Memory and C++ Pointers

CMPT 225

Outline

- C++ objects and memory
- C++ primitive types and memory

  - Note: “primitive types” = int, long, float, double, char, …

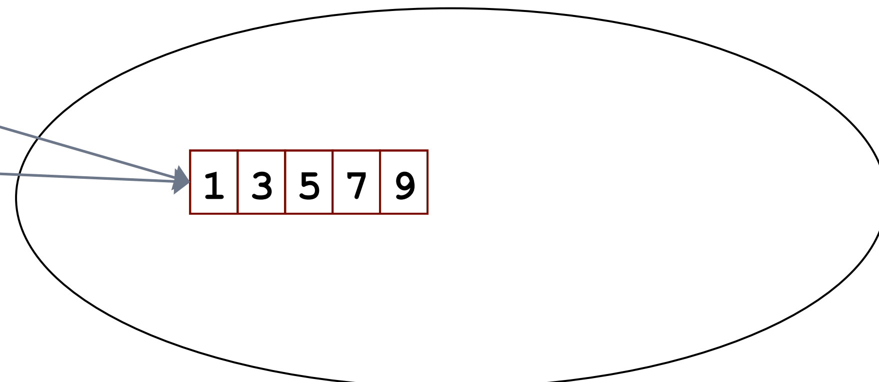
Dynamic Memory Example (from cmpt225_2stack, Java)

```java
// Java code
// in function, f …
int arr[];
arr = getOrdArray(5);
// …
```

```java
public int[] getOrdArray(int n){
    int arr[] = new int[n];
    for (int i = 0; i < arr.length; ++i){
        arr[i] = i * 2 + 1;
    }
    return arr;
}
```

getOrdArray	•	5
	arr	n
f	…	•
	…	arr
…		

stack (static)



heap (dynamic)

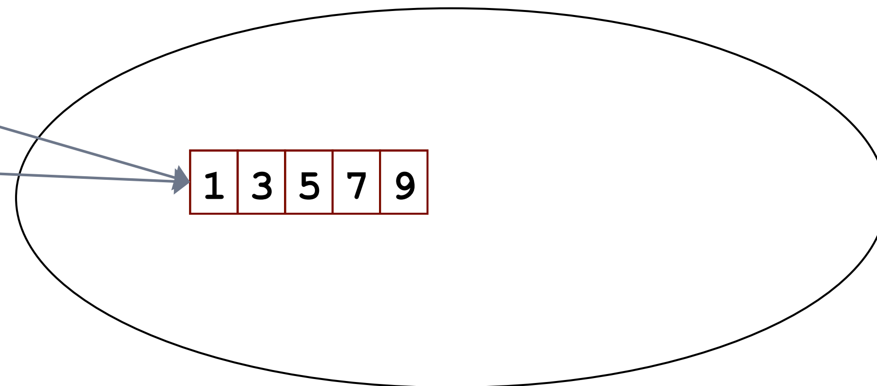
Dynamic Memory Example (from cmpt225_2stack, C++)

```
// in function, f …
// C++ code
int *arr;

arr = getOrdArray(5);
// …
```

```
int * getOrdArray(int n){
    int *arr = new int[n];
    for (int i = 0; i < n; ++i){
        arr[i] = i * 2 + 1;
    }
    return arr;
}
```

getOrdArray	•	5
	arr	n
f	…	•
	…	arr
…		

stack



heap

C++

C++ and Memory

- In C++:
  - Both primitive types and objects can be allocated either on the stack or on the heap
  - Both primitive type and object *value* and *reference* variables are allowed
    - Hence, there needs to be C++ notation to distinguish between the two

Referring to things in C++: Pointers

- There are two ways to refer to things in C++
  - The first is *pointers*
  - The * character is used to denote a pointer

```
// n is a Node object
Node n;
// n is a pointer to a Node object
Node *n;
```

