CONSTRUCTORS AND DESTRUCTORS

Today

- Today we will look *constructors* and *destructors*.
- These are important additional concepts in handling classes and objects.
- We will also briefly cover *polymorphism* and *overloading*, and mention friend classes, composition and derivation.
- This material is taken from Pohl, Chapter 5, mainly 5.1–5.3, 5.7 and 5.10.

ctors and dtors

- An *object* is a *class instance*.
- House metaphor: the blueprint for the house is like a class; the constructed house is like an object)
- 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 constructor always has the same name as the class it constructs.
- 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.
- 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; }
```


- `inline` is (another) new keyword.
- It means that the compiler can try to replace the function call by the function body code; this avoids function call invokation and can speed up program execution;
- `inline` isn't required here, nor is it required by constructors in general

Constructor details

- The default constructor
  - The default constructor takes no arguments
  - If you don't define one, the system creates the default.
  - 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.

- For 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 can be used to convert data from one type to another.
- For 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)
- By default, any constructor with a single argument is assumed to be a conversion constructor.

"point" example with constructors

```
class point {
public:
  point() : x(0), y(0) { }           // default constructor
  point( double u ) : x(u), y(0) { } // conversion.
  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) {
    s = new char[size];
  } // 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:
  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;
} // end of main()
```


- Note the use of `explicit` keyword in constructor.
- This prevents implicit use of constructor as a converter.
- This what we want since the constructor does other things besides just initialize `max_len`.

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.

- This is defined by using a call-by-value argument to a version of a constructor
- For example:

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

Destructors

- Defined as the name of the class preceded by a tilde (~)
- The default destructor will delete an object when the program reaches the end of the scope of that object (block where it is declared).
- You can write your own destructor to free up additional memory used by the object.
- Example, free up the array used by the stack:

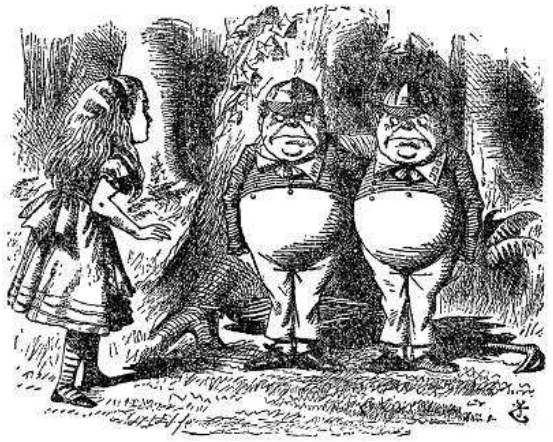
```
class ch_stack {
public:
.
.
~ch_stack() { delete []s; }
private:
.
.
}
```


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 is 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
- Textbook goes into a LOT of detail about this—we'll come back to it more later in the semester.

Friend classes

- Allows two or more classes to share private members and functions
  - e.g., container and iterator classes
- Friendship is not transitive.
- Since friendship violates the usual rules about hiding members, you need to use it with care.



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

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

- This allows any instance of `tweedledum` to access any member of any instance of `tweedledee`.
- However no instance of `tweedledee` can access any private member of `tweedledum`.

Friend functions

- Friendship can also be at the individual function level.
- A non-member friend function can have access to the private components in a class.

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

class tweedledum {
...
  friend void alice()   // prototypes for friend functions
  friend int tweedledee::cheshire ();
...
};
```

- This allows `alice` and `cheshire` to access the data in instances of `tweedledee`.

