

cis15
advanced programming techniques, using c++
lecture # 11.2

topics:

- ctors and dtors
- polymorphism and overloading
- friend classes, composition and derivation

resources:

- Pohl, chapter 5 (mostly sections 5.1-5.3, 5.7, 5.10)

ctors and dtors

- *an object is a class instance* (a house metaphor: the blueprint for the house is like a class; the constructed house is like an object)
- *run-time stack system*: a system of memory allocation commonly used on most computers where a *stack* — the data structure that we discussed last week — is used to keep track of how much memory is available to a program and allocates pieces of it as they are needed; when a function is called, memory required for the function (e.g., its local variables) is allocated from (*pushed onto*) the stack; when the function exits, the memory is freed from (*popped off*) the stack
- in the case of a class, the allocation of memory to create (instantiate) an object is called *construction*; freeing memory (aka deallocation) when the program is done using the object is called *destruction*
- a *ctor* (*constructor*) is a member function used to allocate the memory required by an object
- a *dtor* (*destructor*) is a member function used to deallocate (free) the object's memory, after the object is no longer needed

- the C++ keyword `new` is used to invoke the constructor
- the C++ keyword `delete` is used to invoke the destructor (freeing this memory is also called finalizing or destroying an object)
- constructors can be *overloaded* (i.e., programmers can write their own versions); destructors cannot
- constructors can take arguments; destructors cannot
- ctors and dtors do not have data types; they do not return values

ctors and dtors: constructor example

- example from book:

```
class counter {
public:
    counter( int i ); // ctor declaration
    void reset() { value = 0; }
    int get() const { return value; }
    void print() const { cout << value << '\t'; }
    void click() { value = (value+1) % 100; }
private:
    int value; // 0 to 99
}
// constructor definition:
inline counter::counter( int i ) { value = i % 100; }
```

- remember `inline`: means that the compiler can try to replace the function call by the function body code; this avoids function call invocation and can speed up program execution; but `inline` isn't required here, nor is it required by constructors in general

constructor details

- the default constructor
 - the default constructor takes no arguments
 - you can overload the default constructor with or without arguments of your own
- constructor initializer
 - you can use a constructor to initialize class data members
 - example:
instead of:

```
counter::counter( int i ) { value = i % 100; }
```


you can use a constructor initializer like this:

```
counter::counter( int i=0 ) : value(i%100) { }
```
 - the syntax is as follows:
member-name (expression-list)
where each member is initialized to the expression in parenthesis
- constructors as conversions

– example:

```
#include <iostream>
using namespace std;

class pr_char {
public:
    pr_char( int i=0 ) : c( i % 5 ) { }
    void print() const { cout << rep[c]; }
private:
    int c;
    static const char* rep[5];
};

const char* pr_char::rep[5] = { "a", "b", "c", "d", "e" };

int main() {
    pr_char c;
    for ( int i=0; i<5; i++ ) {
        c = i; // NOTE
    }
}
```

```
        c.print();
        cout << endl;
    }
}
```

- in the line with NOTE, the constructor is called implicitly to convert the integer to `pr_char`
- this isn't necessarily good practice and only works where the constructor is initializing one data element
- except when the keyword `explicit` precedes the constructor definition (see example later)

point example with constructors

```
class point {
public:
    point() : x(0), y(0) { } // default constructor
    point( double u ) : x(u), y(0) { } // convert double to point
    point( double u, double v ) : x(u), y(v) { }
    void print() const;
    void set( double u, double v );
private:
    double x, y;
};
```

stack example with constructors

```
#include <iostream>
using namespace std;

class ch_stack {
public:
    explicit ch_stack( int size ) : max_len(size), top(EMPTY) {
        assert(size > 0);
        s = new char[size];
        assert( s != 0 );
    } // end of ctor()
    void reset() { top = EMPTY; }
    void push( char c ) { s[++top] = c; }
    char pop() { return s[top--]; }
    char top_of() const { return s[top]; }
    bool empty() const { return( top==EMPTY ); }
    bool full() const { return( top==max_len-1 ); }
private:
```

cis15-ozgelen-lectII.2

9

```
enum{ EMPTY = -1 };
char *s;
int max_len;
int top;
};

int main() {
    ch_stack s(200);
    char str[40] = { "hello world!" };
    int i = 0;
    cout << "str=" << str << endl;
    s.reset();
    while( str[i] && ! s.full() ) {
        s.push( str[i++] );
    }
    cout << "reversed str=";
    while ( ! s.empty() ) {
        cout << s.pop();
    }
    cout << endl;
}
```

cis15-ozgelen-lectII.2

10

```
} // end of main()
```

- note use of `explicit` keyword in constructor: this prevents implicit use of constructor as a converter, which is what we want since the constructor does other things besides just initialize `max_len`
- the `assert()` function is used to test whether a pointer value is 0 (null) — the program will not continue if the `assert` condition is false