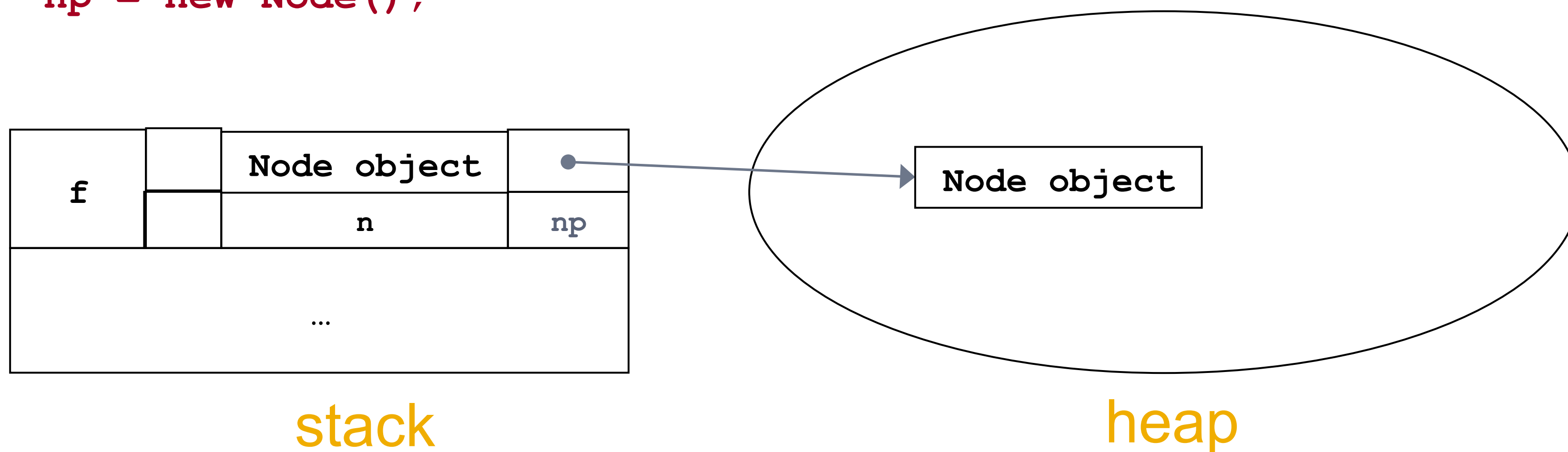
Heap vs. Stack Variables in C++

- Variables in methods are allocated in stack memory
- C++ uses the keyword **new** to allocate space in the heap

```
// n is a Node object, in stack
Node n;
// np is a pointer to a Node variable, np is in stack
Node *np;

// new creates a Node object, in heap
// np points to this object
np = new Node();
```

C++ Objects on Stack/Heap

```
// n is a Node object, in stack
Node n;
// np is a pointer to a Node variable, np is in stack
Node *np;

// new creates a Node object, in heap
// np points to this object
np = new Node();
```



