Copy Control and Memory Management

Overview

- Types of memory
- Copy constructor, assignment operator, and destructor
- Reference counting with smart pointers

References

Static Read-Write Memory

• C++ allows for two forms of global variables:
  • Static non-class variables,
  • Static class variables.

• Static variables are mapped to the global memory. Access to them depends on the visibility specifies.

• We can find a program’s global memory in the so-called read-write .data segment.
The Keyword static

- The keyword **static** can be used to
  - mark the **linkage** of a variable or function **internal**,  
  - retain the **value** of a local variable between function calls,  
  - declare a **class instance variable**,  
  - define a **class method**.
Static class variables must be initialized outside the class.
In combination with the const specifier we can also define read-only global variables or class variables:

```cpp
const int gCounter = 1;
static const int gLocalCounter = 0;

class A
{
private:
    static const int ClassACounter;
};

const int A::ClassACounter = 1;
```
Program Memory: Stack

- All value-based objects are stored in the program’s stack.
- The program stack is automatically allocated and freed.
- References to stack locations are only valid when passed to a callee. References to stack locations cannot be returned from a function.
Stack Frames (C)

- incoming arguments
- stack pointer
- outgoing arguments

- temporaries
- save registers
- arguments
- return address

previous frame

- higher addresses

current frame

next frame
Program Memory: Heap

- Every program maintains a heap for dynamically allocated objects.
- Each heap object is accessed through a pointer.
- Heap objects are not automatically freed when pointer variables become inaccessible (i.e., go out of scope).
- Memory management becomes essential in C++ to reclaim memory and to prevent the occurrences of so-called memory leaks.
List::dropFirst()

```cpp
void dropFirst()
{
    if ( fTop != (ListImpl*)0 )
    {
        ListImpl lNode = fTop;
        fTop = fTop->next;
        if ( fTop == (ListImpl*)0 )
            fLast = (ListImpl*)0;

        delete lNode;
        fCount--;
    }
}
```

Release memory associated with Node object on the heap.
The Dominion Over Objects

- Alias control is one of the most difficult problems to master in object-oriented programming.
- Aliases are the default in reference-based object models used, for example, in Java and C#.
- To guarantee uniqueness of value-based objects in C++, we are required to define copy constructors.
The Copy Constructor

- Whenever one defines a new type, one needs to specify implicitly or explicitly what has to happen when objects of that type are copied, assigned, and destroyed.

- The copy constructor is a special member, taking just a single parameter that is a const reference to an object of the class itself.
```cpp
class SimpleString
{
    private:
        char * fCharacters;

    public:
        SimpleString();
        ~SimpleString();
        SimpleString& operator+(const char aCharacter);
        const char* operator*() const;
};
```
SimpleString: Constructor & Destructor

```cpp
#include <iostream>
#include "SimpleString.h"

using namespace std;

SimpleString::SimpleString()
{
    fCharacters = new char[1];
    *fCharacters = '\0';
}

SimpleString::~SimpleString()
{
    delete fCharacters;
}
```
```cpp
SimpleString& SimpleString::operator+( const char aCharacter )
{
    char *Temp = new char[strlen(fCharacters) + 2];
    unsigned int i = 0;

    for ( ; i < strlen( fCharacters ); i++ )
        Temp[i] = fCharacters[i];

    Temp[i++] = aCharacter;
    Temp[i] = '\0';

    delete fCharacters;
    fCharacters = Temp;
    return *this;
}

const char* SimpleString::operator*() const
{
    return fCharacters;
}
```
Implicit Copy Constructor

```cpp
int main()
{
    SimpleString s1;
    s1 + 'A';
    SimpleString s2 = s1;
    s2 + 'B';

    cout << "S1: " << *s1 << endl;
    cout << "S2: " << *s2 << endl;

    return 0;
}
```

```
Sela:Hit3303 Markus$ ./SimpleString
S1: A
S2: AB
SimpleString(13417) malloc: *** error for object 0x100170: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Sela:Hit3303 Markus$
```
What Has Happened?

