CS2141 – Software Development using C/C++

# The Memory Model

# Memory Management

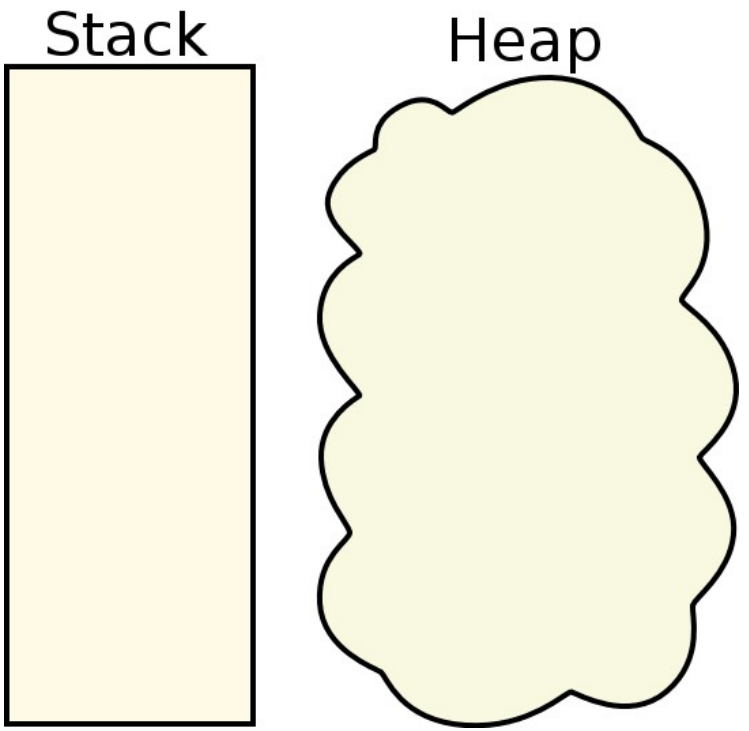
- C++ leaves memory management mostly up to the programmer
  - This makes it possible to write programs that use memory very efficiently
  - This also makes it possible to write programs that waste memory or do not work at all
- Writing efficient (or even working!) programs in C++ requires an understanding of the memory model and memory management operations

# Memory Management Errors

- Errors caused by poor memory management:
  - Using a variable before it has been initialized
  - Allocating memory for storage and not deleting it
  - Using a value after it has been deleted
- To avoid these errors:
  - Always initialize variables
  - Always free memory when done with it
  - Always be sure the memory is no longer in use before deleting it

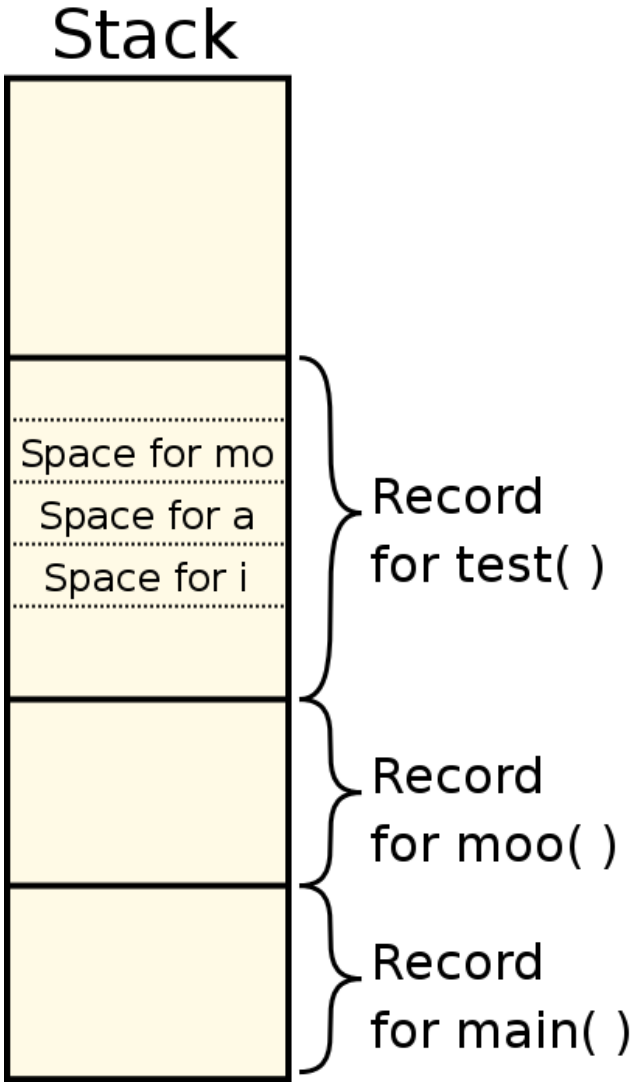
# The Memory Model

- Memory is divided into the *stack* and the *heap*.
- Memory values are then *stack-resident* or *heap-resident*.
  - Stack-resident values are created and freed when a function starts and ends.
  - Heap-resident values are created using the **`new`** operator and freed using the **`delete`** operator.