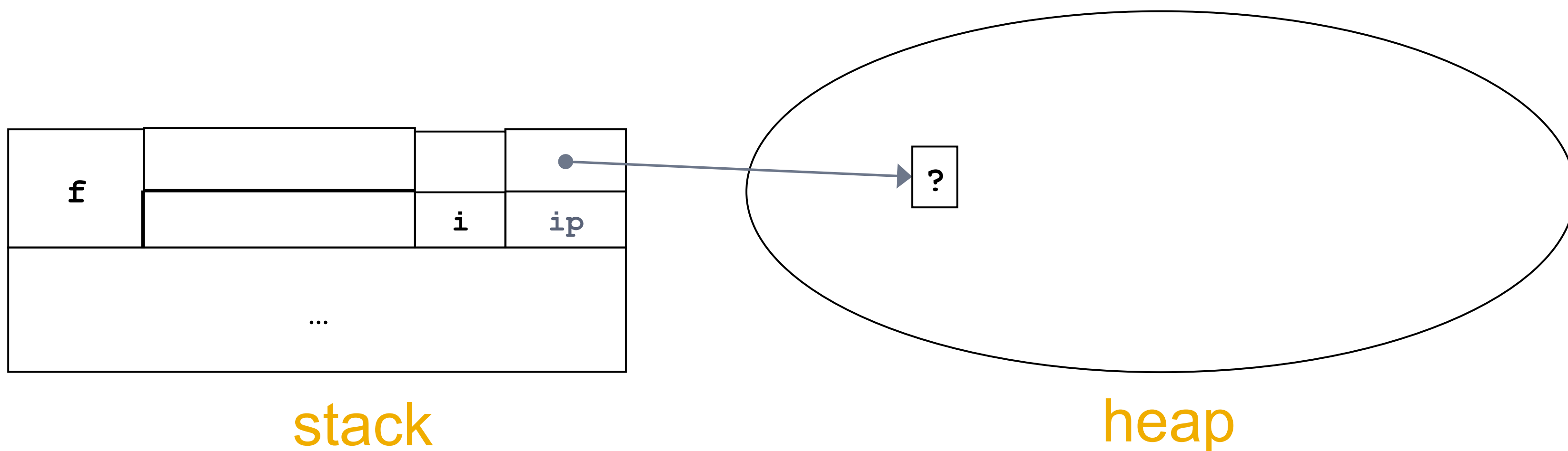
Heap vs. Stack Variables in C++

- In C++, you can do the same with primitive types, e.g.: int

```
// i is an integer variable, in stack
int i;
// ip is pointer to an integer variable, in stack
int *ip;
// new creates an integer variable, in heap
ip = new int;
```

C++ Primitives on Stack/Heap

```
// i is an integer variable, in stack
int i;
// ip is pointer to an integer variable, in stack
int *ip;
// new creates an integer variable, in heap
ip = new int;
```



