Constructors and overloading

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: January 28, 2025

Function overloading, type coercion, operator overloading

Multiset ADT

Multiset ADT

```
typedef int ItemType;

class Multiset
{
public:
  // Default constructor
  Multiset();

  bool empty() const;
  bool full() const;
  unsigned int memberCount() const;
  void insert(ItemType x);
  void remove(ItemType x);
  bool member(ItemType x) const;
  void print() const;

private:
  unsigned int data_count;
  ItemType data[MAX_MEMBERS];
};
```

Default constructors (revisited)

- Default constructor written by the programmer

Default constructors (revisited)

- Default constructor written by the programmer
- constructor creates an empty Multiset

Default constructors (revisited)

- Default constructor written by the programmer
- constructor creates an empty Multiset

Default constructors (revisited)

- Default constructor written by the programmer
- constructor creates an empty Multiset

```
class Multiset {
public:
  Multiset();
  ...
};
```

- Default constructor provided by the compiler

Default constructors (revisited)

- Default constructor written by the programmer
- constructor creates an empty Multiset

```
class Multiset {
public:
  Multiset();
  ...
};
```

- Default constructor provided by the compiler
  - Client code: `Multiset m;`

Default constructors (revisited)

- Default constructor written by the programmer
- constructor creates an empty Multiset

```
class Multiset {
public:
  Multiset();
  ...
};
```

- Default constructor provided by the compiler
  - Client code: `Multiset m;`
  - but not `Multiset m( );`

Declaring another constructor

- Want to insert all elements of an array A of size n into Multiset

Declaring another constructor

- Want to insert all elements of an array A of size n into Multiset

Declaring another constructor

- Want to insert all elements of an array A of size n into Multiset

```
int A[5] = { 2, 5, 4, 3, 1 };
Multiset m;
for (int i = 0; i < 5; i++)
  m.insert(A[i]);
```

- If frequently done, might as well write a constructor

Declaring another constructor

- Want to insert all elements of an array A of size n into Multiset

```
int A[5] = { 2, 5, 4, 3, 1 };
Multiset m;
for (int i = 0; i < 5; i++)
  m.insert(A[i]);
```

- If frequently done, might as well write a constructor

Declaring another constructor

- Want to insert all elements of an array A of size n into Multiset

```
int A[5] = { 2, 5, 4, 3, 1 };
Multiset m;
for (int i = 0; i < 5; i++)
  m.insert(A[i]);
```

- If frequently done, might as well write a constructor

```
class Multiset {
public:
  Multiset();
  Multiset(const ItemType A[], unsigned int n);
  ...
};
```

Using and Implementing the constructor

- Client code

Using and Implementing the constructor

- Client code

Using and Implementing the constructor

- Client code

```
int A[5] = { 2, 5, 4, 3, 1 };
Multiset m(A, 5);  // Invoking the constructor with an array arg
// followed by an integral argument
```

- Implementation

Using and Implementing the constructor

- Client code

```
int A[5] = { 2, 5, 4, 3, 1 };
Multiset m(A, 5);  // Invoking the constructor with an array arg
// followed by an integral argument
```

- Implementation
  - Multiset stored as sorted array

Using and Implementing the constructor

- Client code

```
int A[5] = { 2, 5, 4, 3, 1 };
Multiset m(A, 5);  // Invoking the constructor with an array arg
// followed by an integral argument
```

- Implementation
  - Multiset stored as sorted array

Using and Implementing the constructor

- Client code

```
int A[5] = { 2, 5, 4, 3, 1 };
Multiset m(A, 5);  // Invoking the constructor with an array arg
// followed by an integral argument
```

- Implementation
  - Multiset stored as sorted array

```
Multiset::Multiset(const ItemType A[], unsigned int n) {
  assert(n <= MAX_LENGTH);
  data_count = n; // Copy size
  // Copy array
  for (unsigned int i = 0; i < n; i++)
    data[i] = A[i];
  // Sort to normalize representation
  sort(data, data_count);  // e.g., any sorting algorithm
}
```

