Constructor and Destructor examples:

```cpp
void func () {

    UCSDStudent stu, stu2(“Tracker”), * stup;
    // stu, stu2 ___________________
    // stu is initialized by ___________________
    // stu2 is initialized by ___________________
    // stup is ___________________:
        // like ___________________;       // in Java

    stup = new UCSDStudent;
    // calls ___________________ for a ______ object

    stup = new UCSDStudent ();      // ___________

    stup->number = 10; // accessing field via pointer
    stu.number = 10; // accessing field via an object

    delete stup;       // call to ___________________
}                     // end of scope…call to ___________________
// on ___________________________

<discussion of differences of where to allocate objects>
```

Optional parameters:
What: any number of trailing parameters passed to a function that provides defaults values for parameters not found when calling function.
Where: Can be any function…not just constructors.

Guaranteed initialization:
What: special syntax for constructors.
Why: to initialize data fields
Syntax: data fields are listed in a comma separated list in the order declared with each initial value listed (in parenthesis) after a colon before open curly of the constructor.

Example:
Constructor (void) : datafield (init_value), datafield2 (initial_value), datafield3 (init_value) { …}

// normally: in UCSDStudent.h:
class UCSDStudent {  // first section is private (default)
    char name [20];   // private field
    long number;      // private field

public:
    UCSDStudent (void) { } // ____________
    UCSDStudent (char * nm, long nmbr = 0) :
        number (nmbr) {
            strcpy (name, nm); // G.I. can’t initialize arrays
        }
    ~UCSDStudent (void) { } // ____________

    void setNumber (UCSDStudent &);  // ____________
};  // semi-colon is ______________
Scope Resolution:
  Scope: where a symbol is __________
  Resolution: to resolve ___________

Syntax:
  C++: ClassName :: symbol_to_resolve
  Java: ClassName.symbol_to_resolve

Uses:
  C++/Java: ________________________
  C++: __________________________

Member methods defined outside class definition:

// normally: in UCSDStudent.c:
void UCSDStudent:: setNumber (UCSDStudent & stur) {
    number = stur.number;  //this->number = stur.number
    stur.number = 0;
}

main () {

    UCSDStudent stu ("Tracker", 123), stu2;
    stu2.setNumber (stu);  // pass by _________
    // stu’s number = 0…changed _________
}

Think about for future:
  1. Why define a method outside/inside its class definition?
  2. Why/when to use guaranteed initialization syntax?
Operator overloading:
What: Defining a meaning for use of existing operators for user defined types (also for system defined non-primitive types).
When: Works only with OBJECTS.
   - Comparing pointers is still comparing pointers.

Ex:

```c
void func () {
    UCSDStudent stu1, stu2;  // RTS objects
    ...
    if (stu1 == stu2) {...  // overloaded == call
    ...
}
```

Translates:
if (__________________)…// non-member method
or
    if (__________________)…// member method

Which one is really called at run-time?
Ans:
______________________________.

We can define an overloaded operator as either:
- __________
- __________
- __________
Casting overloading:
What: operator overloading involving casting operators.
Why: Defines the meaning of casting your object to a primitive type.
How: Casting overloads have no return type.
   - Why: _______________________
When: Called implicitly or explicitly.
Uses: Accessor methods.
Ex:
   class UCSDStudent {
   ...
    public:
    operator char * (void) {
       return name;
    
   }
   }

void func () {
   char * pointer; // local variable
   UCSDStudent stu;    // RTS
   ...
    pointer = stu;      // implicit call to char *
   // above is a contrived example
   // you will use operator char * to
   // display the name of the UCSDStudent
   // in your debug messages.
   
}
More on implicit conversion:

    int xxx = 3.5;  // xxx gets 3
    float fff = 10; // fff gets 10.0

    int xxx = (int) 3.5;  // explicit conversion

Hierarchy exists in the language for implicit conversion:
- lowest level of conversion is convert to a
  - __________________

    while (cin) {  // what is going on?

    // ___________________________

      }

    }

same as:

Istreams: used to read input in C++
Ex:
    cin >> some_variable;

Ostreams: used to display objects to the user in C++
Ex:
    cout << “Hello World\n”;
Combining istreams and ostreams:

Ex:

```cpp
#include <iostream>
char buffer[BUFSIZ];
long number;
cout << "Please enter the student name";
cin >> buffer;
cout << "Please enter the student number";
cin >> number;
```

Above are not “cin commands” and “cout commands”
- “cout” is ____________________________
- “cin” is ____________________________

cout << "Hello World\n";

is the same as:

___________________

What is the prototype (signature) for operator<<?
- Parameters?
- Return value?

Example:

cout << "Hello World" << "\n";

which translates to:

______________________________

⇒ must return ________
- New instance of ________?
- ___: we must return ________________________!
Order of expression evaluation to better understand operator `<<`:

```cpp
int func (int xxx) {
    return xxx;
}

aaa = func (1+2) + 3;  // Original expression.
```

Is “6;” a reasonable line of code?

Consider:
```cpp
aaa = bbb = ccc = 10;
```

Apply these same ideas to `cout` and operator `<<`:
```cpp
ST.Write (cout << “The table is”) << “\n”,;  // Original expression.
```
For your assignment:

operator const char *:
Use: To display the name of the UCSDStudent in your debug messages:

    cerr << (char *) stu;       // in Driver code
or
    cerr << (char *) *(table[index]);   // in HashTable code
or
    cerr << (char *) * element;    // in HashTable code

To account for stderr message in a diff script:

    ./Driver -x < alphabet >& my.out

Using >& will redirect both stdout and stderr to the same output.
Discussion

HW6:

Sample executables in public area: Calc1, Calc2 and Driver1, Driver2

Look at Base:
- Note the missing/meaningless function bodies.
- They are going to be called inside of your HashTable code.

Look at Variable:
- Note the meaningful function bodies.

Look at the virtual methods:
- Note the virtual call
  - The function to the method of the unknown object.
  - The compiler only knows about Base.
  - At run-time the child object’s method is called.

Sample object: UCSDStudent for the Driver
- Refer to Variable for your design of UCSDStudent.
- It will be SIMILAR, not identical.

Your job for hw6 coding:
- UCSDStudent
- Empty HashTable methods

Constructor:
- don’t forget to allocate the array: table and collision flags
  - table = ______________; // allocates the table
  - next is a loop of initialization to null
    - The HashTable algorithm depends upon null table locations.
Locate:
What: The engine of the HashTable.
   - The code that would have been in both Insert, Lookup, if you didn’t have a Locate method.
     o Eliminating code duplication.
How:
   - All the hashing goes there.
     o Getting the numeric attribute (call to hashCode)
     o Increment
     o Loop of checking locations
   - sets “index” to be used by Insert and Lookup

<table>
<thead>
<tr>
<th>Locate returns true</th>
<th>Locate returns false</th>
<th>Locate returns true</th>
<th>Locate returns false</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The importance of the Lookup/Remove methods:
We are looking for more information than we had.

The probe sequence (list of index values for an item) is not stored!
   - It’s just the value of index in your hashing loop inside of Locate.
   - Lookup does not have a loop.
   - Insert does not have a loop.