CSE 100:
REFERENCES AND TEMPLATES

Goals for today

- Draw memory model diagrams for C++ references
- Explain how destructors work
- Explain pass-by-reference and constants in C++
- Extend the BSTNode class to use templates

PA1: Implementing BST operations in C++

- Quick note:

height() – returns the height of the BST. Empty trees have height 0

An int-based BST in C++

In: BSTNodeInt.h

```
class BSTNodeInt {
public:
  BSTNodeInt* left;
  BSTNodeInt* right;
  BSTNodeInt* parent;
  int const data;

  BSTNodeInt( const int & d );
};
```

In: bstTest.cpp

```
#include "BSTNodeInt.h"
#include <iostream>

using namespace std;

int main()
{
    BSTNodeInt* n1 = new BSTNodeInt(5);
    cout << "Created a BST node with data "
         << n1->data << endl;
    delete n1;
}
```

Fixing the memory leak!

You must `delete` every piece of data you create with `new`. But usually there's no need to also set the pointer to NULL.

`delete` will call the object's destructor, if one is defined.

Memory allocation and destructors

```
class MyClass {
public:
  std::vector<int> vec;
  std::vector<int>* vecPtr;
  MyClass();



};
```

```
MyClass::MyClass() {
    vec = std::vector<int>(10);
    vecPtr = new std::vector<int>(10);
}
```

```
int main() {
  MyClass* x;
  x = new MyClass();

  MyClass* y = x;

  delete y;
}
```

Does this code have a memory leak?

[✓ A. Yes]

B. No

(In discussion, explain why it does or doesn't)

Memory allocation and destructors

```
class MyClass {
public:
  std::vector<int> vec;
  std::vector<int>* vecPtr;
  MyClass();
  ~MyClass();  // Destructor
};

int main() {
  MyClass* x;
  x = new MyClass();

  MyClass* y = x;

  delete y;
}
```

```
MyClass::MyClass() {
    vec = std::vector<int>(10);
    vecPtr = new std::vector<int>(10);
}

// Must delete anything the class created
// with new!
MyClass::~MyClass() {
    delete vecPtr;
}
```

References in C++

An int-based BST in C++

In: `BSTNodeInt.h`

```
class BSTNodeInt {
public:
  BSTNodeInt* left;
  BSTNodeInt* right;
  BSTNodeInt* parent;
  int const data;

  BSTNodeInt( const int & d );
};
```

Parameter passing and assignment in C++ is done by value, by default. If you want to avoid a copy, you must use references

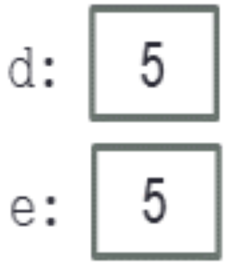
(NOTE: for ints pass by reference to this constructor doesn't make sense, but for our templated class it will)

References in C++

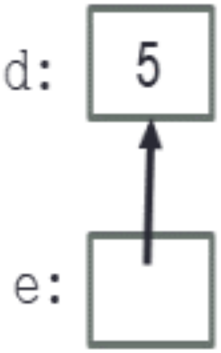
```
int main() {
  int d = 5;
  int & e = d;


}
```

Which diagram represents the above code?

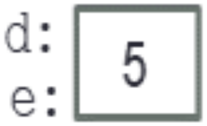
A.



[✓ B.]



C.



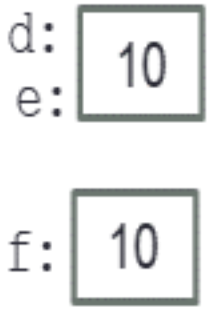
D. This code causes an error

References in C++

```
int main() {
  int d = 5;
  int & e = d;
  int f = 10;
  e = f;


}
```

How does the diagram change with this code?