Yet Another Constructor

- Want to create a Multiset with n copies of the same item x

Yet Another Constructor

- Want to create a Multiset with n copies of the same item x

Yet Another Constructor

- Want to create a Multiset with n copies of the same item x

```
Multiset(ItemType x, unsigned int n);

Client code: Multiset m(999, 5);  // A multiset of 5 copies of 999
```

- implementation

Yet Another Constructor

- Want to create a Multiset with n copies of the same item x

```
Multiset(ItemType x, unsigned int n);

Client code: Multiset m(999, 5);  // A multiset of 5 copies of 999
```

- implementation

Yet Another Constructor

- Want to create a Multiset with n copies of the same item x

```
Multiset(ItemType x, unsigned int n);

Client code: Multiset m(999, 5);  // A multiset of 5 copies of 999
```

- implementation

```
Multiset::Multiset(ItemType x, unsigned int n) {
  data_count = n;
  for (unsigned int i = 0; i < n; i++)
    data[i] = x;
}
```

Other uses of constructors

- Assignments

Other uses of constructors

- Assignments
- Anonymous objects can be useful and efficient

Other uses of constructors

- Assignments
- Anonymous objects can be useful and efficient

Other uses of constructors

- Assignments
- Anonymous objects can be useful and efficient

```
// ordinary variables initialized using default constructor
Counter c1, c2;
// ordinary variable initialized using initializing constructor
Counter c3(0, 3);
// unnamed instance constructed with default constructor
c1 = Counter( );
// unnamed instance constructed with initializing constructor
c2 = Counter(0, 10);

Counter ctr1[MAX];
ctr1[5] = Counter(0,3);
```

Constructor Overloading

- When we define multiple constructors for a class, we say it is *overloaded*

Constructor Overloading

- When we define multiple constructors for a class, we say it is *overloaded*
- The compiler picks the right constructor

Constructor Overloading

- When we define multiple constructors for a class, we say it is *overloaded*
- The compiler picks the right constructor
  - based on the types of the arguments given

Constructor Overloading

- When we define multiple constructors for a class, we say it is *overloaded*
- The compiler picks the right constructor
  - based on the types of the arguments given
  - Can’t have two constructors with the same argument types

Constructors for arguments and return values

- Creating anonymous objects for function call

Constructors for arguments and return values

- Creating anonymous objects for function call

Constructors for arguments and return values

- Creating anonymous objects for function call

```
House h1(500000);
...
House p = h1.add(House(1000000));
```

- Creating anonymous for the purpose of returning it

Constructors for arguments and return values

- Creating anonymous objects for function call

```
House h1(500000);
...
House p = h1.add(House(1000000));
```

- Creating anonymous for the purpose of returning it

Constructors for arguments and return values

- Creating anonymous objects for function call

```
House h1(500000);
...
House p = h1.add(House(1000000));
```

- Creating anonymous for the purpose of returning it

```
House House::add(const House &other) const {
  if (price == 0 && other.price == 0) {
    // return instance made with default constructor
    return House( );
  }
  else
    return House(price + other.price);
}
```

Overloading In General

- Can overload any function, not just constructors

Overloading In General

- Can overload any function, not just constructors
  - Again, correct one chosen by argument types

Overloading In General

- Can overload any function, not just constructors
  - Again, correct one chosen by argument types

Overloading In General

- Can overload any function, not just constructors
  - Again, correct one chosen by argument types

```
int myMax(int a, int b) {
  if (a > b)
    return a;
  else
    return b;
}
float myMax(float a, float b) {
  if (a > b)
    return a;
  else
    return b;
}
int main( ) {
  // invoke myMax(float, float)
  cout << myMax(1.2f, 4.7f);
  // invoke myMax(int, int)
  cout << myMax(3, 4);
  return 0;}
```

Choosing a Function

- Either type or number of args must be different

Choosing a Function

- Either type or number of args must be different
  - How about different return types only? (nope!)

