# C++ Parameters, Return Values, References, Const, Copy Constructors

Dec 6th

full GUI

C++ alloc, dealloc, passing, mutating obj.

Dec 13th

final integration

app works

C++ virtualization

# Review pointers

- **4 main operations:**
  - **Declaring:** use a * in declaration
    - **int *myIntPointer;**
  - **Assignment:** must match type
    - **myIntPointer = &myInt;** *// & is address of operator*
    - **myIntPointer = new int;** *// new returns a pointer*
  - **Copying:** from one pointer to another
    - **int *myOtherIntPointer = myIntPointer;**
  - **De-referencing:** use * in expression to get to the value
    - ***myOtherIntPointer = 7;**
    - **int justAnInt = *myIntPointer;**

# Which of these won't compile?

Expression *evp = &ExpressionValue(27.0);

A ExpressionValue(27.0);

B ExpressionValue ev1(28.0);

C ExpressionValue evp2 = new ExpressionValue(29.0);

D new ExpressionValue(30.0);

E All of the above will compile

# Which of these won't compile?

**A**

```
ExpressionValue ev1(9.0);
ExpressionValue *evp2 = &ev1;
cout << evp2->getValue();
```

**B**

```
ExpressionValue *evp3 = new ExpressionValue(10.0);
cout << (*evp3).getValue();
```

pointer

evp3 → ...

(*evp3).

**C**

```
ExpressionValue ev4(11.0);
cout << (&ev4)->getValue();
```

object

address of

syntactic sugar

**D** All of the above will compile

# Delete

- **Returns an item back to the "free list"**
  - So it can be allocated again.
  - Doesn't overwrite memory
  - Doesn't affect pointer or other pointers to the same storage

- **As a C++ programmer you need to make sure you:**
  - Delete any memory you allocate when you are done with it
  - Not delete any memory that you are still using
  - Not reference any deleted memory

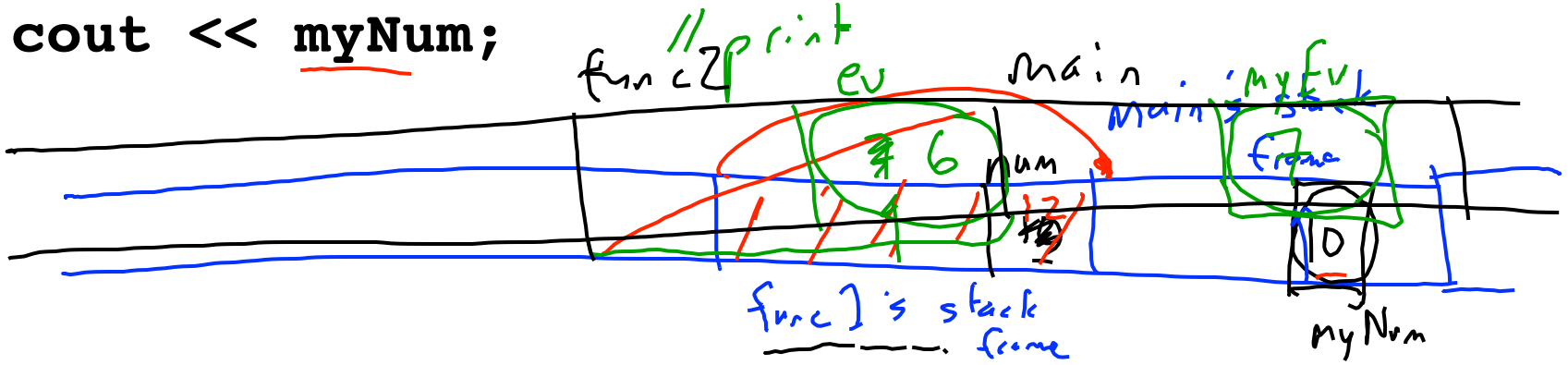
# Passing Parameters

- Like Java, C++ is pass by value (i.e., parameters are copied)

```
void func1(int num) {
    cout << num;
    num = 12;
}
```

```
int myNum = 10;
func1(myNum);
cout << myNum;
```

A) 1010
B) 1012
C) 1210
D) 1212
E) other



# Passing Parameters (objects)

```
void func2(ExpressionValue ev) {
    cout << ev.getValue();
    ev.setValue(6);
}

ExpressionValue myEv(7);
func2(myEv);
cout << myEv.getValue() << endl;
```