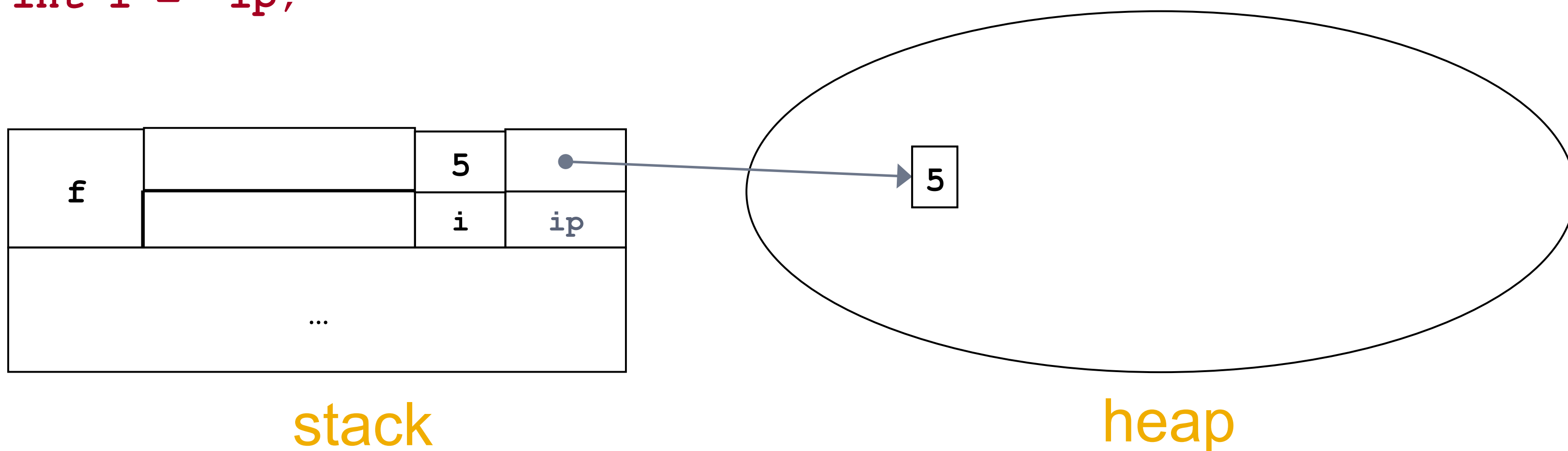
C++ Following Pointers

- How do we access the contents of the thing a pointer points to?
  - This is called “dereferencing” a pointer
    - The * notation is used again

```
// ip is pointer to an integer variable, in stack
int *ip;
// new creates an integer variable, in heap
ip = new int;

// *ip is the contents of the new integer
*ip = 5;
int i = *ip;
```

C++ Following Pointers

```
// ip is pointer to an integer variable, in stack
int *ip;
// new creates an integer variable, in heap
ip = new int;

// *ip is the contents of the new integer
*ip = 5;
int i = *ip;
```



C++ Following Pointers: Objects

- There is a shorthand for following pointers and accessing object methods / variables
  - Uses the -> characters

```
// np is a pointer to a Node variable, np is in stack
// new creates a Node object, in heap
// np points to this object
Node *np = new Node(5);

// both of these run the getData method on the Node object
int i = (*np).getData();
int i = np -> getData();
```

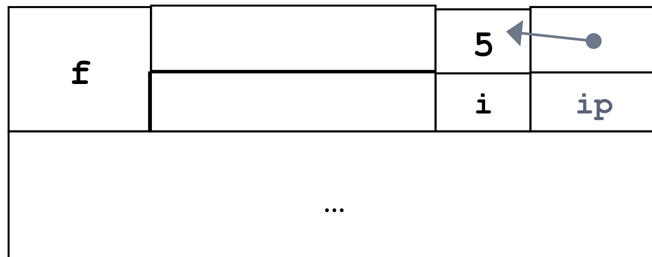
C++ Obtaining Addresses

- C++ allows one to obtain the address of an existing object / variable
  - This is called "referencing"
    - Uses the & operator ("address of")

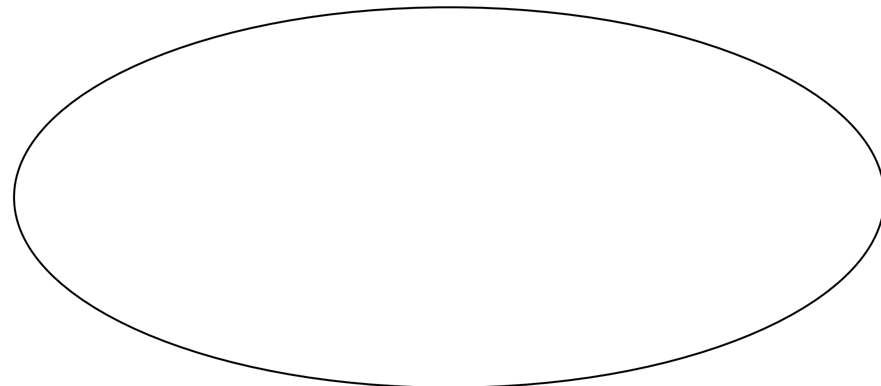
```
// i is an integer variable, in stack
int i;
// ip is pointer to an integer variable, in stack
int *ip;

// ip refers to the memory where i resides
ip = &i;
```

C++ Obtaining Addresses

```
// i is an integer variable, in stack
int i;
// ip is pointer to an integer variable, in stack
int *ip;

// ip refers to the memory where i resides
ip = &i;

*ip = 5;
```



