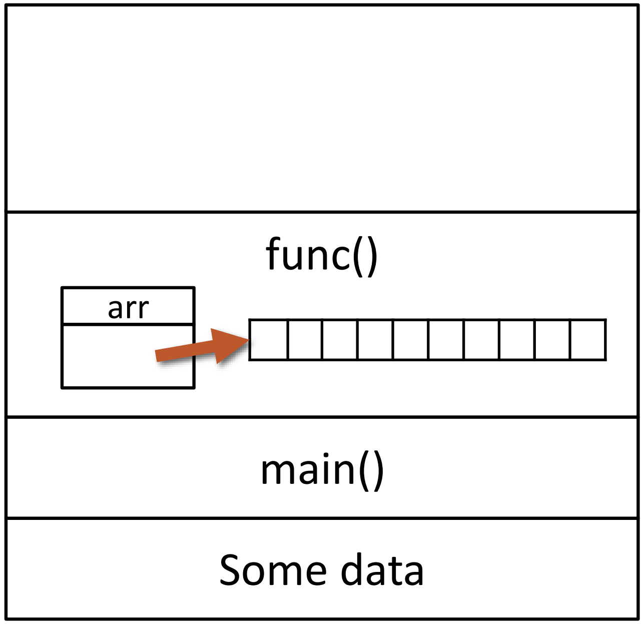
- Dynamic Memory Allocation
- Memory Leak
- Class Destructor

- Worksheet 8

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;
}
```



func()

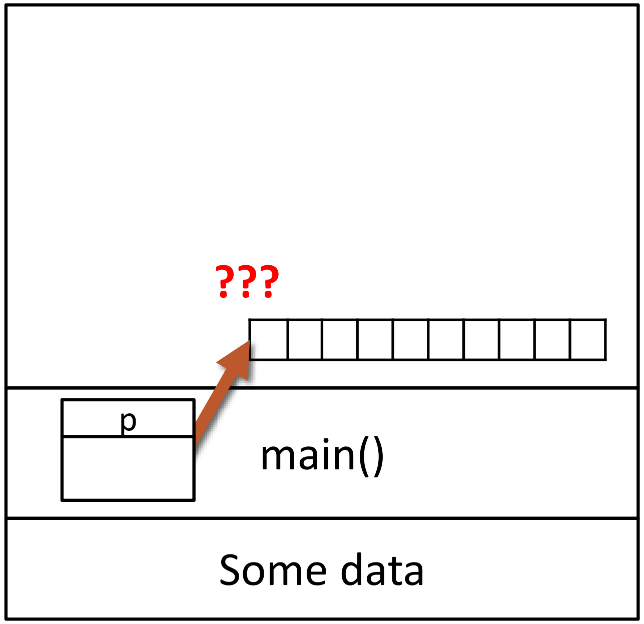
arr

main()

Some data

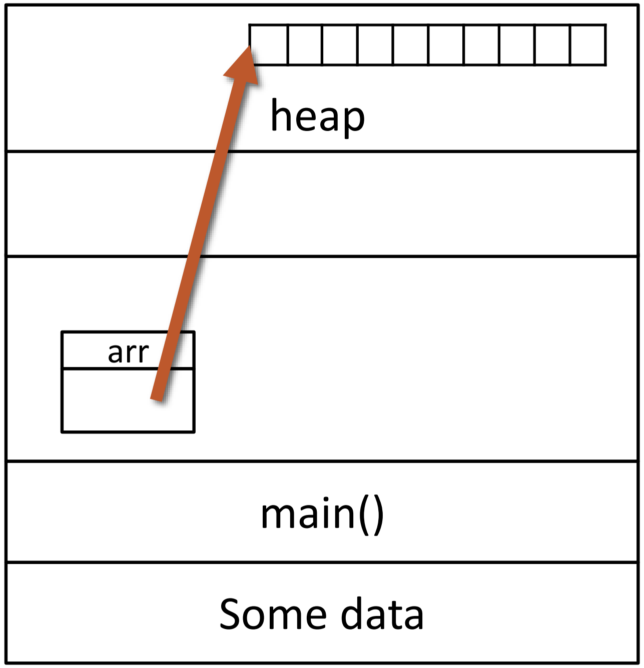
Dynamic Allocation of Memory

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



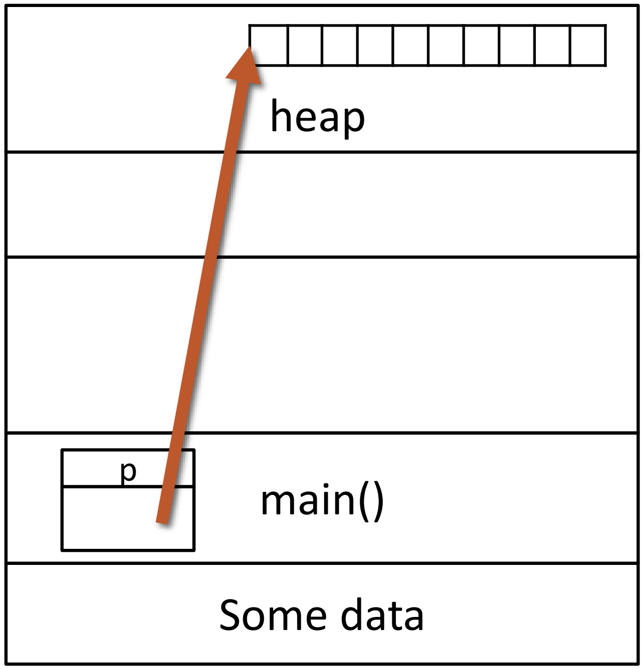
Dynamic Allocation of Memory