A) 66
B) 67
C) 76
D) 77
E) other

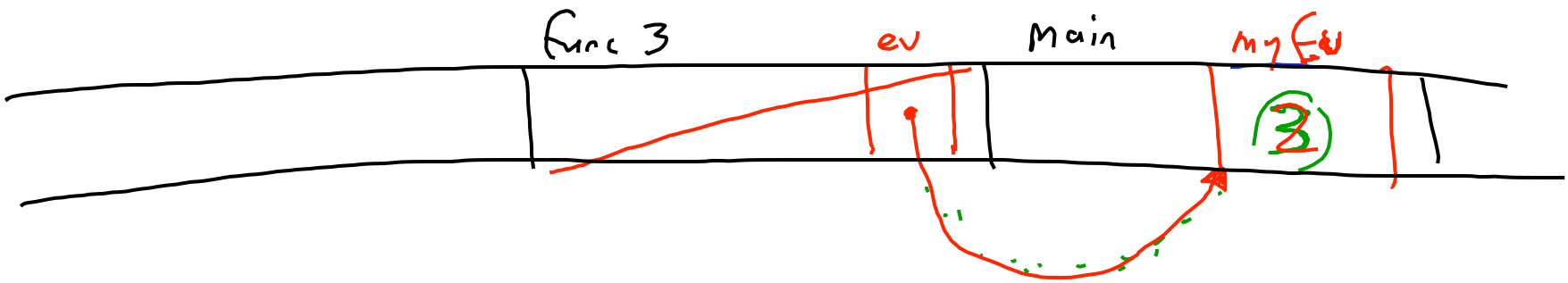
# Passing Parameters (pointers)

```
void func3(ExpressionValue *ev) {
    cout << ev->getValue();
    ev->setValue(3);
}

main
ExpressionValue myEv(2);
func3(&myEv);
cout << myEv.getValue() << endl;
```

23

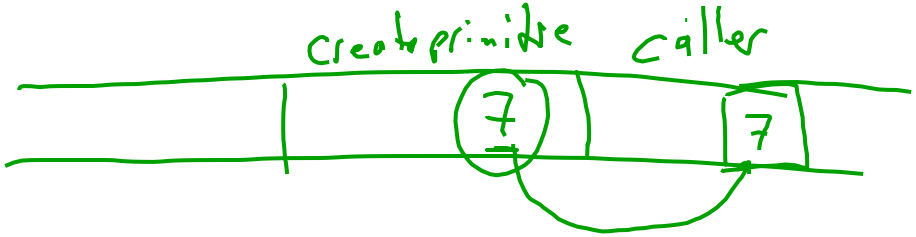
A) 22
B) 23
C) 32
D) 33
E) other



# Returning values from functions

- **Like arguments, return values are pass by value**
- **Return values are straightforward for primitives:**

```
int createPrimitive() {
    int value = 7;
    return value;
}
```



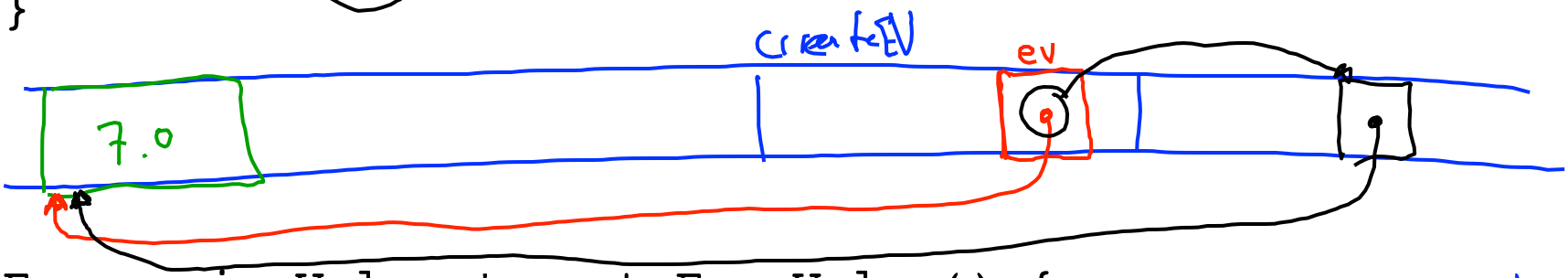

- **Less clear what happens exactly with objects**

we'd like to avoid...