Choosing a Function

- Either type or number of args must be different
  - How about different return types only? (nope!)

Choosing a Function

- Either type or number of args must be different
  - How about different return types only? (nope!)

```
int myMax(int a, int b) {
  if (a > b)
    return a;
  else
    return b;
}

int myMax(int a, int b, int c) {
  return myMax(a, myMax(b, c));
}

int main( ) {
  // invoke myMax(int, int)
  cout << myMax(3, 4);
  // invoke myMax(int, int, int)
  cout << myMax(3, 4, 5);
  return 0;}
```

Overloading Class Member Functions

Overloading Class Member Functions

```
// header
void print( ) const;
void print(ostream &output_stream) const;
// client code
L.print();
L.print(cout);
// definition
void Multiset::print( ) const {
  print(cout);
}
void Multiset::print(ostream& output_stream) const {
  for (unsigned int i = 0; i < data_count; i++) {
    if (i != 0)
      // comma separation for all except the first member
      output_stream << ", ";
    output_stream << data[i];
  }
}
```

Type coercions

- AKA implicit (static or dynamic) type conversion

Type coercions

- AKA implicit (static or dynamic) type conversion
- Occurs when evaluating expressions, passing values to functions, and returning values from functions

Type coercions

- AKA implicit (static or dynamic) type conversion
- Occurs when evaluating expressions, passing values to functions, and returning values from functions
- No warning produced by compiler unless has possibility of information loss

Type coercions

- AKA implicit (static or dynamic) type conversion
- Occurs when evaluating expressions, passing values to functions, and returning values from functions
- No warning produced by compiler unless has possibility of information loss
- Coercion order:

Type coercions

- AKA implicit (static or dynamic) type conversion
- Occurs when evaluating expressions, passing values to functions, and returning values from functions
- No warning produced by compiler unless has possibility of information loss
- Coercion order:
  - `{double <- float <- long int <- int <- short int <- char`

Type coercions

- AKA implicit (static or dynamic) type conversion
- Occurs when evaluating expressions, passing values to functions, and returning values from functions
- No warning produced by compiler unless has possibility of information loss
- Coercion order:
  - `{double <- float <- long int <- int <- short int <- char`
    - No warnings are provided for type upgrade given in the above order

Type coercions

- AKA implicit (static or dynamic) type conversion
- Occurs when evaluating expressions, passing values to functions, and returning values from functions
- No warning produced by compiler unless has possibility of information loss
- Coercion order:
  - `{double <- float <- long int <- int <- short int <- char`
    - No warnings are provided for type upgrade given in the above order
    - “safe” coercion

Type coercions (examples)

Type coercions (examples)

Type coercions (examples)

Type coercions (examples)

```
void myMax(float f1, float f2); // 1A
void myMax(int i1, int i2); // 1B
myMax(7, 9);


void zipIt(float f1); // 2A
void zipIt(string s1); // 2B
String s = "Trouble";
zipIt(s);

void zoom(float f1); // 3A
void zoom(string s1); // 3B
int x = 14;
zoom(x);
```

Type coercions (examples)

```
void myMax(float f1, float f2); // 1A
void myMax(int i1, int i2); // 1B
myMax(7, 9);


void zipIt(float f1); // 2A
void zipIt(string s1); // 2B
String s = "Trouble";
zipIt(s);

void zoom(float f1); // 3A
void zoom(string s1); // 3B
int x = 14;
zoom(x);
```

- 1: None (1B)

Type coercions (examples)

```
void myMax(float f1, float f2); // 1A
void myMax(int i1, int i2); // 1B
myMax(7, 9);


void zipIt(float f1); // 2A
void zipIt(string s1); // 2B
String s = "Trouble";
zipIt(s);

void zoom(float f1); // 3A
void zoom(string s1); // 3B
int x = 14;
zoom(x);
```