- **Shallow copy:**
  \[s2.fCharacters == s1.fCharacters\]

- **Double free:**
  `delete s2.fCharacters`, which was called in `s2 + 'B'`. 
We need an explicit copy constructor!

class SimpleString
{
private:
    char * fCharacters;

public:
    SimpleString();
    ~SimpleString();
    SimpleString( const SimpleString& aOtherString );
    SimpleString& operator+=( const char aCharacter );
    const char* operator*() const;
};
The Explicit Copy Constructor

- When a copy constructor is called, then all instance variables are uninitialized in the beginning.
Explicit Copy Constructor in Use

```cpp
int main()
{
    SimpleString s1;
    s1 + 'A';
    SimpleString s2 = s1;
    s2 + 'B';
    cout << "S1: " << *s1 << endl;
    cout << "S2: " << *s2 << endl;
    return 0;
}
```
What Has Happened?

```cpp
int main()
{
    SimpleString s1;
    s1 + 'A';
    SimpleString s2 = s1;
    s2 + 'B';

    cout << "S1: " << *s1 << endl;
    cout << "S2: " << *s2 << endl;

    <delete s1;>
    <delete s2;>

    return 0;
}
```

Deep copy: `s2.fCharacters != s1.fCharacters`
That’s it. No more problems, or?
A Simple Assignment

```cpp
int main()
{
    SimpleString s1;
s1 + 'A';
SimpleString s2 = s1;
s2 + 'B';

s1 = s2;
cout << "S1: " << *s1 << endl;
cout << "S2: " << *s2 << endl;

return 0;
}
```

Output:
```
S1: AB
S2: AB
SimpleString(13522) malloc: *** error for object 0x100180:
pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
```
What Has Happened?

Shallow copy:
s2.fCharacters == s1.fCharacters

Double free:
delete s2.fCharacters, which is the same as s1.fCharacters.
Rule Of Thumb

• Copy control in C++ requires three elements:
  • a copy constructor
  • an assignment operator
  • a destructor

• Whenever one defines a copy constructor, one must also define an assignment operator and a destructor.
We need an explicit assignment operator!
When the assignment operator is invoked, then all instance variables are initialized in the beginning. We need to release the memory first!
Explicit Assignment Operator in Use

```cpp
int main()
{
    SimpleString s1;
    s1 += 'A';
    SimpleString s2 = s1;
    s2 += 'B';

    s1 = s2;
    cout << "S1: " << *s1 << endl;
    cout << "S2: " << *s2 << endl;

    return 0;
}
```

Terminal output:
```
Sela:HIT3303 Markus$ ./SimpleString
S1: AB
S2: AB
Sela:HIT3303 Markus$ 
```
What Has Happened?

```cpp
int main()
{
    SimpleString s1;
    s1 += 'A';
    SimpleString s2 = s1;
    s2 += 'B';

    s1 = s2;
    cout << "S1: " << *s1 << endl;
    cout << "S2: " << *s2 << endl;

    <delete s1;>
    <delete s2;>

    return 0;
}
```

Deep copy: $s2.fCharacters \neq s1.fCharacters$
Cloning: Alias Control for References
Copying Pointers

```cpp
int main()
{
    SimpleString* ps1 = new SimpleString();
    (*ps1) + 'A';
    SimpleString* ps2 = ps1;
    (*ps2) + 'B';

    cout << "S1: " << **ps1 << endl;
    cout << "S2: " << **ps2 << endl;

    delete ps1;
    delete ps2;

    return 0;
}
```

```
Sela:HIT3303 Markus$ ./SimpleString
S1: AB
S2: AB
SimpleString(13660) malloc: *** error for object 0x100170: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
SimpleString(13660) malloc: *** error for object 0x100100: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Sela:HIT3303 Markus$ _
```
What Has Happened?