stack



heap

C++ Memory Pitfalls

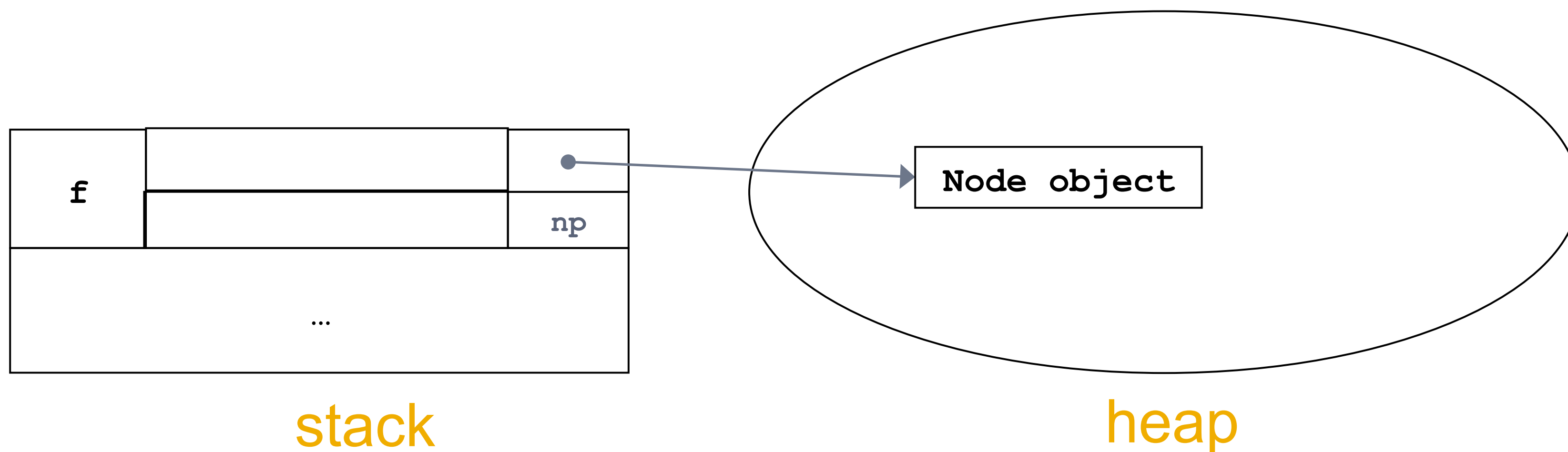
Taking Out the Trash in C++

- Java does Garbage Collection for you
- C++ you need to do it yourself
  - If you don't want an object any longer, call delete
    - If it's an array, call delete [ ], which calls delete on all array elements

- Bugs result if mistakes are made

C++ Delete

```
// np is a pointer to a Node variable, np is in stack
// new creates a Node object, in heap, np points to this object
Node *np = new Node();

// delete frees the heap memory referred to by np
delete np;
```



Stack Objects?

- In C++ objects can be in stack memory (unlike Java)
- Delete is automatically called on them when a method returns
  - Don’t manually delete them

C++ Stack Objects

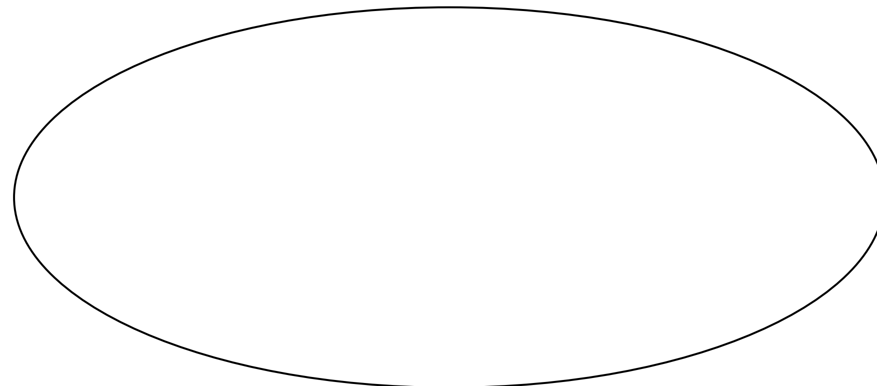
```
// in function, f …
Node n;
g();
// …
```

```
void g (){
   Node m;
   Node r;
}
```

delete is called on m and r

g	…	nodeObj	nodeObj
	…	r	m
f	…		nodeObj
	…		n
…			

stack (static)



heap (dynamic)