- 1: None (1B)
- 2: None (2B)

Type coercions (examples)

```
void myMax(float f1, float f2); // 1A
void myMax(int i1, int i2); // 1B
myMax(7, 9);


void zipIt(float f1); // 2A
void zipIt(string s1); // 2B
String s = "Trouble";
zipIt(s);

void zoom(float f1); // 3A
void zoom(string s1); // 3B
int x = 14;
zoom(x);
```

- 1: None (1B)
- 2: None (2B)
- 3: Safe (3A)

Type coercions (examples ctd.)

Type coercions (examples ctd.)

Type coercions (examples ctd.)

Type coercions (examples ctd.)

```
void whoosh(char c1); // 4A
void whoosh(string s1); // 4B
double pi = 3.14159;
whoosh(pi);

void crunch(string s1, string s2); // 5A
void crunch(string s1); // 5B
double e = 2.71828;
crunch(e);
```

Type coercions (examples ctd.)

```
void whoosh(char c1); // 4A
void whoosh(string s1); // 4B
double pi = 3.14159;
whoosh(pi);

void crunch(string s1, string s2); // 5A
void crunch(string s1); // 5B
double e = 2.71828;
crunch(e);
```

- 4: Unsafe and possibly warning (4A)

Type coercions (examples ctd.)

```
void whoosh(char c1); // 4A
void whoosh(string s1); // 4B
double pi = 3.14159;
whoosh(pi);

void crunch(string s1, string s2); // 5A
void crunch(string s1); // 5B
double e = 2.71828;
crunch(e);
```

- 4: Unsafe and possibly warning (4A)
- 5: Error!

Type coercions (examples ctd.)

Type coercions (examples ctd.)

```
void mixed(int i1, double d1); // 6A
void mixed(double d1, int i1); // 6B
int k3 = 3, k4 = 4;
mixed(k3, k4);

void mixed(int i1, double d1); // 7A
void mixed(double d1, int i1); // 7B
double r5 = 55.5, r6 = 66.6;
mixed(r5, r6);
```

- 6: both safe but ambiguous

Type coercions (examples ctd.)

```
void mixed(int i1, double d1); // 6A
void mixed(double d1, int i1); // 6B
int k3 = 3, k4 = 4;
mixed(k3, k4);

void mixed(int i1, double d1); // 7A
void mixed(double d1, int i1); // 7B
double r5 = 55.5, r6 = 66.6;
mixed(r5, r6);
```

- 6: both safe but ambiguous
- 7: both unsafe and ambiguous

Operator overloading

- `operator` keyword

Operator overloading

- `operator` keyword
  - Gives more than one meaning to the same operator

Operator overloading

- `operator` keyword
  - Gives more than one meaning to the same operator
  - Operands (arguments to operators) are new data types

Operator overloading

- `operator` keyword
  - Gives more than one meaning to the same operator
  - Operands (arguments to operators) are new data types
  - thus, overloading the operator

Operator overloading

- `operator` keyword
  - Gives more than one meaning to the same operator
  - Operands (arguments to operators) are new data types
  - thus, overloading the operator
  - Uses keyword operator

Operator overloading

- `operator` keyword
  - Gives more than one meaning to the same operator
  - Operands (arguments to operators) are new data types
  - thus, overloading the operator
  - Uses keyword operator

Operator overloading

- `operator` keyword
  - Gives more than one meaning to the same operator
  - Operands (arguments to operators) are new data types
  - thus, overloading the operator
  - Uses keyword operator

```
// equality operator
bool operator== (const House &h) const;

// assignment operator
House &operator= (const House &h);
```

Operator overloading (example)