```
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;
}
```



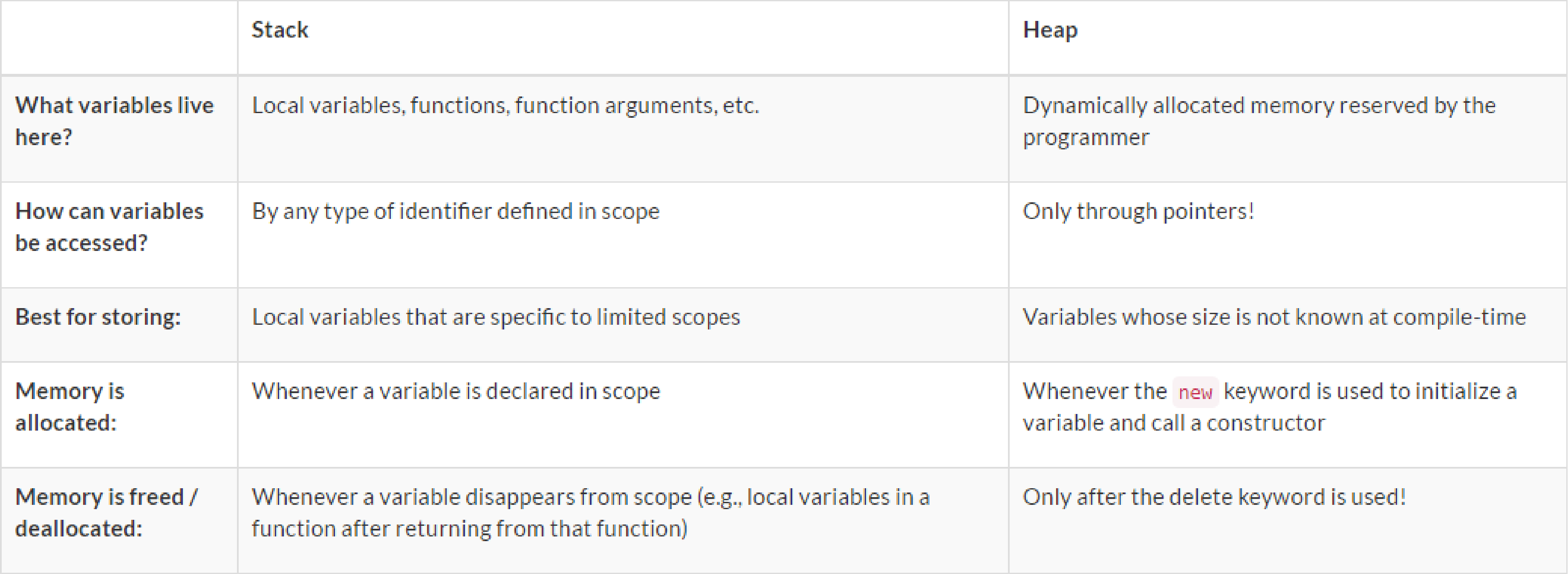
Stack vs. Heap

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



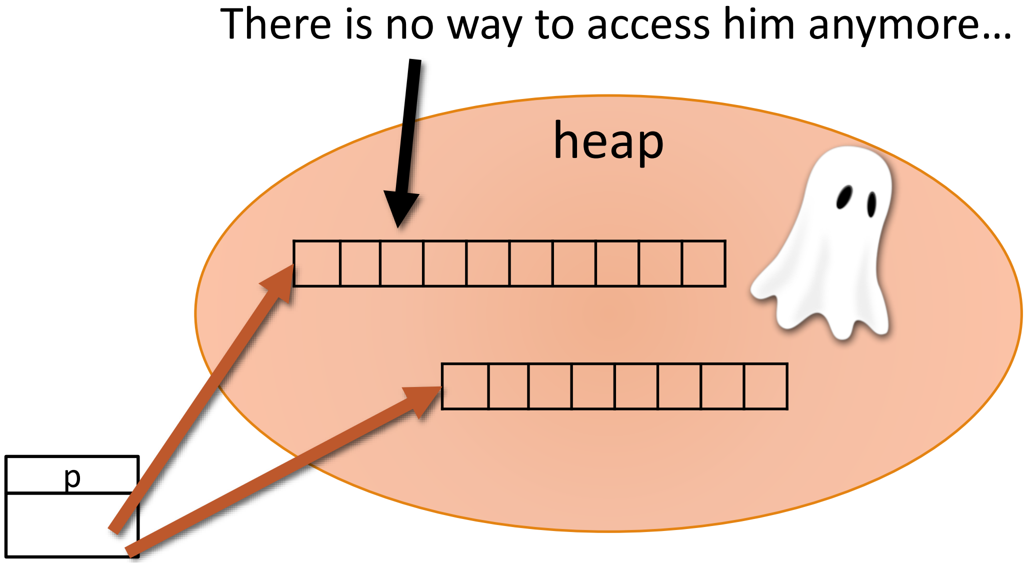
	Stack	Heap
What variables live here?	Local variables, functions, function arguments, etc.	Dynamically allocated memory reserved by the programmer
How can variables be accessed?	By any type of identifier defined in scope	Only through pointers!
Best for storing:	Local variables that are specific to limited 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 must 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;
}
```