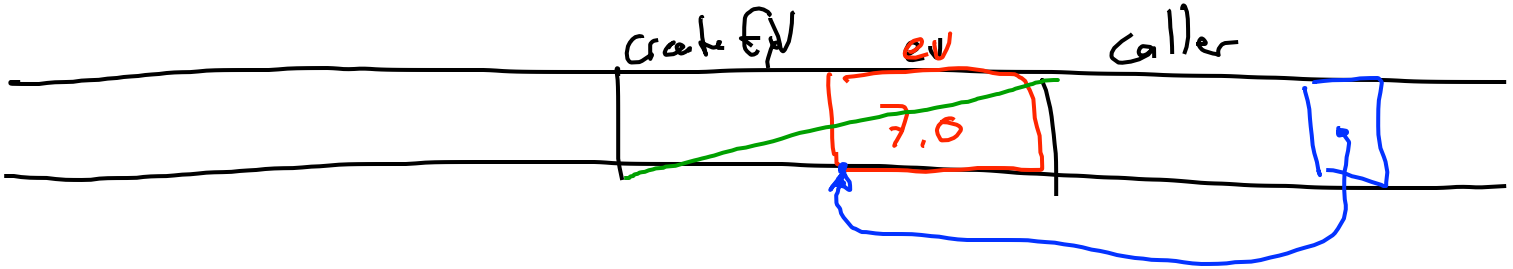
```
ExpressionValue createExprValue() {
    ExpressionValue expressionValue(7.0);
    return expressionValue;
}
```

# Returning pointers to things

```
ExpressionValue *createExprValue() {
    ExpressionValue *ev = new ExpressionValue(7.0);
    return ev;
}
```




```
ExpressionValue *createExprValue() {
    ExpressionValue expressionValue(7.0);
    return &expressionValue;
}
```

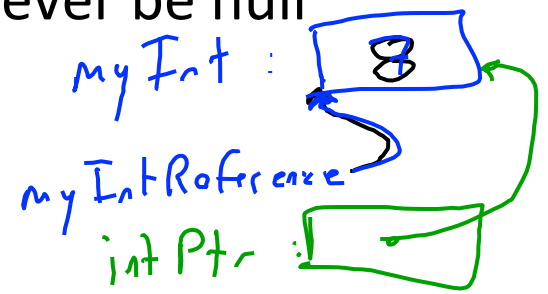


# Reference Variables

- **Some people dislike pointers:**
  - Pointers can be uninitialized or be null
  - Different notation is used for stack & heap alloc'ed objects
    - foo.getValue() vs. bar->getValue()
- **C++ provides Reference variables**
  - Declared using a &
  - They "refer" to another variable, but look like normal vars.
  - Must be initialized when created; can never be null

```
int myInt = 7;
int &myIntReference = myInt;
myIntReference = 8;
cout << myInt << endl;  // prints out 8
```

int * intPtr =



# Why Reference Variables? (Reason 1)

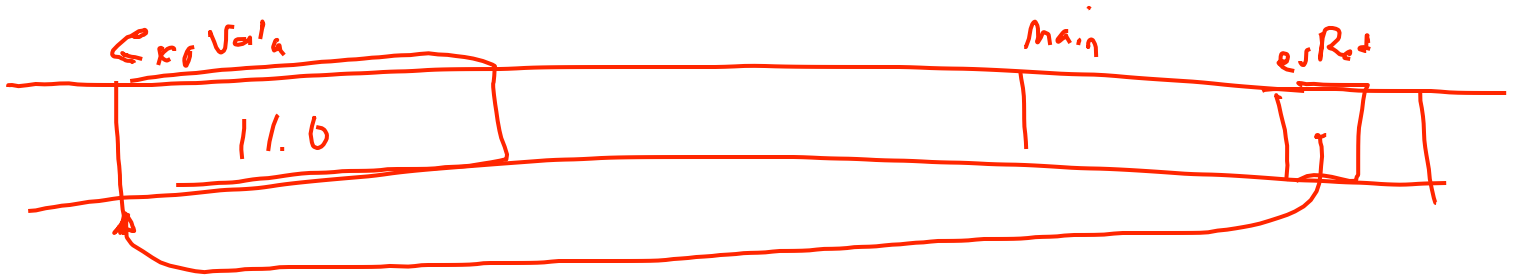
- They allow us to use same notation for heap allocations

```
int &myIntRef = *new int;
myIntRef = 7;
cout << myIntRef << endl;
```

```
int *intPtr = new int;
*intPtr = 7;
cout << *intPtr << endl;
```

main

```
ExpressionValue &evRef = *new ExpressionValue(11.0);
cout << evRef.getValue() << endl;
```



# Why Reference Variables? (Reason 2)

- For passing arguments and return values (w/o copying)

