Lecture No.15
Reference Variables

CC-213 Data Structures
Department of Computer Science
University of the Punjab

Slides modified very slightly from the late Dr. Sohail Aslam's lectures at VU

Reference Variables

- The symbol “&” has a few different purposes depending on where it occurs in code.
- When it appears in front of a variable name, it is the address operator, i.e., it returns the address of the variable in memory.

```
int x;
int* ptr = &x;
```

Reference Variables

- The symbol “&’ can also appear after a type in a function signature:
- For example, insert and remove from the BinarySearchTree class.

```
void insert( const EType& x );
void remove( const EType& x );
```

Reference Variables

- Or, in case we designed the BinarySearchTree class to hold integers only, i.e., no templates

```
void insert( const int& x );
void remove( const int& x );
```

Reference Variables

- The “&” indicates a parameter that is a *reference variable*.
- Consider the following three different functions:

Reference Variables

```
// example 1
int intMinus1( int oldVal)
{
    oldVal = oldVal – 1;
    return oldVal;
}
```

Reference Variables

```
// example 2
int intMinus2( int* oldVal)
{
    *oldVal = *oldVal – 2;
    return *oldVal;
}
```

Reference Variables

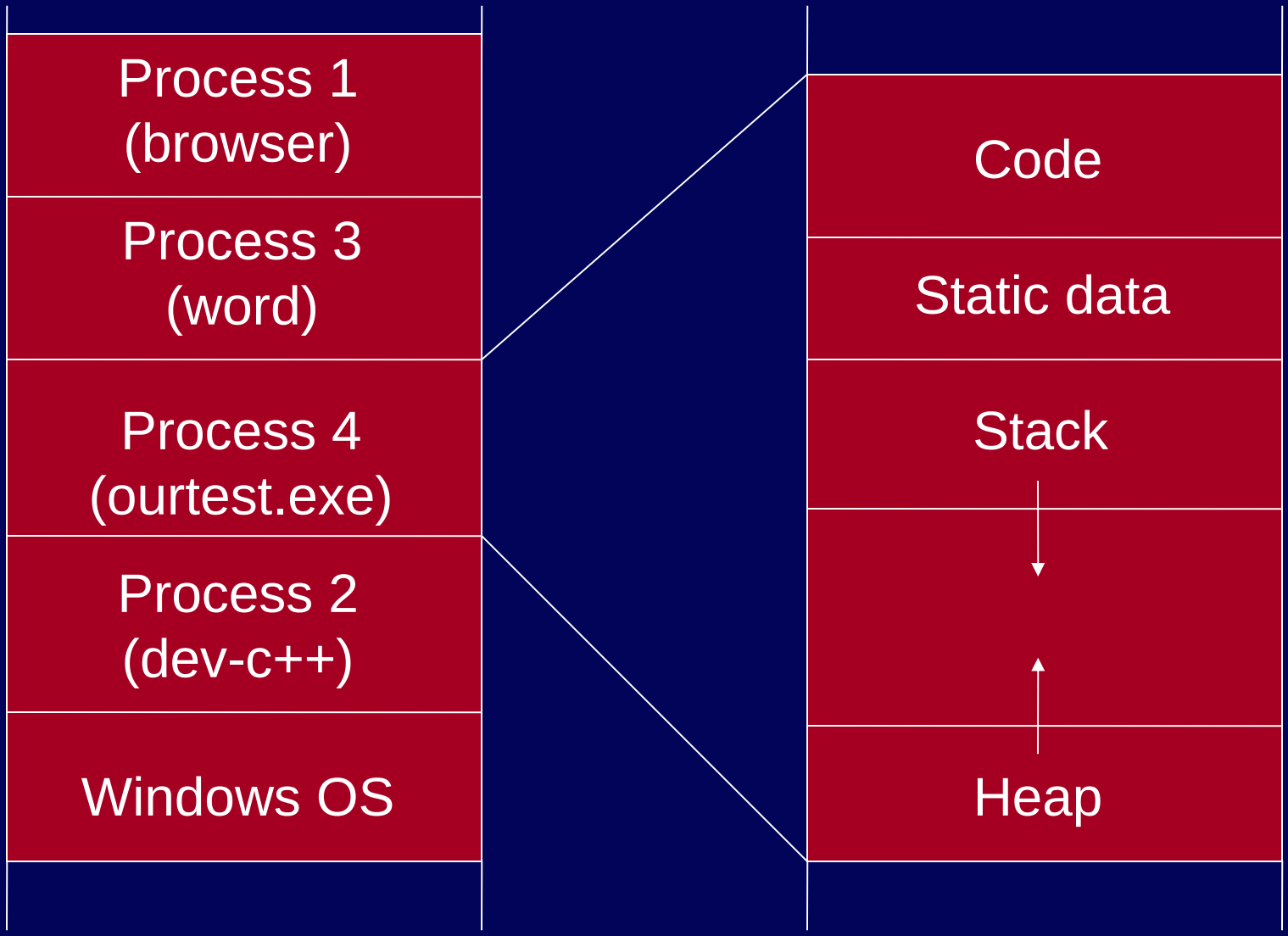
```
// example 3
int intMinus3( int& oldVal)
{
    oldVal = oldVal – 3;
    return oldVal;
}
```

Reference Variables

The caller function: calling **intMinus1**

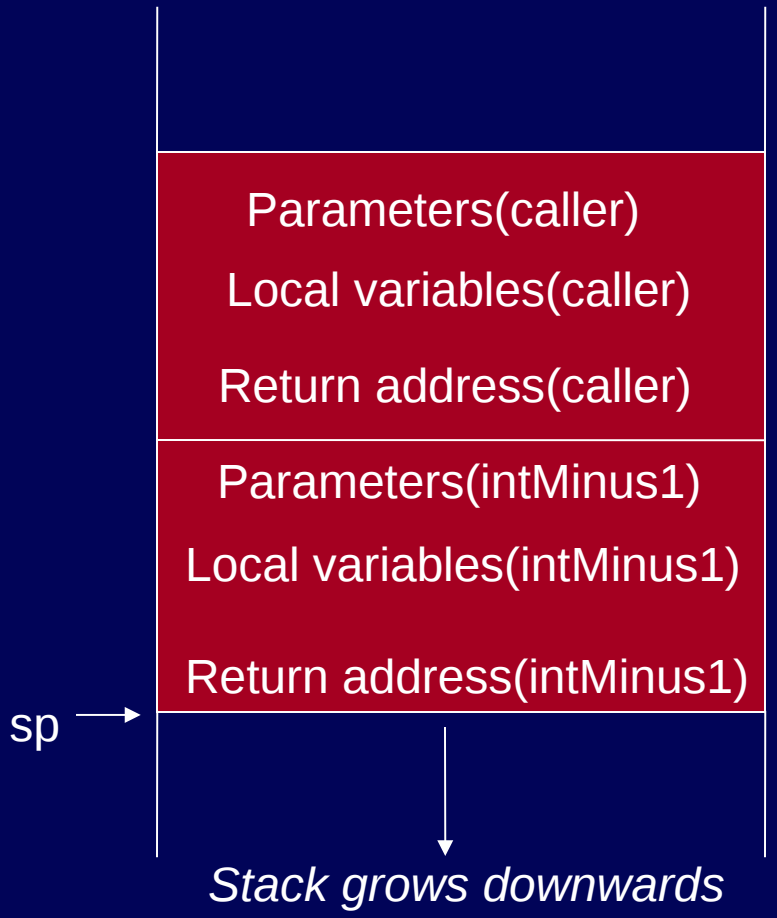
```
void caller()
{
   int myInt = 31;
   int retVal;
   retVal = intMinus1( myInt );
   cout << myInt << retVal;
}
```