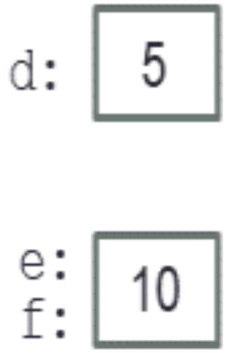
A.



d:
e: 10

f: 10

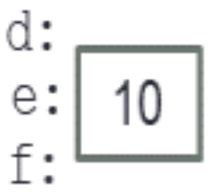
B.



d: 5

e:
f: 10

C.



d:
e: 10
f:

D. Other or error

Pointers and references, together! Draw the picture for this code

```
int a = 5;
int & b = a;
int* pt1 = &a;
```

What are three ways to change the value in the box to 42?

The const keyword

```
int main() {
  int const d = 5;
  int & e = d;
}
```

Does this code have an error? If so, why?

~~A. No, there is no error~~

B. Yes, there is an error (what is it?)

The const keyword

```
const int d = 5;

int const d = 5;
```

These mean the same thing. d cannot be reassigned, and the data stored in d (if is it mutable) may not be changed.

The const keyword

```
const int d = 5;

int const d = 5;
```

These mean the same thing. d cannot be reassigned, and the data stored in d (if is it mutable) may not be changed.

```
const int & e = d;

int const & e = d;
```

These also mean the same thing, as each other. e is an alias for d and cannot change the data stored in d

The const keyword

```
const int d = 5;

int const d = 5;
```

These mean the same thing. d cannot be reassigned, and the data stored in d (if is it mutable) may not be changed.

```
const int & e = d;

int const & e = d;
```

These also mean the same thing, as each other. e is an alias for d and cannot change the data stored in d

```
int f = 42;

const int * p = &f;

int const * p = &f;
```

These also mean the same thing, as each other. p is a pointer to f and cannot change the data stored in f

The const keyword

```
const int d = 5;

int const d = 5;
```

These mean the same thing. d cannot be reassigned, and the data stored in d (if is it mutable) may not be changed.

```
const int & e = d;

int const & e = d;
```

These also mean the same thing, as each other. e is an alias for d and cannot change the data stored in d

```
int f = 42;

const int * p = &f;

int const * p = &f;
```

These also mean the same thing, as each other. p is a pointer to f and cannot change the data stored in f

```
int * const p = &f;
```

This one is NOT THE SAME!

The const keyword: Rules

- The initially declared variable sets the rules about whether the data it stores is const or not.
- All pointers or references to that same data must be *at least as restrictive* in terms of how they allow the data to be changed.

The pesky 'const' keyword (for your review)

For each of the following statements, state whether

A. The value stored in the variable a cannot be changed after the statements
B. The value stored in the variable b cannot be changed after the statements
C. Both A and B
D. Neither A nor B
E. This statement(s) does not make sense/causes an error in C++

```
const int a = 5;
```

```
int a = 5;
const int & b = a;
```

```
const int a = 5;
const int & b = a;
```

```
const int a = 5;
int & const b = a;
```

```
int a = 5;
int * const b = &a;
```

```
const int a = 5;
const int * const b = &a;
```

BST, with templates:

```
template<typename Data>

class BSTNode {
public:
  BSTNode<Data>* left;
  BSTNode<Data>* right;
  BSTNode<Data>* parent;
  Data const data;

  BSTNode( const Data & d ) :
     data(d) {
    left = right = parent = NULL;
  }

};
```

BST, with templates:

```
template<typename Data>

class BSTNode {
public:
  BSTNode<Data>* left;
  BSTNode<Data>* right;
  BSTNode<Data>* parent;
  Data const data;

  BSTNode( const Data & d ) :
     data(d) {
    left = right = parent = NULL;
  }

};
```

How would you create **a BSTNode object** on the runtime stack?

A. BSTNode n(10);
B. BSTNode<int> n(10);
C. BSTNode<int> n = new BSTNode<int>(10);

Automatic type deduction with “auto”

BST, with templates:

```
auto p = new BSTNode<int>(10);
```

```
template<typename Data>

class BSTNode {
public:
  BSTNode<Data>* left;
  BSTNode<Data>* right;
  BSTNode<Data>* parent;
  Data const data;

  BSTNode( const Data & d ) :
     data(d) {
    left = right = parent = 0;
  }

};
```