cis15-ozgelen-lectII.2

11

copy constructors

- this is a somewhat complicated detail that has to do with what happens when an object is used as a call-by-value argument to a function
- we mentioned briefly about the use of the run-time stack and how memory is allocated and deallocated when functions are called
- when the arguments to functions are primitive data types (e.g., `int`), then this is easy
- but when the arguments to functions are objects, what happens locally inside the function? how is a "local copy" made for use inside the function?
- this is where a *copy constructor* is needed
- and is defined by using a call-by-value argument to a version of a constructor

```
ch_stack::ch_stack( const ch_stack& stk ) :
    max_len(stk.max_len), top(stk.top) {
    s = new char[stk.max_len];
    assert( s != 0 );
    memcpy( s, stk.s, max_len );
} // end of copy ctor()
```

cis15-ozgelen-lectII.2

12

destructors

- defined as the name of the class preceded by a tilde (~)
- example:

```
class ch_stack {
public:
    ...
    ~ch_stack() {
        delete []s;
    }
private:
    ...
};
```

polymorphism and overloading

- *polymorphism*—giving different meanings to the same function or operator, i.e., “having many forms”; lets us use different implementations of a single class
- *overloading*—creating new versions of functions with the same or different arguments
- when you overload a function, the name of the function is the same, but what it does is different from the default
- operators can also be overloaded
- *signature matching* is what the compiler uses when there are multiple versions of a function (or operator) to determine which version should be invoked
- book goes into a LOT of detail about this—we’ll come back to it more later in the term

friend classes

- allows two or more classes to share private members
- e.g., container and iterator classes
- friendship is not transitive

```
class tweedledee {
    ...
    friend class tweedledum;

    int cheshire();
    ...
};
```

friend functions

- friendship can also be at the individual function level—at the class level, all functions are friend functions
- at the function level, a non-member function can have access to the private components in a class
- example:

```
void alice() {
    ...
}

class tweedledee {
    ...
    friend void alice(); // prototype for friend function
    int cheshire(); // member function
    ...
};
```

hierarchy with composition and derivation

- composition:
 - creating objects with other objects as members
- derivation:
 - defining classes by expanding other classes
 - like “extends” in java
 - example:

```
class SortIntArray : public IntArray {
    public:
        void sort();
    private:
        int *sortBuf;
}; // end of class SortIntArray
```
 - “base class” (IntArray) and “derived class” (SortIntArray)
 - derived class can only access public members of base class

– public vs private derivation:

- * public derivation means that users of the derived class can access the public portions of the base class
- * private derivation means that all of the base class is inaccessible to anything outside the derived class
- * private is the default

derivation, continued.

- encapsulation
 - derivation maintains encapsulation
 - i.e., it is better to expand IntArray and add sort() than to modify your own version of IntArray
- friendship
 - not the same as derivation!!
 - example:
 - * b2 is a friend of b1
 - * d1 is derived from b1
 - * d2 is derived from b2
 - * b2 has special access to private members of b1, as a friend
 - * but d2 does not inherit this special access
 - * nor does b2 get special access to d1 (derived from friend b1)

composition, derivation and friend classes: example

```
#include <iostream>

using namespace std;

enum raceclass{ f1, nascar };

class baggage {
public:
    friend class car; // 'car' members can access to private member 'load'
    baggage( int i=0 ) : load(i) {};
    friend void carry( baggage& bg, int load ); // 'carry' has access
    void print() const { cout << "current load: " << load << endl; }
private:
    int load;
};
```

```

class car {
public:
    car ( int sp = 0, int acc = 10 ) : speed(sp), acceleration(acc) {};
    bool forward(){ speed += acceleration; }
    bool reverse() { speed -= acceleration; }
    bool hitbreak() { speed = 0; }
    void setSpeed( int s ) { speed = s; }
    baggage& getBaggage() { return bg; }
    void setLoad( int l ) { bg.load = l; }
    void print() const { cout << "current speed: " << speed << endl; }
    void printLoad() const { bg.print(); }
private:
    int speed;
    int acceleration;
    baggage bg;
};

void carry ( baggage& bg, int load ) {
    bg.load +=load;
}

```

```

// derived from 'car'. a sportsCar object 'IS A' car object as well.
//Therefore can use it's public members.
class sportsCar : public car {
public:
    sportsCar( raceclass r = nascar ) : rc(r) {}
    bool fullThrottle() {
        if ( rc == f1 )
            setSpeed ( 300 );
        else if ( rc == nascar )
            setSpeed ( 250 );
    }
private:
    int rc;
};

```

```

int main() {
    car c1;
    sportsCar s1(f1);

    carry(c1.getBaggage(),10);
    c1.printLoad(); // this function accesses private 'load'

    c1.forward();
    c1.print();

    s1.forward(); // Using member function of car class
    s1.print();
    s1.fullThrottle();
    s1.print();

    return 0;
}

```