Memory Organization



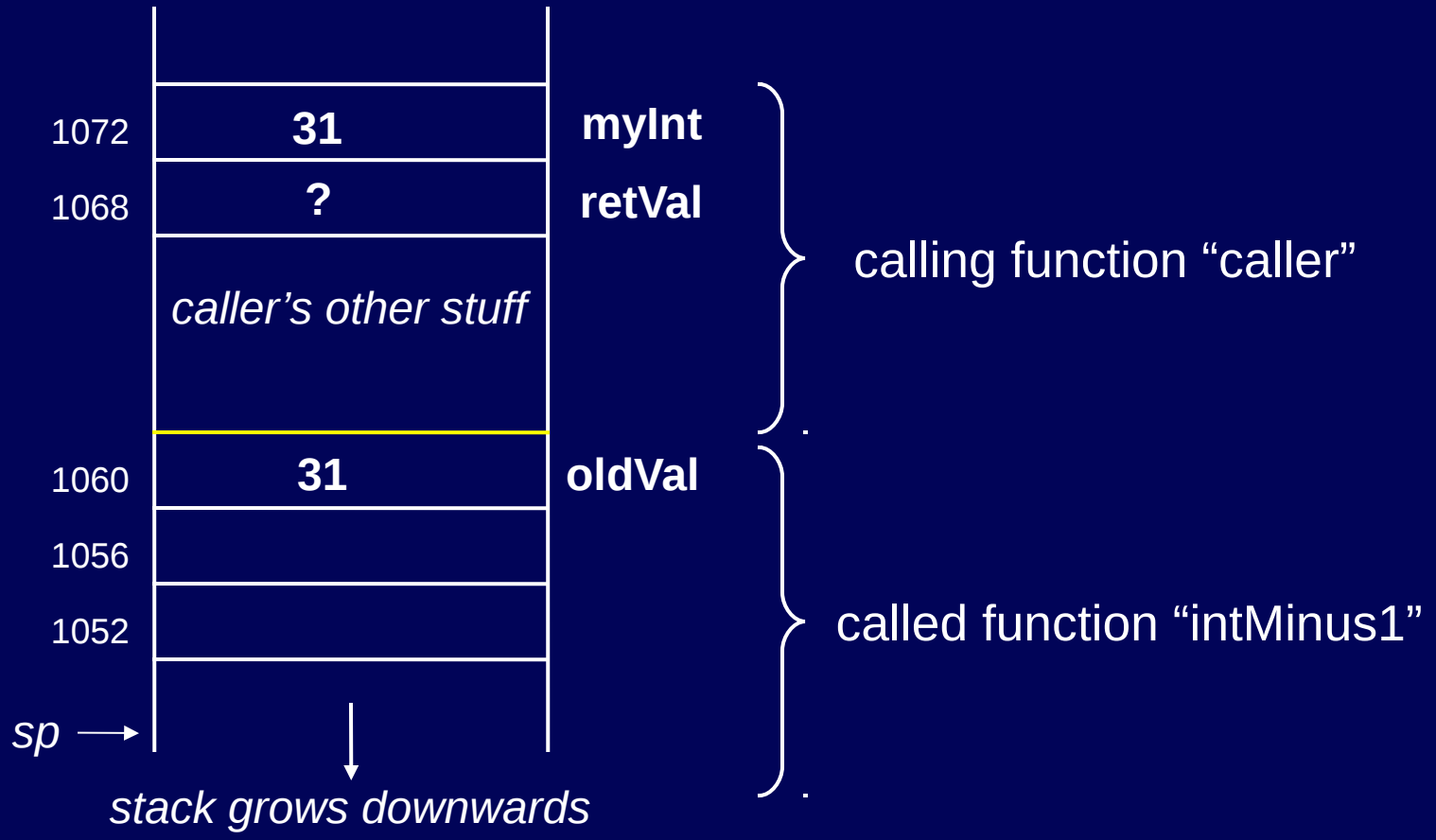
Reference Variables

- Call stack layout



Reference Variables

Call stack layout when `intMinus1` is called:



Reference Variables

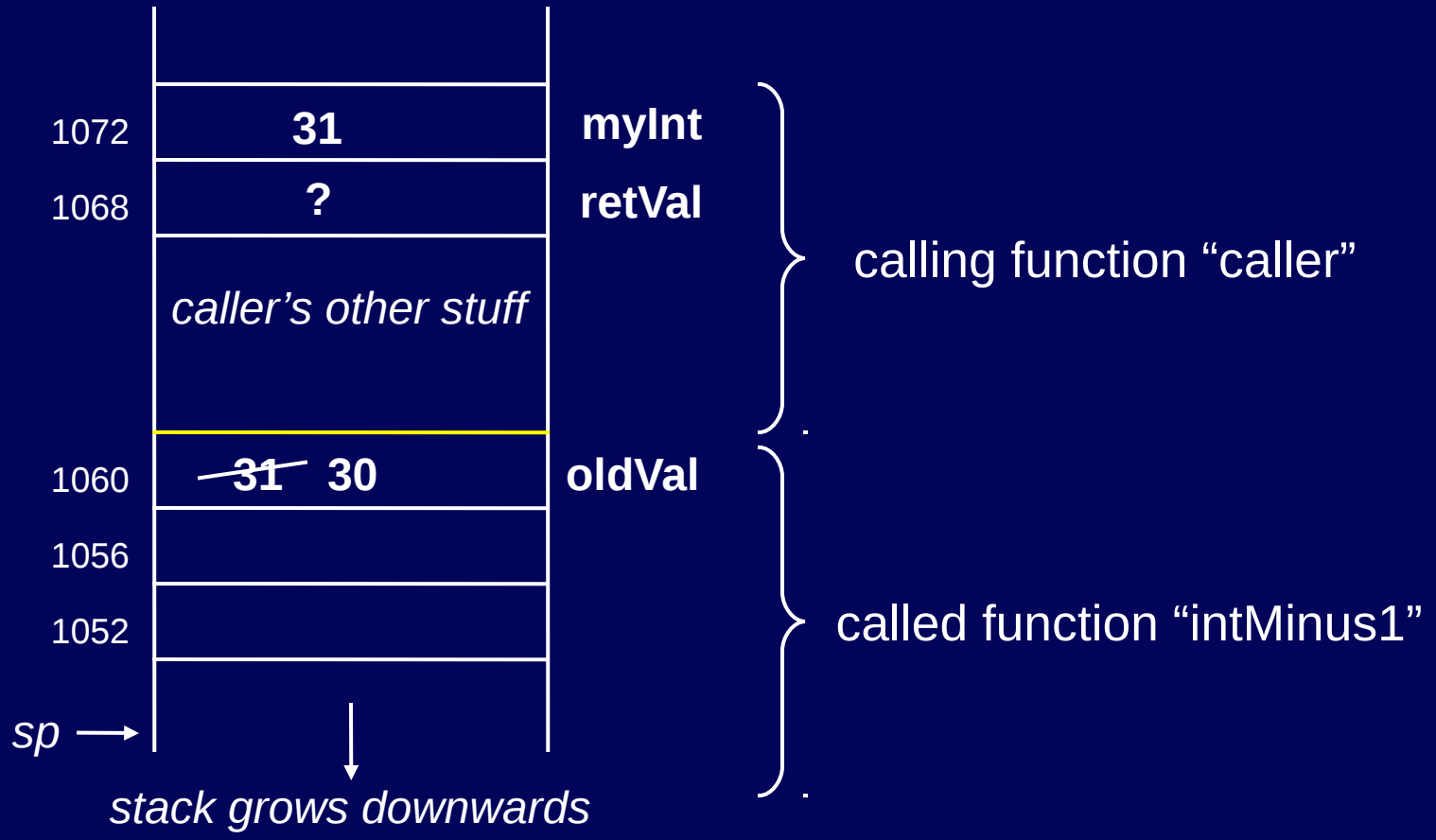
- How are `myInt` and `oldVal` related?
- Passing `myInt` to the function `intMinus1` results in a *copy* of `myInt` to be placed in parameter `oldVal` in the call stack.
- Alterations are done to the copy of “31” (stored in `oldVal`) and not the original `myInt`.