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`)
- "Derived class" (`SortIntArray`)
- Derived class can only access public members of base class
- Complete example: `array5.cpp`
- `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 and friendship

- Friendship is not the same as derivation!
- Example:
  - $b2$ is a friend of $b1$
  - $d1$ is derived from $b1$
  - $d2$ is derived from $b2$
- In this case:
  - $b2$ 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.h>
using namespace std;

class IntArray {
  friend class IntArrayIter;
public:
  void init();
  void cleanup();
  void setSize( size_t value );
  size_t getSize();
  void setElem( size_t index, int value );
  int getElem( size_t index );
private:
  int *elems;
  size_t numElems;
}; // end of class IntArray
```



```
class IntArrayIter {
public:
  void init();
  void cleanup();
  void setIter( IntArray *array );
  void goFirst();
  void goNext();
  int getCur();
  int curIsValid();

private:
  IntArray *array;
  size_t curItem;
}; // end of class IntArrayIter
```



```
class StatsIntArray : public IntArray {
public:
  int mean();
private:
  int *buf;
}; // end of class StatsIntArray

// IntArray member functions
void IntArray::init() {
  numElems = 0;
  elems = 0;
} // end of IntArray::init()
```

```
void IntArray::cleanup() {
  free( elems );
  elems = 0;
  numElems = 0;
} // end of IntArray::cleanup()

void IntArray::setSize( size_t value ) {
  if ( elems != 0 ) {
    free( elems );
  }
  numElems = value;
  elems = (int *)malloc( value * sizeof( int ));
} // end of IntArray::setSize()
```



```
size_t IntArray::getSize() {
  return( numElems );
} // end of IntArray::getSize()

void IntArray::setElem( size_t index, int value ) {
  if (( index < -0 ) || ( index >= numElems )) {
    cerr << "bad index";
    exit( 1 );
  }
  elems[index] = value;
} // end of IntArray::setElem()

int IntArray::getElem( size_t index ) {
  if (( index < -0 ) || ( index >= numElems )) {
    cerr << "bad index";
    exit( 1 );
  }
  return( elems[index] );
} // end of IntArray::getElem()
```



```
// IntArrayIter member functions

void IntArrayIter::init() {
  array->init();
} // end of IntArrayIter::init()

void IntArrayIter::cleanup() {
  array->cleanup();
  array = 0;
} // end of IntArrayIter::cleanup()

void IntArrayIter::setIter( IntArray *array ) {
  this->array = array;
  goFirst();
} // end of IntArrayIter::setIter()

void IntArrayIter::goFirst() {
  curItem = 0;
} // end of IntArrayIter::goFirst()
```



```
void IntArrayIter::goNext() {
  curItem++;
} // end of IntArrayIter::goNext()

int IntArrayIter::getCur() {
  if ( ! curIsValid() ) {
    cerr << "no current item\n";
    exit( 1 );
  }
  return curItem;
} // end of IntArrayIter::getCur()

int IntArrayIter::curIsValid() {
  return (( array != 0 ) &&
          ( curItem >= 0 ) &&
          ( curItem < array->numElems ));
} // end of IntArrayIter::curIsValid()
```

```
// StatsIntArray member functions

int StatsIntArray::mean() {
  int total = 0;
  IntArrayIter iter;
  iter.setIter( this );
  for ( iter.goFirst(); iter.curIsValid(); iter.goNext() ) {
    total += getElem( iter.getCur() );
  }
  return( total / getSize() );
} // end of StatsIntArray::mean()
```



```
int main() {
  StatsIntArray powersOf2;
  IntArrayIter iter;
  powersOf2.init();
  powersOf2.setSize( 8 );
  powersOf2.setElem( 0, 1 );
  iter.setIter( &powersOf2 );
  iter.goFirst();
  for ( iter.goNext(); iter.curIsValid(); iter.goNext() ) {
    powersOf2.setElem( iter.getCur(), 2*powersOf2.getElem( iter.getCur()-1 ));
  }
  cout << "here are the elements:\n";
  for ( iter.goFirst(); iter.curIsValid(); iter.goNext() ) {
    cout << "powerOf2=" << powersOf2.getElem( iter.getCur() ) << "\n";
  }
  cout << "mean = " << powersOf2.mean() << "\n";
  iter.cleanup();
  powersOf2.cleanup();
}
```


Summary

- This lecture has looked at:
  - Donstructors and destructors
  - Polymorphism, overloading
  - Friends
  - Composition and derivation
- For most of these topics, it has been a first look; we will come back to them over and over again through the semester.