# Stack Resident Values

- When a function starts, an *activation record* for that function is put on the stack.
  - The activation record contains, among other things, the parameters and local variables of the function.
  - Local variables are anything not created with the **new** operator.



# Stack Resident Values cont.

- Consider the following code:

```
void test( float d )
{
   int i;
   int a[10];
   int * b = new int[20];
   myObject mo;
   myObject * jo = new MyObject( );
   ...
}
```

- What is on the stack?

# Stack-resident Values cont.

- Good:  Can be allocated and freed quickly
- Bad:  The size of stack-resident values must be known at compile time
- Good and bad:  The *lifetime* of stack-resident values is limited, but very predictable
  - Values are allocated when a function starts, and cease to exist when the function returns
  - Trying to use a stack-resident value after its function exits typically leads to errors

# A Lifetime Error

- What's wrong here?

```
char * fullLine( ) {
   char buffer[256]; // Declare a buffer.
   char ch;
   int i;
   cin.get( ch );    // Reads a single char
   for( i = 0; ch != '\n' && i < 255; ++i )
   {
      buffer[i] = ch;
      cin.get( ch );
   }
      buffer[i] = '\0';
   return buffer;     // Return the line.
}
```

# A Lifetime Error cont.

- Once the function returns, the contents of **`buffer`** will be gone
- Possible ways around this:
  - A global buffer (not recommended)
  - Pass the array in by reference:

```
char * fullLine( char * buffer )
```

  - Declare the buffer using new:

```
char * buffer = new char[256];
```

# Size Error 1

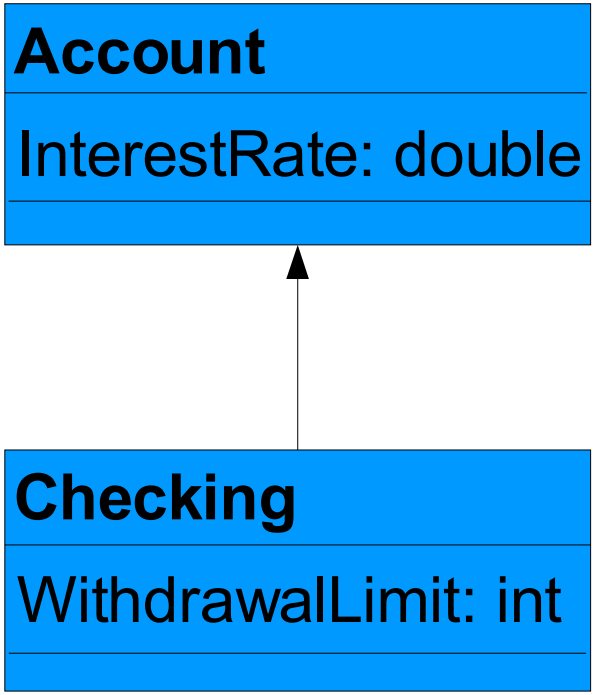
- With arrays, be careful not to overwrite other values on the stack by going past the ends:

```
char buffer[256];   // Global array
char * brokenReadLine( )
{
   gets( buffer );  // Don't use gets!
   return buffer;
}
```

- Since **`gets()`** has no idea how big **`buffer`** is, it could easily overwrite other data following **`buffer`** in memory

# Size Error 2 (Slicing)

- Class *Checking* extends class *Account*, so class *Account* has two data members

