CS 3710 Vocational Languages
October 5, 2012

Define New Types

- In C++, you can define new types as either structs or classes.
- `author` in `struct_book` is public
- `author` in `class_book` is private
- All member functions defined inside the class/struct definition are inline.

Creating Objects

- Automatic memory allocation, where the system allocates and de-allocates the memory associated with the object:
  ```cpp
  Book book1();
  int x=book1.year_v();
  ...
  delete book1;
  ```
- Dynamic memory allocation, where programmers allocate and de-allocate the memory associated with the object:
  ```cpp
  Book* book2 = new Book();
  int x=book2->year_v();
  ...
  delete book2;
  ```

Constructors

- Constructors are responsible for the initialization of members of the created object.
  - A default constructor is a constructor that has no parameters.
  - A copy constructor is a constructor that is invoked to make a copy of an existing object.
  - If any type of constructor is defined in a class, C++ compiler won’t generate a default constructor.
- If NO constructor is defined in a class, compiler generates:
  - Default constructor:
    ```cpp
    Book::Book(){//empty body}
    ```
  - Copy constructor (member-wise shadow copy):
    ```cpp
    Book::Book(const Book other):title(other.title), year(other.year){//empty body}
    ```
Compiler Generated Constructors are not sufficient

- A constructor creates an object in the following steps:
  1. Allocate memory to hold the object;
  2. Initialize the members using specified initial values if provided.
     Otherwise it uses the member’s default constructor.
  3. Executes the constructor body

- Compiler generated default constructor:
  ```
  Book::Book()//empty body
  ```

- Title: empty string, which is initialized by the STL string class
- Year: garbage

Programmer Defined Constructors

```c
#include "Book.h"

//default constructor
Book::Book():year(0){}

//copy constructor, same as
//the compiler generated one
Book::Book(const Book& aBook)
{
  title=aBook.title;
  year=aBook.year;
}

//constructor with parameters
Book::Book(string title_in, int year_in)
{
  title=title_in;
  year=year_in;
}
```

### Assignment Operator=

- Assign an object of a class type to another object of the same class.
- Assignment operator and copy constructor are similar, in that they both perform memberwise copy from one object to another.
- Difference: assignment operator **overwrites** object on the left-hand-side of =, while copy constructor initializes the object value the first time.
- Assigned object has to exist previously.

```c
Book& Book::operator=(const Book& rhs)
{
  //check for self-assignment
  if(this!=&rhs)
  {
    title=rhs.title;
    year=rhs.year;
  }
  return *this;
}
```

- Passing object parameter:
  - involves copy constructor (create a new copy of the parameter object)
- Returning object from a function:
  - involves copy constructor (create a new copy of the return object)
- Assigning the returned object to a variable:
  - Involves assignment operator (overwrite the existing object)

```c
int main(){
  //use default constructor
  Book a;
  //use the constructor with parameters
  Book b("C", 1990);
  //use copy constructor
  Book c=a;
  //use assignment operator=
  Book d=b;
  //use ?
  c=a.swap(b);
}
```
### The rule of three

- When a class allocates memory dynamically (`new`) in its **constructor**, it requires every copy of the object to deal with the memory allocation/de-allocation correctly.
- If a class defines one of the following, it should probably explicitly define all three:
  - Copy constructor (memory allocation)
  - Assignment operator (memory allocation/de-allocation)
  - Destructor (memory de-allocation)

### Overloading Operators

- Operators are functions, which can be defined inside a class (member function) or outside a class.
- Just as function can be overloaded using different signatures, operators can be overloaded using different operands.

```cpp
bool Book::operator==(Book other)
{
//return 1 if book and other are the same.
return (year_v==other.year() &&
name_v.compare(other.name()) == 0);
}
bool Book::operator==(string otherTitle)
bool Book::operator==(int otherYear)
```

### Operator member function

- **Index operator** (`operator[]`) and assignment operator (`operator=`) have to be **member function** if defined.
- **When a binary operator** is defined as a member function, its left-hand operand is bound to the object on which it is invoked.

```cpp
bool Book::operator==(Book other);
if(book1==book2) ...
//first operand is book1, other is book2
```

### Operator functions

- To make sure the argument is passed more efficiently, pass reference using `&`.
- To make sure the passed argument is not modified in the function, add `const`.

```cpp
bool Book::operator==(const Book& other);
```
Overloading istream operators

- This is defined outside of a function.
  ```cpp
cout << foo;
```
- The operand to the right of `<<` is inserted into the stream.
- The return type of `<<` is the stream itself, which has been modified.
- The returned stream is not a new stream, but the stream on the LHS of the operator.

Overloading istream operator

- To declare `<<`, add to Book.h
  ```cpp
class Book{
public:
  friend ostream& operator<<(ostream& os, const Book& b);
};
```
- The first argument of `<<` is the ostream instance.
- `<<` and `>>` can never be member function.
- Iostream.h defines `<<` and `>>` as member functions for all built-in types.

Example

```cpp
#include "Book.h"
int main()
{
  Book a("C", 1990);
  cout << "book a is" << a << endl;
  return 0;
}
```