Reference Variables

- The original `myInt` remains *unchanged*.
- For this reason, this technique of passing parameters is called *pass by value*.
- Alterations are done to the copy of “31” (stored in `oldVal`) and not the original `myInt`.

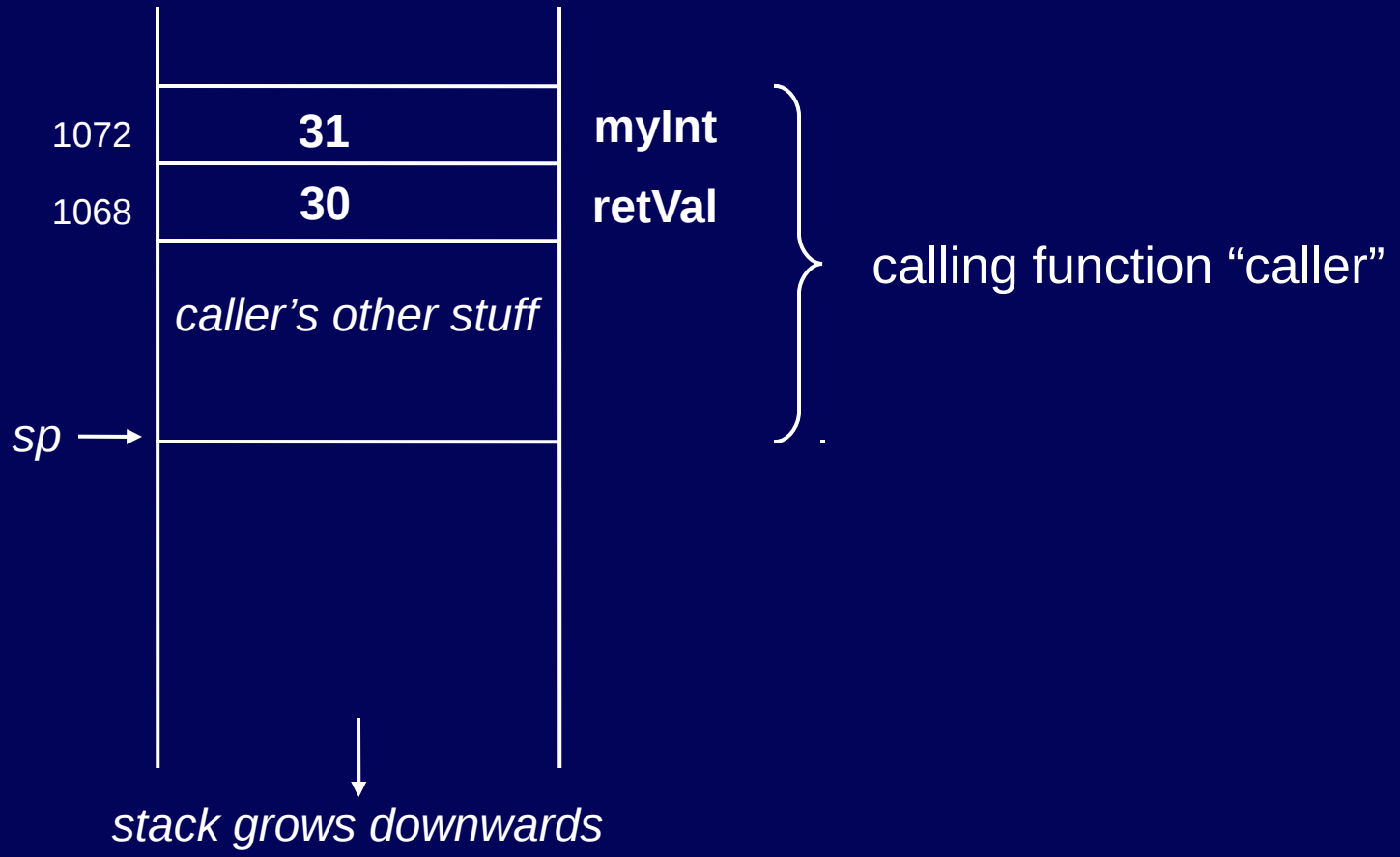
Reference Variables

Call stack layout after subtraction in `intMinus1`:



Reference Variables

Call stack layout after return from `intMinus1`:



Reference Variables

- We could have called **`intMinus1`** as

```
void caller()
{
   int retVal;
   retVal = intMinus1( 31 );// literal
   cout << myInt << retVal;
}
```

- Because it is the value that is passed. We can always pass a literal or even an expression in call-by-value.

Reference Variables

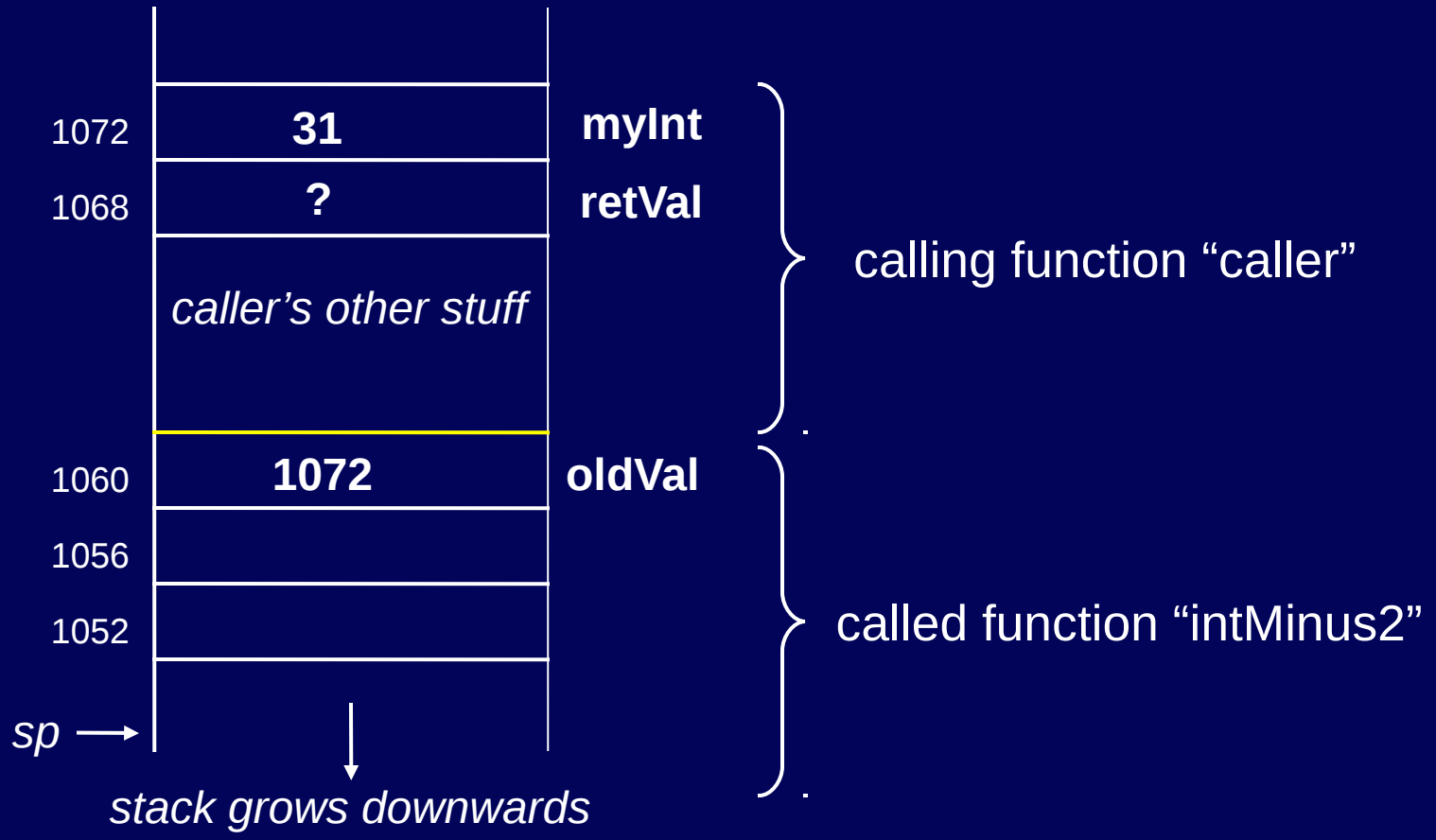
- If the programmer wanted to actually change a variable's value from within a function, one way would be to send a pointer:

```
void caller()
{
   int retVal;
   int myInt = 31;
   retVal = intMinus2( &myInt );
   cout << myInt << retVal;
}
```

- Call this call-by-pointer.