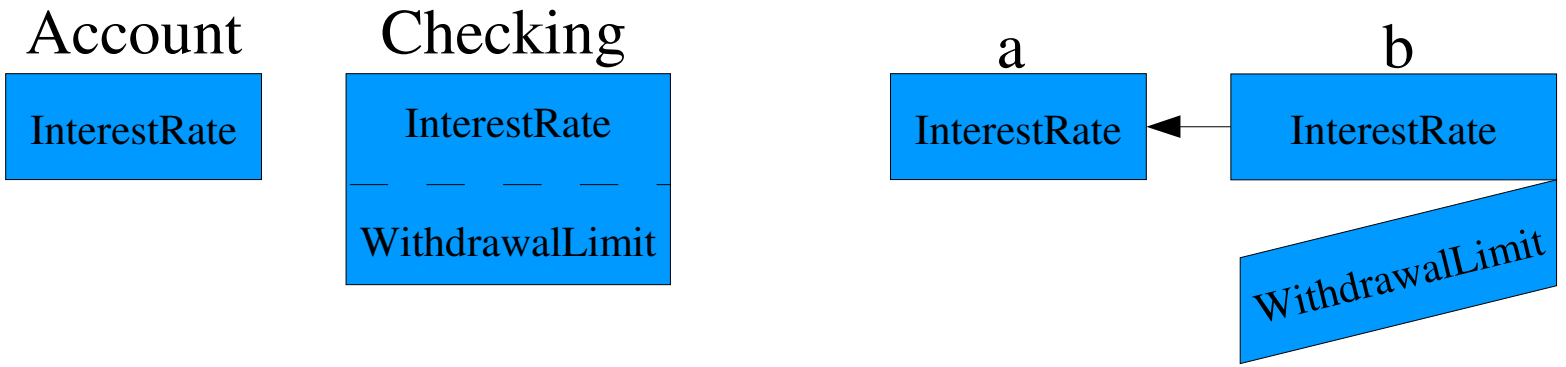


# Size Error 2 cont.

- Suppose we have the following code:

```
Checking b;
Account a = b;
```

- When **b** is assigned to **a**, the members of **b** that are not part of class **Checking** will be sliced off:




- Slicing only happens with stack-resident objects!

# Heap-resident Values

- The *heap*, or *free store*, is separate from the stack
- Unlike the stack, the creation and destruction of heap values is entirely up to the programmer
  - Heap values are allocated with **`new`** operator
  - The **`delete`** operator is used to free heap values
  - Use the **`delete[]`** operator to free arrays. This ensures that the destructor gets called for all objects in the array, not just the first one

# Heap-resident values cont.

- Heap-resident values are typically accessed through a pointer, which is often on the stack.
- Checking for memory allocation errors is left up to the programmer:

```
void doThings( )
{
   float * data = new float[150];
   if( data == 0 )   // new will return 0
      return;        // if an error occured
   ...
   delete[] data;
}
```

# Heap Errors

- Forgetting to allocate the memory and using a pointer as if it had been initialized:

```
char * text;
text[5] = 'q';  // oops!
```

- The compiler is not obligated to check for this.
- Using delete on the same memory multiple times:

```
delete [] text;
...
delete [] text;
```

- Sometimes this goes undetected and causes no harm, other times the program crashes.

# Heap Errors cont.

- Forgetting to free unused memory:

```
int * oops;
for( int i = 0; i < 500; i++ )
{
   oops = new int[1000];
   ... // Stuff that doesn't delete oops
}
```

  - This is sometimes called a *memory leak*.
  - Memory leaks are not always harmful, but if a program is running for a long time or allocating lots of memory very quickly, it could run out of memory.

# Heap Errors cont.

- Attempting to use memory after deleting it:

```
float * scores = new float[35];
...
delete [] scores;
...
scores[7] = 93.2;  // oh no!
```

  - Usually caused by too intensively trying to avoid memory leaks.
  - Usually this is a fatal error.

# Avoiding Heap Errors

- A simple rule: Every time a programmer uses the new operator, he should be able to identify the situations when the associated delete should be issued
- Two common ways to avoid errors:
  - Hide the memory allocation in an object, making the object the owner of that heap-resident memory.
    - When the object is destroyed, it should delete that memory
    - If the object is stack-resident, the lifetime of the heap-resident value is then easily predictable
  - Maintain a reference count as part of the value

# Example 1

```
class Storage {
   public:
   Storage( int s ) { space = new int[s]; }
   int & operator[]( int i ) { return space[i]; }
   ~Storage( ) { delete [] space; }
   private:
   int * space;
};

void doThings( int n )
{                                  // The array allocated
   Storage data( n );              // by data will be
   data[n - 1] = 8;                // deleted when the
}                                  // function returns.
```

# Example 2

```
class Storage {
   public:
   Storage(int s) : {refCount = 0;space = new int[s];}
   int refCount;
   int * space;
};

Storage * data = new Storage( 25 );
data->refCount++;
...
// Only delete it if there are no references to it
if( data->refCount == 0 )
   delete data.space;
```