Shallow copy:

```
ps2. == ps1
```

Double free:
delete ps2, which is the same as ps1.
Solution: A clone() Method

- It is best to define the destructor of a class virtual always in order to avoid problems later.
The Use of clone()

```cpp
SimpleString* SimpleString::clone()
{
    return new SimpleString(*this);
}
```

```cpp
int main()
{
    SimpleString* ps1 = new SimpleString();
    (*ps1) + 'A';
    SimpleString* ps2 = ps1->clone();
    (*ps2) + 'B';
    cout << "S1: " << **ps1 << endl;
    cout << "S2: " << **ps2 << endl;
    delete ps1;
    delete ps2;
    return 0;
}
```
Problems With Cloning

- The member function `clone()` must be defined `virtual` to allow for proper redefinition in subtypes.
- Whenever a class contains a virtual function, then its destructor is required to be defined `virtual` as well.
- The member function `clone()` can only return one type. When a subtype redefines `clone()`, only the super type can be returned.
Non-virtual Cloning Does Not Work!

- One could define clone() non-virtual and use overloading. But this does not work as method selection starts at the static type of the pointer.

```cpp
SimpleString* pToString = new SubtypeOfSimpleString();
SimpleString* c1 = pToString->clone(); // SimpleString::clone()
```
Reference-based Semantics: When Do We Destroy Objects?
Reference Counting

• A simple technique to record the number of active uses of an object is reference counting.

• Each time a heap-based object is assigned to a variable the object’s reference count is incremented and the reference count of what the variable previously pointed to is decremented.

• Some compilers emit the necessary code, but in case of C++ reference counting must be defined (semi-)manually.
Smart Pointers: Handle

template<class T>
class Handle
{
  private:
    T* fPointer;
    int* fCount;

    void addRef();
    void releaseRef();

  public:
    Handle( T* aPointer = (T*)0 );
    Handle( const Handle<T>& aOtherHandle );
    ~Handle();

    Handle& operator=( Handle<T>& aOtherHandle );
    T& operator*();
    T* operator->();
};
The Use of Handle

• The template class Handle provides a pointer-like behavior:
  • Copying a Handle will create a shared alias of the underlying object.
  • To create a Handle, the user will be expected to pass a fresh, dynamically allocated object of the type managed by the Handle.
  • The Handle will own the underlying object. In particular, the Handle assumes responsibility for deleting the owned object once there are no longer any Handles attached to it.
Handle: Constructor & Destructor

Create a shared counter

Decrement reference count
Handle: addRef & releaseRef

Increment reference count

Decrement reference count and delete object if it is no longer referenced anywhere.
Handle: Copy Control

```cpp
Handle< T >& aOtherHandle )
{
    fPointer = aOtherHandle.fPointer;
    fCount = aOtherHandle.fCount;
    addRef();           // increment use
}

Handle& operator=( Handle< T >& aOtherHandle )
{
    aOtherHandle.addRef();    // increment use
    releaseRef();             // release old handle
    fPointer = aOtherHandle.fPointer;
    fCount = aOtherHandle.fCount;
    return *this;
}
```
Handle: Pointer Behavior

```
T& operator*()
{
    if (fPointer)
        return *fPointer;
    else
        throw std::runtime_error("Dereference of unbound handle!");
}

T* operator->()
{
    if (fPointer)
        return fPointer;
    else
        throw std::runtime_error("Access through unbound handle!");
}
```
Using Handle

```cpp
int main()
{
    Handle<SimpleString> hs1( new SimpleString() );
    *hs1 += 'A';
    Handle<SimpleString> hs2( hs1->clone() );
    *hs2 += 'B';
    Handle<SimpleString> hs3 = hs1;

    cout << "HS1: " << **hs1 << endl;
    cout << "HS2: " << **hs2 << endl;
    cout << "HS3: " << **hs3 << endl;

    return 0;
}
```

Sela: HIT3303 Markus$ ./SimpleString
HS1: A
HS2: AB
HS3: A
Sela: HIT3303 Markus$
Reference Counting Limits

- Reference counting **fails** on circular data structures like double-linked lists.
- Circular data structures require extra effort to reclaim allocated memory. **Know solution: Mark-and-Sweep**