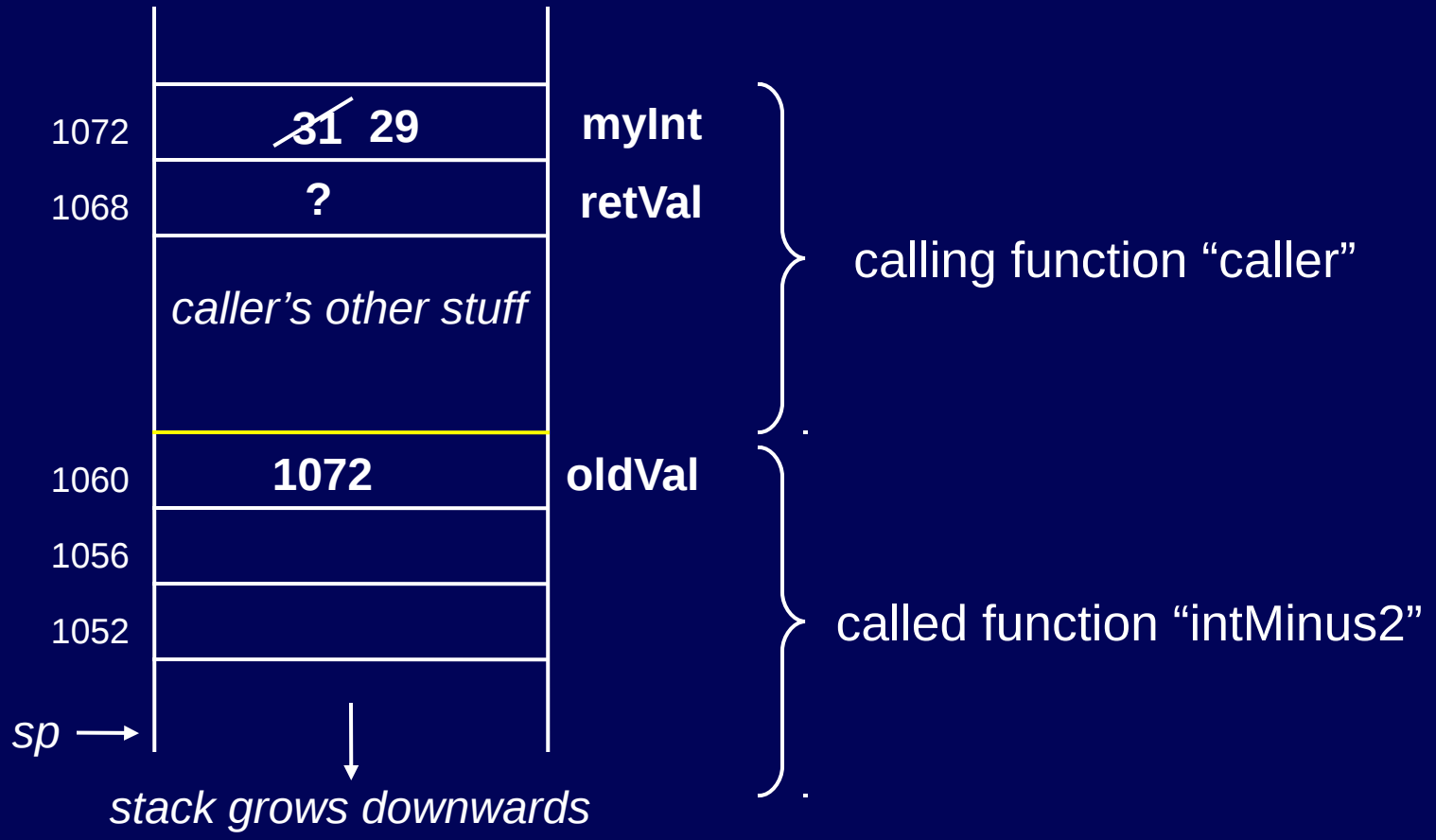
Reference Variables

Call stack layout when `intMinus2` is called:



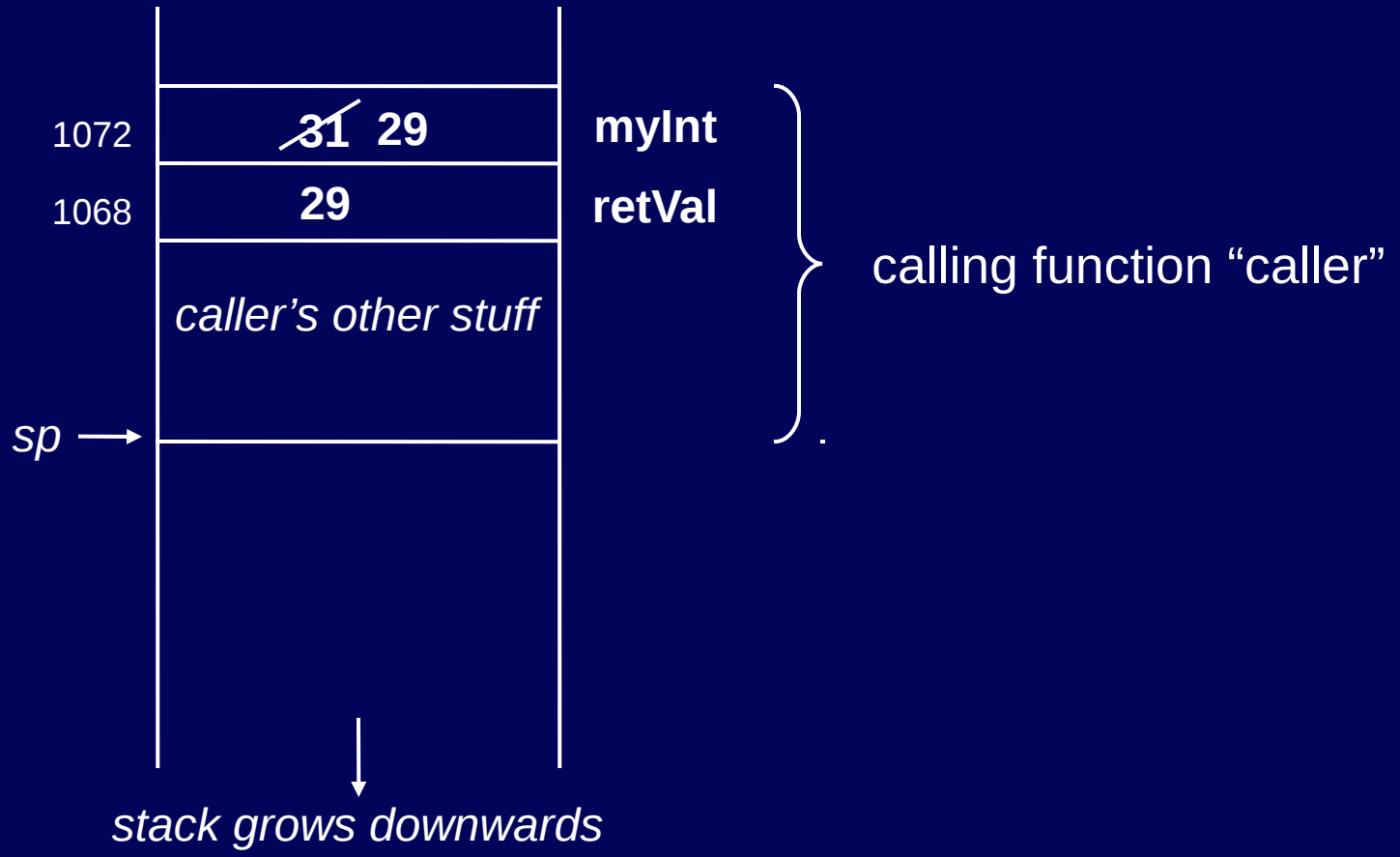
Reference Variables

Call stack layout after **`*oldVal = *oldVal – 2;`**



Reference Variables

Call stack layout after return from **`intMinus2.`**



Reference Variables

- Suppose we want a function to change an object.
- But we don’t want to send the function a copy. The object could be large and copying it costs time.
- We don’t want to use pointers because of the messy syntax.