Memory Pitfalls

- Two major bug types can result if mistakes are made
  - Memory leaks
  - Dangling pointers

Memory Leaks

- Memory leaks occur if there is heap memory which is not pointed to by *any* variable (at any scope)
  - No pointers to the memory in the current method nor any below it on the stack
    - Including global variables
- There is no way to access the memory
- The system will not use the memory for another object/variable
- Eventually, you might run out of memory

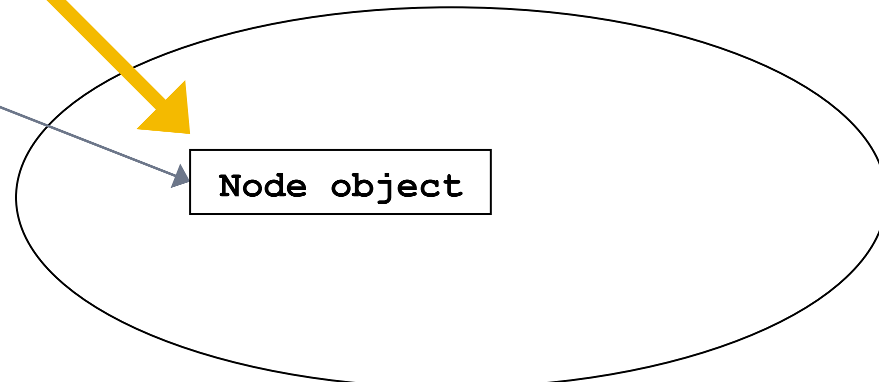
C++ Memory Leak

```
// in function, f …
g();
// …
```

```
void g (){
   Node *m = new Node();
}
```

This memory is not accessible

stack (static)

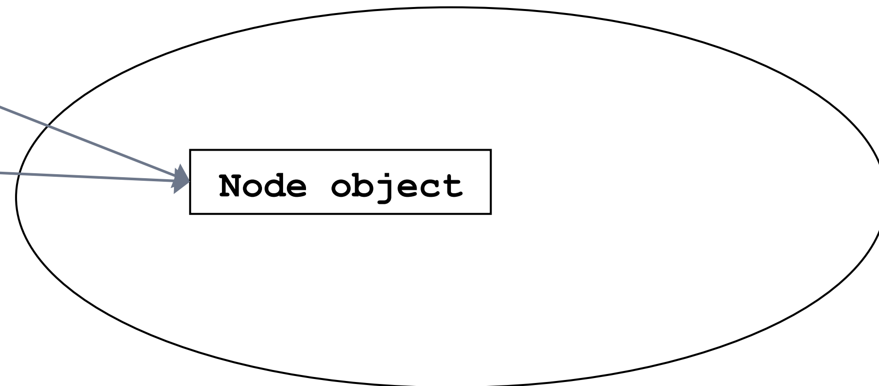


heap (dynamic)

C++ Memory Leak?

```
// in function, f …
Node *n;
n = g();
// …
```

```
Node * g (){
   Node *m = new Node();
   return m;
}
```



stack (static)

heap (dynamic)