Operator overloading (example)

```
class House {
  string address;
  string owner;
  unsigned int cost;
  bool fireplace;
public:
  // default constructor
  House();

  // initializing constructor

  House(const string &initAddress,
  const string &initOwner,
  unsigned int initCost,
        bool initFireplace);


// copy constructor
House(const House &original);
```

Implementing ==

- Let’s say we want to implement a function called isEquals

Implementing ==

- Let’s say we want to implement a function called isEquals

Implementing ==

- Let's say we want to implement a function called isEquals

```
bool House::isEquals(const House &h) const {
  if (address != h.address) return false;
  if (owner != h.owner) return false;
  if (cost != h.cost) return false;
  if (fireplace != h.fireplace) return false;
  return true;
}
```

- We could have implemented it as follows

Implementing ==

- Let's say we want to implement a function called isEquals

```
bool House::isEquals(const House &h) const {
  if (address != h.address) return false;
  if (owner != h.owner) return false;
  if (cost != h.cost) return false;
  if (fireplace != h.fireplace) return false;
  return true;
}
```

- We could have implemented it as follows

Implementing ==

- Let's say we want to implement a function called isEquals

```
bool House::isEquals(const House &h) const {
  if (address != h.address) return false;
  if (owner != h.owner) return false;
  if (cost != h.cost) return false;
  if (fireplace != h.fireplace) return false;
  return true;
}
```

- We could have implemented it as follows

```
bool House::operator==(const House &h) const {
  ...
    }
```

The == operator

- Can now use it as an operator

The == operator

- Can now use it as an operator

The == operator

- Can now use it as an operator

```
House h1, h2;
...  // initialize fields of h1 and h2

if (h1 == h2) {
  // do something useful
 }
```

Implementing assignment operator (=)

- First attempt:

Implementing assignment operator (=)

- First attempt:

Implementing assignment operator (=)

- First attempt:

```
void House::operator=(const House &h) {
  address = h.address;
  owner = h.owner;
  cost = h.cost;
  fireplace = h.fireplace;
}
```

- All good, works for a = b

Implementing assignment operator (=)

- First attempt:

```
void House::operator=(const House &h) {
  address = h.address;
  owner = h.owner;
  cost = h.cost;
  fireplace = h.fireplace;
}
```

- All good, works for a = b
- But does not allow assignment statements to be chained

Implementing assignment operator (=)

- First attempt:

```
void House::operator=(const House &h) {
  address = h.address;
  owner = h.owner;
  cost = h.cost;
  fireplace = h.fireplace;
}
```

- All good, works for a = b
- But does not allow assignment statements to be chained
- e.g. a = b = c = d won't work

Implementing assignment operator (=)

- First attempt:

```
void House::operator=(const House &h) {
  address = h.address;
  owner = h.owner;
  cost = h.cost;
  fireplace = h.fireplace;
}
```

- All good, works for a = b
- But does not allow assignment statements to be chained
- e.g. a = b = c = d won't work
- for this, need to mutable House type object (i.e. reference)

Implementing assignment operator (=)

- Updated version

Implementing assignment operator (=)

- Updated version

Implementing assignment operator (=)

- Updated version

```
// & is used for efficiency only!
House &House::operator=(const House &h) {
  if (this != &h) {
    address = h.address;
    owner = h.owner;
    cost = h.cost;
    fireplace = h.fireplace;
  }
  return *this;
}
```

- `this` is a pointer to the reference object

Implementing assignment operator (=)

- Updated version

```
// & is used for efficiency only!
House &House::operator=(const House &h) {
  if (this != &h) {
    address = h.address;
    owner = h.owner;
    cost = h.cost;
    fireplace = h.fireplace;
  }
  return *this;
}
```

- `this` is a pointer to the reference object
- `*this` is the “contents” of the reference object

Assignment operator (=)

- Client code

Assignment operator (=)

- Client code

Assignment operator (=)

- Client code

```
House h1, h2, h3;
h1.setCost(500); h2.setcost(700); h3.setCost(900);

h1 = h2 = h3;  // same as h1.operator=(h2.operator=(h3));

h1.printCost(); // prints 900
```