Reference Variables

- The answer: *call-by-reference* (or pass-by-reference):

```
void caller()
{
        int retVal;
        int myInt = 31;
        retVal = intMinus3( myInt );
        cout << myInt << retVal;
}
```

Reference Variables

- The & after `int` means that `oldVal` is an integer reference variable.

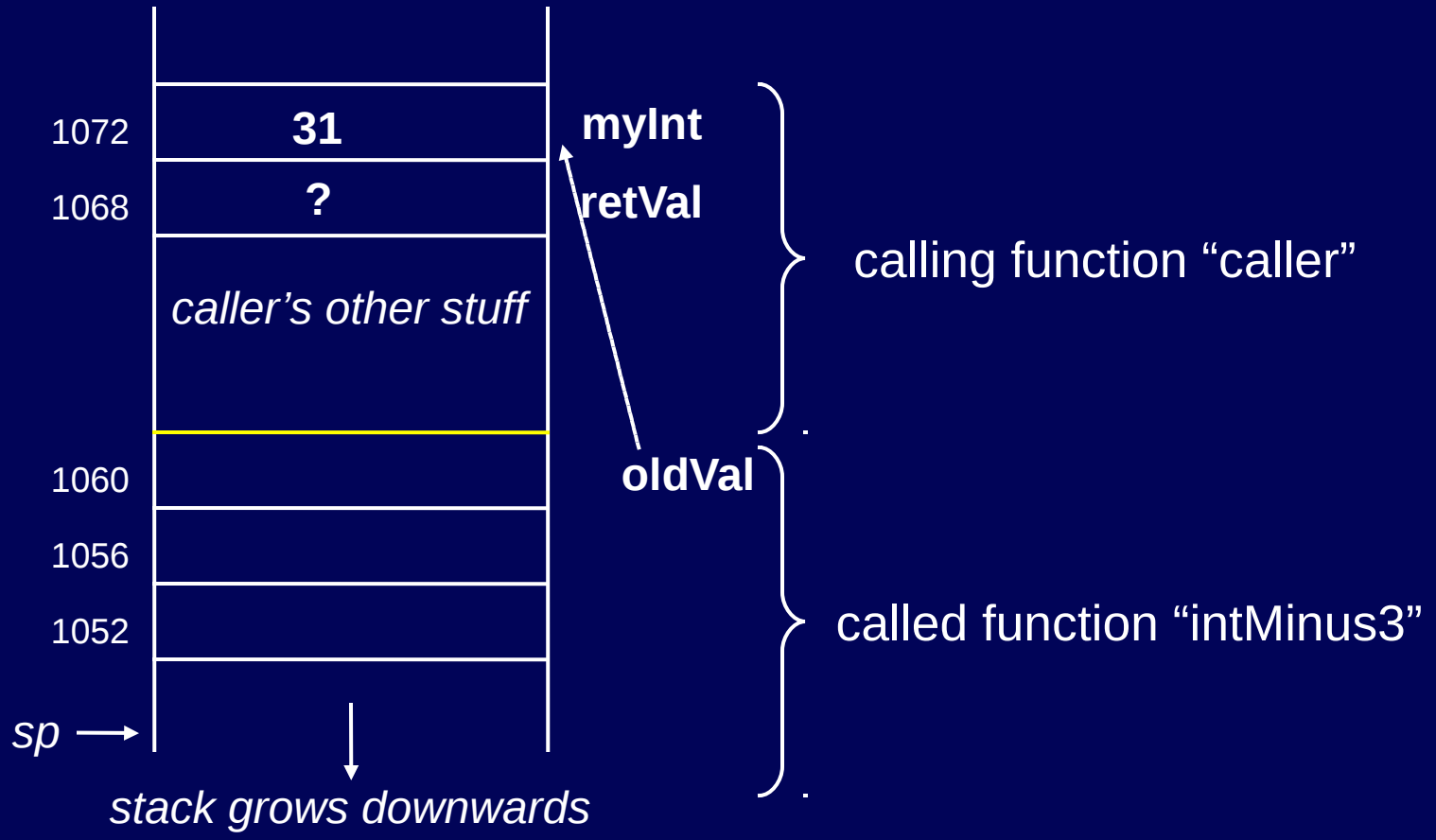
```
// example 3
int intMinus3( int& oldVal)
{
   oldVal = oldVal – 3;
   return oldVal;
}
```

Reference Variables

- So what *is* a reference variable?
- The idea is: the integer object `myInt` is used exactly as it exists in the caller. The function simply reaches it through a *different name*, `oldVal`.
- The function `intMinus3` cannot use the name `myInt` because it is in the caller's scope.
- But both variable names refer to the same object (same memory cell).