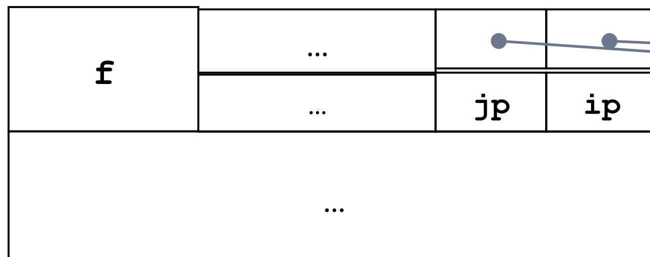
Dangling Pointers

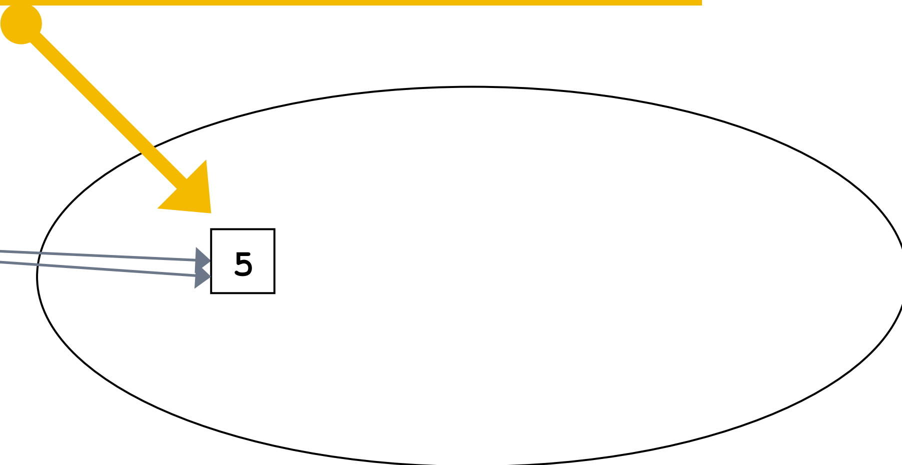
- Once you call delete, or a method returns, memory is gone
- If you try to refer to this memory you will get an error*
  - If it is being used by something else
    - Which will likely happen, but the error symptoms can be confusing

C++ Dangling Pointer

```
// in function, f …
int *ip = new int;
int *jp = ip;
*ip = 5
delete ip;
// …
*jp = 6;
```

This memory is not available





stack (static)

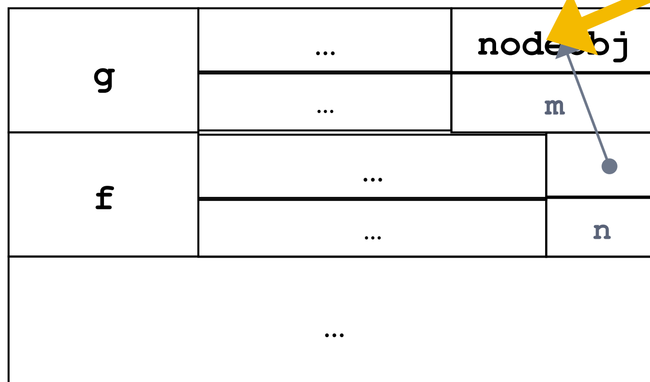
heap (dynamic)

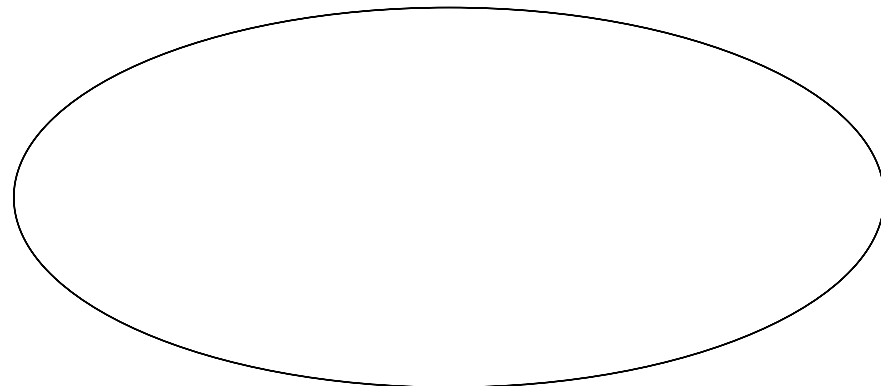
C++ Dangling Pointer?

```
// in function, f …
Node *n;
n = g();
// …
```

```
Node * g (){
    Node m;
    return &m;
}
```

This memory is not available





stack (static)

heap (dynamic)

References, the other way

C++ References

- There are two ways to do refer to things in C++:
  - Pointers
    - Which we just did
  - References