Friend Function

Nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmer to write long and complex codes. Using a friend function or/and a friend class, private/protected data can be access from non-member friend function

```
class Distance

{

   private:

      int meter;

   public:

      Distance(): meter(0) { }

      //friend function

      friend int addFive(Distance);

};
```

```
// friend function definition
int addFive(Distance d)
{
   //accessing private data from non-member
function
   d.meter += 5;
   return d.meter;
}

int main()
{
   Distance D;
   cout<<"Distance: "<< addFive(D);
   return 0;
}
```

Function Overloading

What will be the output?

```
class printData {
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
      cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};
```

```
int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Operator Overloading

```
class Test
{
   private:
      int count;

   public:
       Test(): count(5){}

       void operator ++()
       {
          count = count+1;
       }

       void Display() { cout<<"Count: "<<count; }
};
```

```
int main()
{
    Test t;

    // this calls "function void operator ++()" function
    ++t;
    t.Display();
    return 0;
}
```

Project 7: Centenial

Time due: 9:00 PM Friday, March 13th

- The game will focus on two players, a single computer Player and a single human Player.
- The human Player will always go first.
- Players traverse the board from 1 through 12.
- Each player will begin with a marker initially placed outside the board.
- Each player will need to move their marker from spot 1 to spot 12 in order sequentially.
- The first player reaching slot 12 will win the game.
- In their turn, a player rolls three six-sided dice and then moves their marker.
- If the player has not yet entered the board, a roll with any number of dice totaling the value one is needed to enter slot 1 on the board.
- Once slot 1 is achieved, a roll with any number of dice totaling the value two is needed to enter slot 2.
- The value of a single die or the sum of any combination of two or three dice can be used to move to the next slot.
- A player can move multiple slots in a single turn by combining dice values in different ways.

Project 7

What to Turn in:

1. The text files named **Die.h** and **Die.cpp** that implement the Die class
2. The text files named **Player.h** and **Player.cpp** that implement the Player class
3. The text files named **Board.h** and **Board.cpp** that implement the Board class
4. The text files named **Centennial.h** and **Centennial.cpp** that implement the Centennial class, and
5. The text file named **main.cpp** which will hold your main program.
6. Your source code should have helpful comments that explain any non-obvious code.
7. Report: A file named **report.doc** or **report.docx** , or **report.txt** that contains:
   - A brief description of notable obstacles you overcame.
   - A list of the test data that could be used to thoroughly test your functions, along with the reason for each test.

Thanks!

Questions?

Complete Evaluation for the class

Today’s discussion slides can be found at

https://github.com/zubaerimran/W20-CS31-1J/blob/master/week9/winter20_cs31_w9.pdf

Next Discussion: Final Exam Review

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