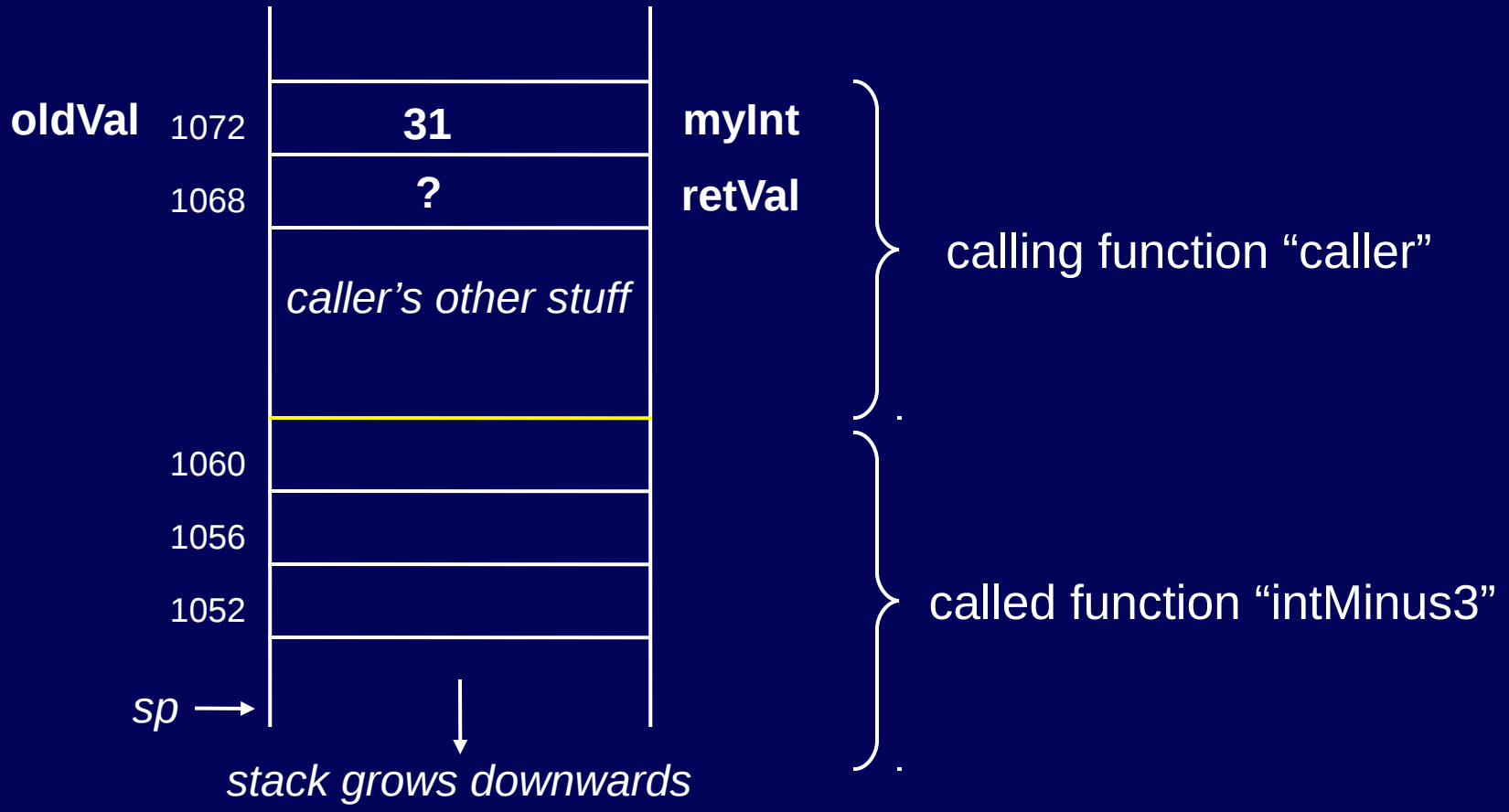
Reference Variables

Call stack layout when `intMinus3` is called:



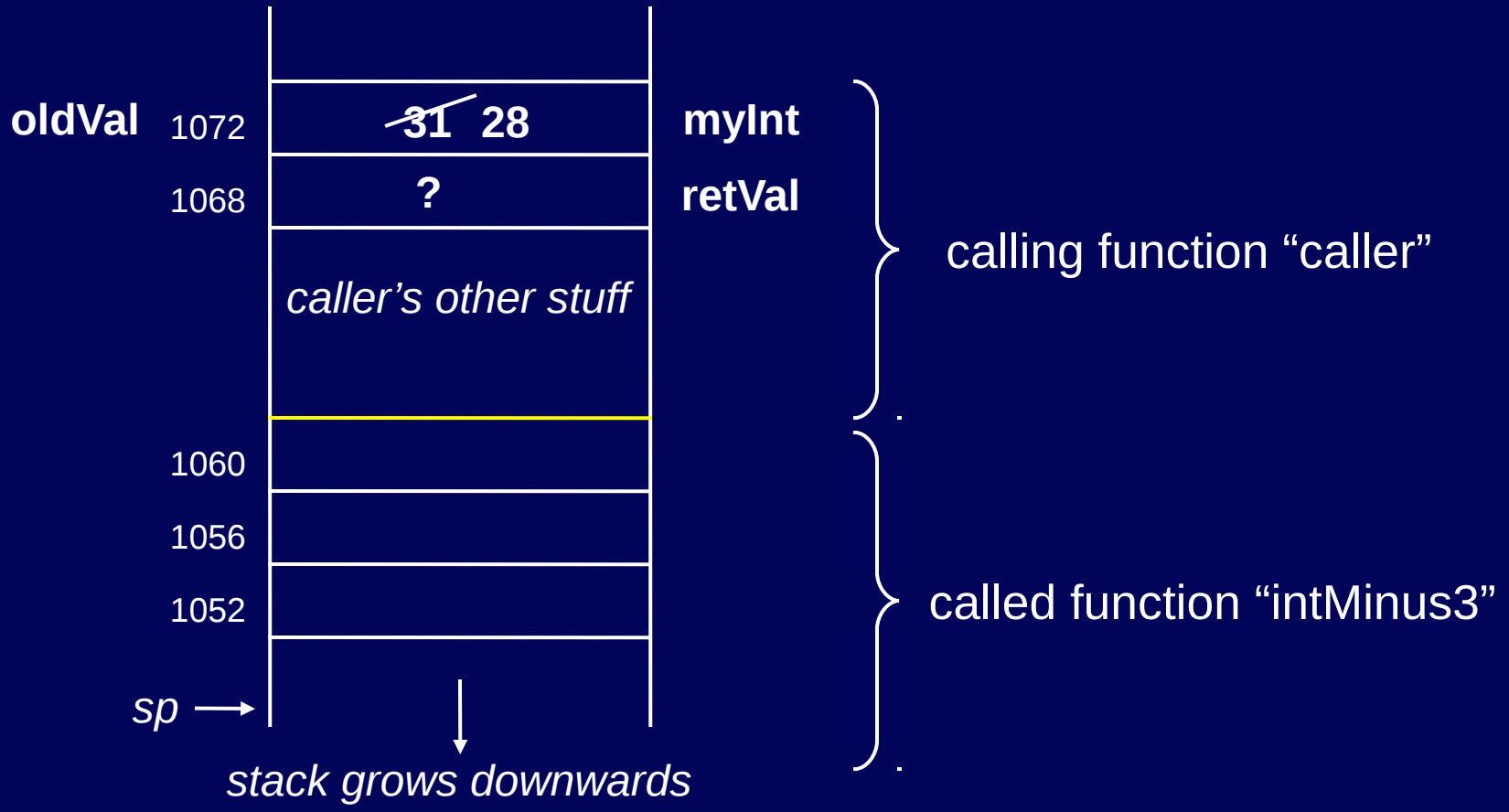
Reference Variables

Call stack layout when `intMinus3` is called:



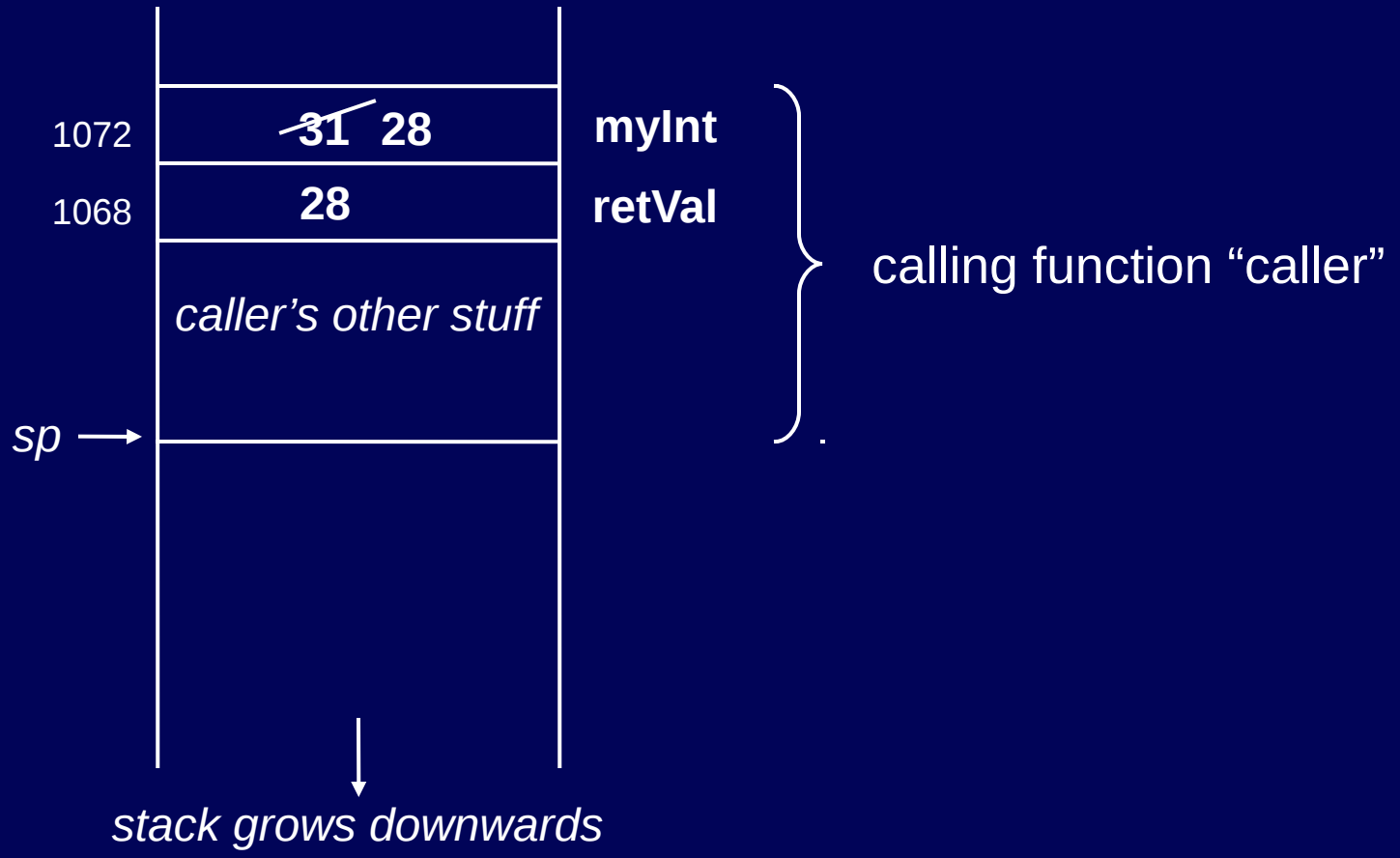
Reference Variables

Call stack layout after `oldVal = oldVal - 3`:



Reference Variables

Call stack layout after return from `intMinus3`:



Reference Variables

- The compiler may actually implement call-by-reference using pointers as we did in example 2.
- The obtaining of address and de-referencing would be done behind the scene.
- We should think in terms of the “renaming” abstraction.

Reference Variables

- One should be careful about transient objects that are stored by reference in data structures.
- Consider the following code that stores and retrieves objects in a queue.

Reference Variables

```
void loadCustomer( Queue& q)
{
  Customer c1(“irfan”);
  Customer c2(“sohail”;
  q.enqueue( c1 );
  q.enqueue( c2 );
}
```

Reference Variables

```
void serviceCustomer( Queue& q)
{
  Customer c = q.dequeue();
  cout << c.getName() << endl;
}
```

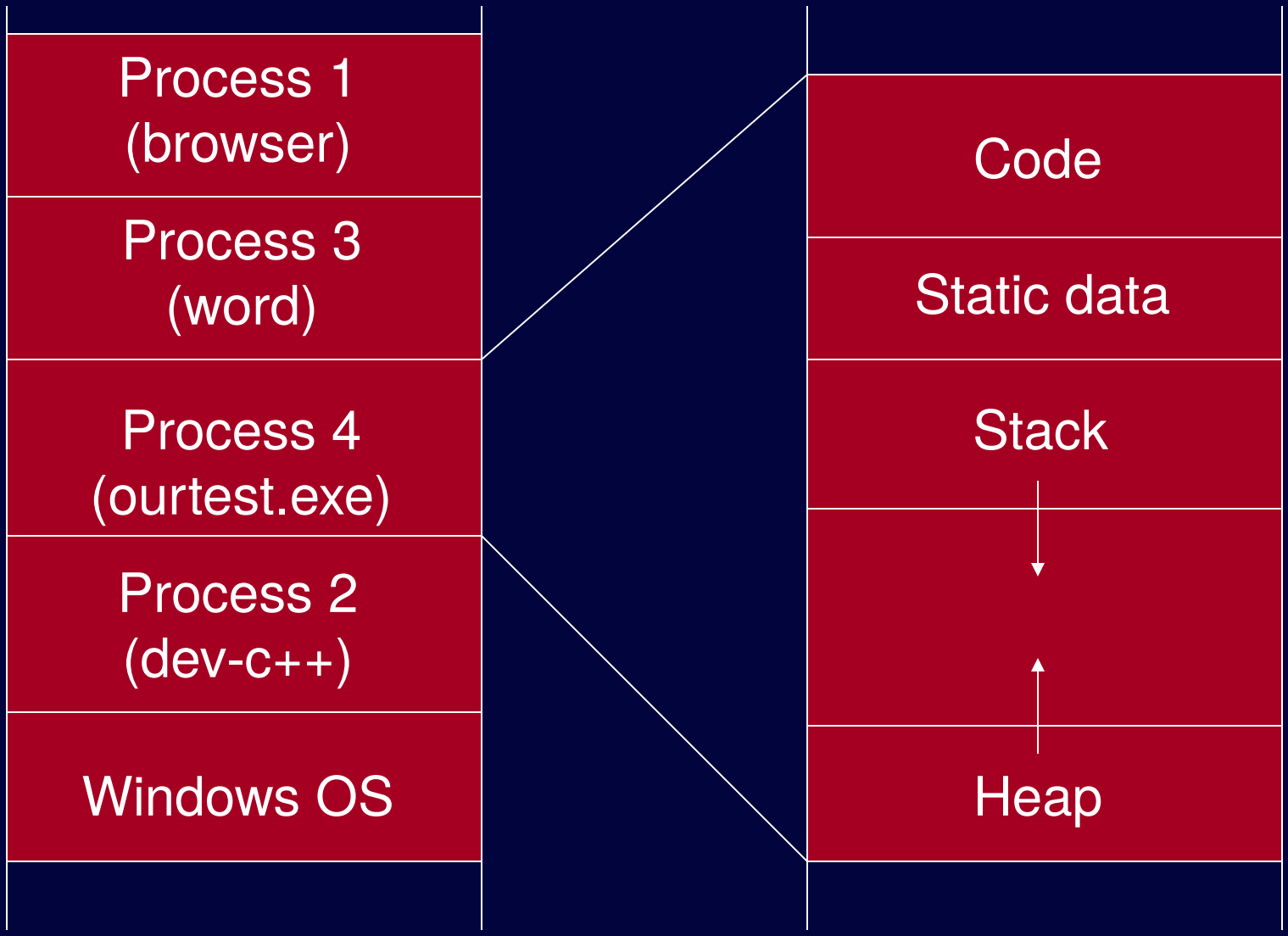
- We got the reference but the object is gone!
- The objects were created on the call stack. They disappeared when the `loadCustomer` function returned.

Reference Variables

```
void loadCustomer( Queue& q)
{
  Customer* c1 = new Customer(“irfan”);
  Customer* c2 = new Customer(“sohail”;
  q.enqueue( c1 ); // enqueue takes pointers
  q.enqueue( c2 );
}
```