Implementing addition operator (+)

Implementing addition operator (+)

```
House House::operator+ (const House &h) {

  House newHouse;
  newHouse = *this;

  newHouse.address += " + " + h.address;
  newHouse.owner += " + " + h.owner;
  newHouse.cost += h.cost;
  newHouse.fireplace = newHouse.fireplace  h.fireplace;

  return newHouse;
}
```

Implementing increment operator (+=)

Implementing increment operator (+=)

```
House &House::operator+= (const House &h) {

  address += " + " + h.address;
  owner += " + " + h.owner;
  cost += h.cost;
  fireplace = fireplace  h.fireplace;

  return *this;

}
```

Reimplementing addition operator (+)

- Simpler version based on +=

Reimplementing addition operator (+)

- Simpler version based on +=

Reimplementing addition operator (+)

- Simpler version based on +=

```
House House::operator+ (const House &h) {

  House newHouse;
  newHouse  = *this;

  newHouse += h;

  return newHouse;

}
```

Overloading non-member operations

- What if you did not write the House class?

Overloading non-member operations

- What if you did not write the House class?
  - can’t implement addition (+) as a member function of House!

Overloading non-member operations

- What if you did not write the House class?
  - can’t implement addition (+) as a member function of House!
  - no problem, implement it as a non-member function with an additional House argument (standing for the reference object)

Overloading non-member operations

- What if you did not write the House class?
  - can’t implement addition (+) as a member function of House!
  - no problem, implement it as a non-member function with an additional House argument (standing for the reference object)

Overloading non-member operations

- What if you did not write the House class?
  - can't implement addition (+) as a member function of House!
  - no problem, implement it as a non-member function with an additional House argument (standing for the reference object)

```
House operator+ (const House &h1, const House &h2) {
  House newHouse;
  newHouse  = h1;
  newHouse += h2;
  return newHouse;
}
```

- Similarly for the case when the first operand is a primitive type

Stream Operators

- Similar for stream operator << in C++

Stream Operators

- Similar for stream operator << in C++
- Want to add a stream insertion operator (operator«) to the House class

Stream Operators

- Similar for stream operator << in C++
- Want to add a stream insertion operator (operator«) to the House class

Stream Operators

- Similar for stream operator << in C++
- Want to add a stream insertion operator (operator«) to the House class

```
myStream << h1;

void operator<< (ostream &out, const House &h) {
  out << "HOUSE" << endl;
  out << "Location: "  << address   << endl;
  out << "Owner: "     << owner     << endl;
  out << "Cost: "      << cost      << endl;
  out << "Fireplace: " << fireplace << endl;
  out << endl;
}
```

- One issue: fields (e.g. address) are private!

Overloading non-member operations

Overloading non-member operations

```
class House {
  void print(ostream &out) const;
  ...
};

void House::print(ostream &out) const{
  out << "HOUSE"<< endl;
  out << "Location: "<< address<< endl;
  out << "Owner: "<< owner<< endl;
  out << "Cost: "<< cost<< endl;
  out << "Fireplace: "<< fireplace<< endl;
  out << endl;
}

void House::print() const{
  print(cout);}

void operator<< (ostream &out, const House &h) {
  h.print(out);}
```

Overloading non-member operations

- But `cout << h2 << endl;` will give compile time error!

Overloading non-member operations

- But `cout << h2 << endl;` will give compile time error!
- Use the following implementation instead:

Overloading non-member operations

- But `cout << h2 << endl;` will give compile time error!
- Use the following implementation instead:

Overloading non-member operations

- But `cout << h2 << endl;` will give compile time error!
- Use the following implementation instead:

```
ostream &operator<< (ostream &out, const House &h) {
  h.print(out);
  return out;
}
```

- e.g. The operator returns the stream for the next thing