C++ References

- C++ also has *references* in addition to *pointers*
- References can be thought of as a restricted form of pointer
  - A few differences, key ones:
    - References cannot be NULL, pointers can
    - References cannot be reassigned, pointers can
      - This means they must be assigned at declaration time
    - Different syntax for access
      - Leads to cleaner code (but perhaps harder to understand)

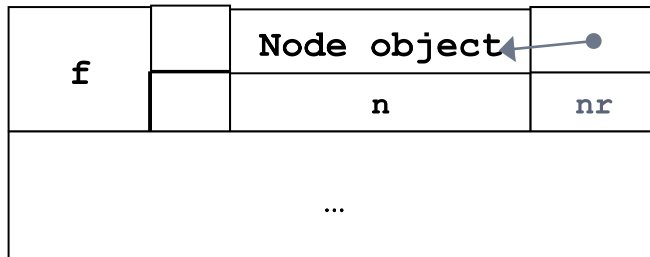
C++ References Syntax

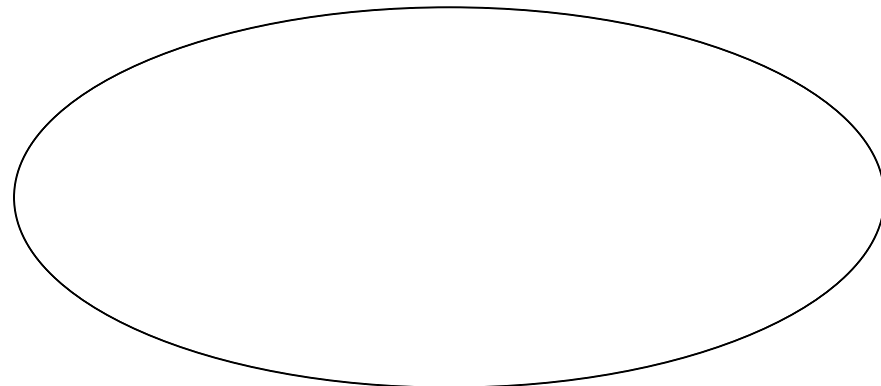
- The & character is used to denote references
  - Yes, the same character as address-of

```
// n is a Node object, in stack
Node n;
// nr is a reference to a Node object, in stack
// nr refers to the object n
Node &nr = n;
```

C++ Objects on Stack/Heap

```
// n is a Node object, in stack
Node n;
// nr is a reference to a Node object, in stack
// nr refers to the object n
Node &nr = n;
```





stack

heap

C++ References Syntax cont.

- References are used with same syntax as Java
  - Use the . character

```
// n is a Node object, in stack
Node n;
// nr is a reference to a Node object, in stack
// nr refers to the object n
Node &nr = n;

// both of these call the getData() method on the Node
int i = n.getData();
int i = nr.getData();
```

What are references for?

- Often used for function / method parameters
  - “Pass by reference” vs. “Pass by value”

```
void foo (int x) {
   x=2;
}

int main () {
   int y = 4;
   foo(y);
   cout << y;

   return 0;
}
```

```
void foo (int& x) {
   x=2;
}

int main () {
   int y = 4;
   foo(y);
   cout << y;

   return 0;
}
```

Summary

Summary

- Where do variables go?
  - C++
    - If it's a variable declaration, in stack
    - If it's a *new* statement, in heap
      - In C++, both primitive types and objects can go in either stack or heap

Summary

- How do I refer to variables?
  - C++
    - Pointers
      - * notation
        - * in type to denote "it's a pointer to a"
        - * in usage to denote "follow this pointer to"
    - References
      - & notation

Summary

- How do I manage memory?
  - C++
    - Call delete manually (or delete [ ] for arrays)
      - Watch out for bugs
        - Memory leaks (forgot to delete)
        - Dangling pointers/references (deleted when you shouldn't have)

Readings

- Carrano
  - Ch. 4.1