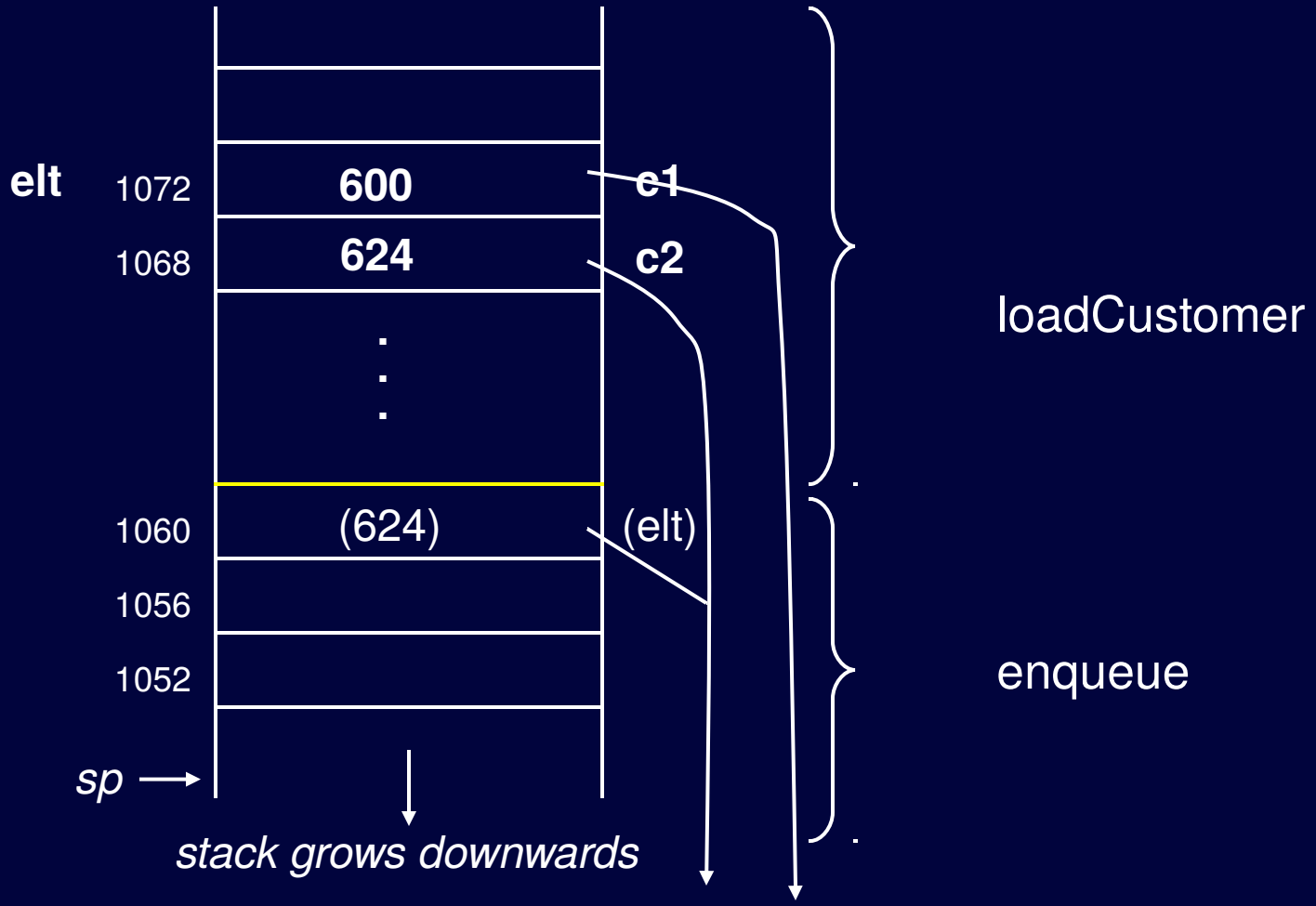
- The pointer variables c1 and c2 are on the call stack. They will go but their contents (addresses) are queued.
- The Customer objects are created in the heap. They will live until explicitly deleted.

Memory Organization



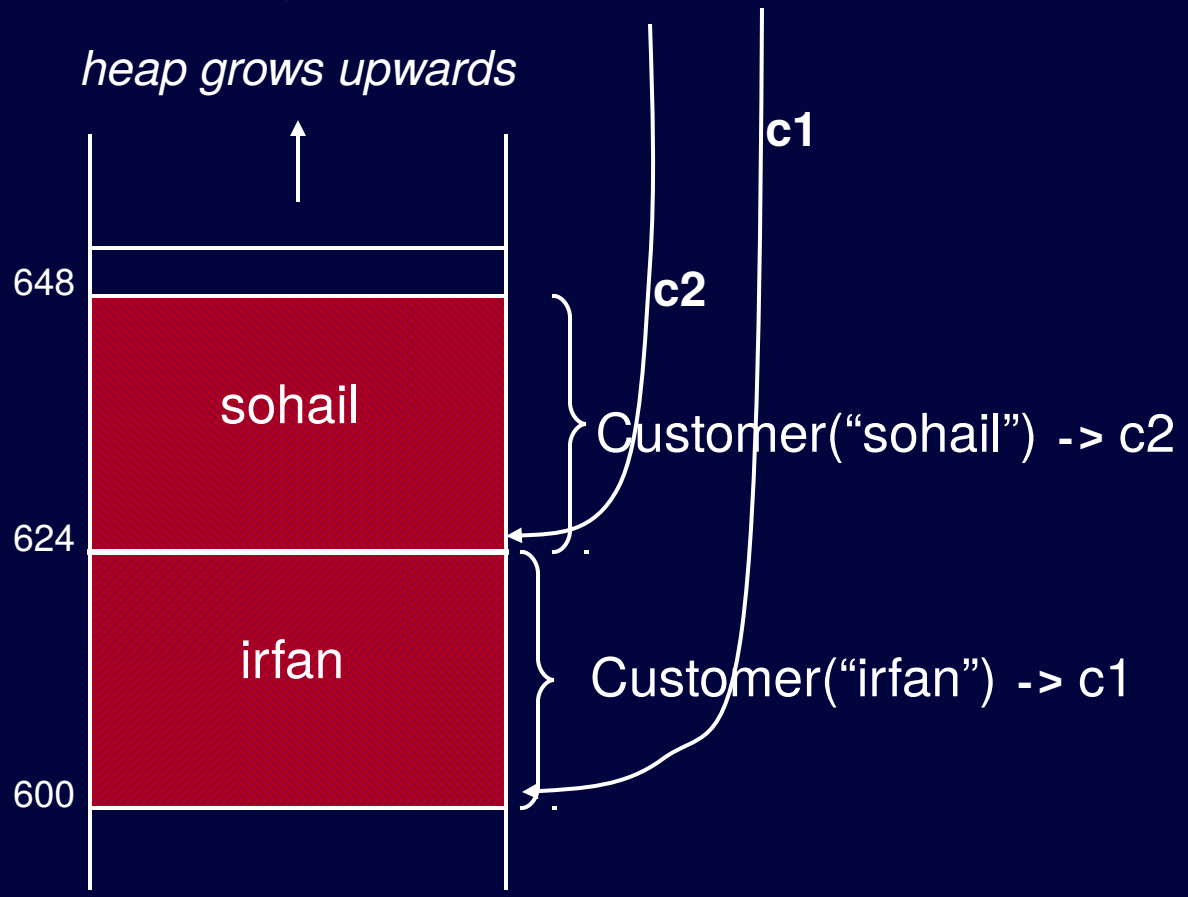
Reference Variables

Call stack layout when q.enqueue(c2) called in loadCustomer.



Reference Variables

Heap layout during call to loadCustomer.



Reference Variables

```
void serviceCustomer( Queue& q)
{
  Customer* c = q.dequeue();
  cout << c->getName() << endl;
  delete c; // the object in heap dies
}
```

- Must use the c-> syntax because we get a pointer from the queue.
- The object is still alive because it was created in the heap.

The `const` Keyword

- The `const` keyword is often used in function signatures.
- The actual meaning depends on where it occurs but it generally means something is to held constant.
- Here are some common uses.

The `const` Keyword

- *Use 1*: The `const` keyword appears before a function parameter. E.g., in a chess program:

```
int movePiece(const Piece& currentPiece)
```

- The parameter must remain constant for the life of the function.
- If you try to change the value, e.g., parameter appears on the left hand side of an assignment, the compiler will generate and error.

The `const` Keyword

- This also means that if the parameter is passed to another function, that function must not change it either.
- Use of const with reference parameters is very common.
- This is puzzling; why are we passing something by reference and then make it constant, i.e., don't change it?
- Doesn't passing by reference mean we *want* to change it?

The `const` Keyword

- The answer is that, yes, we don't want the function to change the parameter, but neither do we want to use up time and memory creating and storing an entire copy of it.
- So, we make the original object available to the called function by using pass-by-reference.
- We also mark it constant so that the function will not alter it, even by mistake.

The `const` Keyword

- *Use 2*: The `const` keyword appears at the end of class member's function signature:

```
EType& findMin( ) const;
```

- Such a function cannot change or write to member variables of that class.
- This type of usage often appears in functions that are suppose to read and return member variables.

The `const` Keyword

- *Use 3*: The `const` keyword appears at the beginning of the return type in function signature:

```
const EType& findMin( ) const;
```

- Means, whatever is returned is constant.
- The purpose is typically to protect a reference variable.
- This also avoids returning a copy of an object.