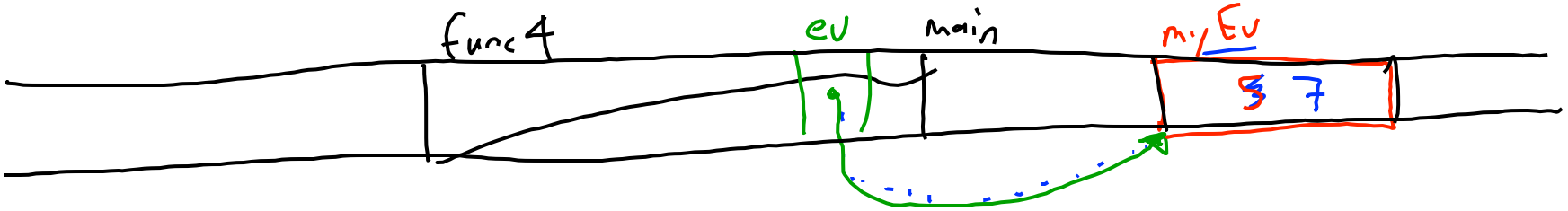
the object

```
void func4(ExpressionValue &ev) {
    cout << ev.getValue();
    ev.setValue(7);
}
```

5 7

A) 55
B) 57
C) 75
D) 77
E) other

```
ExpressionValue myEv(5);
func4(myEv);
cout << myEv.getValue() << endl;
```



# Why Reference Variables? (Reason 2, cont.)

- **Okay to return heap allocated values (w/o copy)**

```
ExpressionValue &createExprValue() {
    return *new ExpressionValue(7.0);
}
```


- **Still bad to return stack allocated variables**

```
ExpressionValue &dontDoThis() {
    ExpressionValue stackAllocated(7.0);
    return stackAllocated;
}
```

# Refernces

- **Is this valid Reference delcaration?**

```
ExpressionValue &evReference;
```

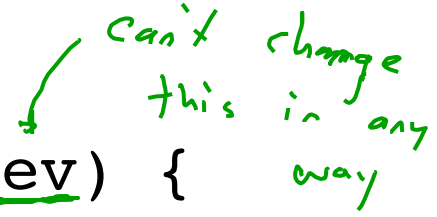
**A)** **It is valid**

**B)** **It is invalid**

# Const parameters

- **We previously saw that functions could be marked as const.**
  - Compiler ensures that function doesn't change "this" object
  - E.g., **double** getValue() **const**;

- **We can mark parameters as const**
  - Can not change the value of that parameter

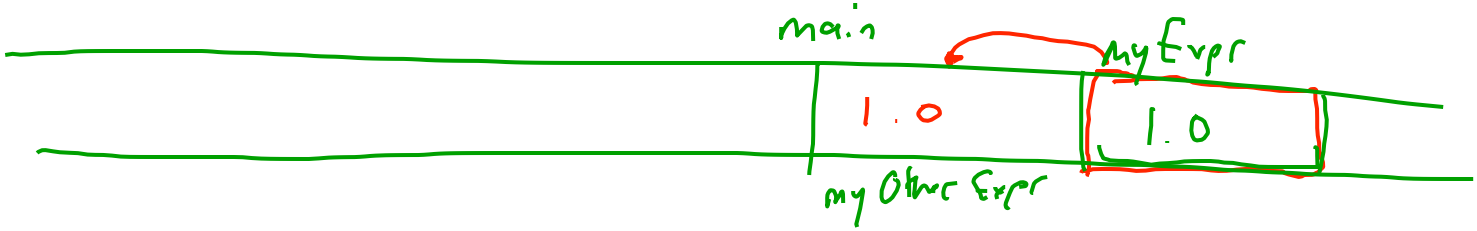



```
void func4(const ExpressionValue &ev) {
    cout << ev.getValue(); // fine
    ev.setValue(7);        // not allowed
}
```

# Copy Constructors

- **What happens when we copy an object?**

```
ExpressionValue myExpr(1.0);
ExpressionValue myOtherExpr = myExpr;
```




- **It invokes a copy constructor**
  - Be default, it does a bit-wise copy of the object
  - Can override, by declaring:

```
ExpressionValue(const ExpressionValue&);
```

# Copy Constructors (cont.)

- **Same if we pass/return objects (not pointers/references)**

```
void func2(ExpressionValue ev) {
    cout << ev.getValue();
    ev.setValue(6);
}

ExpressionValue createExprValue() {
    ExpressionValue expressionValue(7.0);
    return expressionValue;
}
```