UNIVERSITY of LOUISVILLE
dare to be great

CPP Chapters 5-6-7:
Memory, Namespaces, Friends and Operator Overloading

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Lecture Outline

- Chapter 5
  - Inline function
  - This pointer
- Chapter 6
  - Dynamic Memory Allocation with New and Delete
- Chapter 7
  - Namespaces

- Random C++
  - Operator Overloading
  - Friend functions

Chapter 5: Inline functions

- An inline function is one in which the function code replaces the function call directly.
- Inline class member functions
  - if they are defined as part of the class definition, implicit
  - if they are defined outside of the class definition, explicit, i.e.using the keyword, *inline*.
- Inline functions should be short (preferable one-liners).
  - Why? Because the use of inline function results in duplication of the code of the function for each invocation of the inline function

Example of Inline functions

```
class CStr
{
  char *pData;
  int nLength;
    …
  public:
     …
    char *get_Data(void) {return pData; }//implicit inline function
    int getlength(void);
     …
};

inline void CStr::getlength(void) //explicit inline function
{
  return nLength;
}
    …

int main(void)
{
  char *s;
  int n;
  CStr a("Joe");
  s = a.get_Data();
  n = b.getlength();
}
```

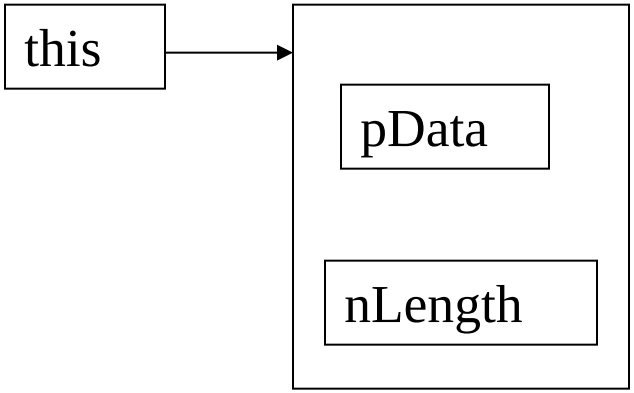
Inline functions within class declarations

Inline functions outside of class declarations

In both cases, the compiler will insert the code of the functions get_Data() and getlength() instead of generating calls to these functions

The "*this*" pointer

- Within a member function, the *this* keyword is a pointer to the current object, i.e. the object through which the function was called
- C++ passes a hidden *this* pointer whenever a member function is called
- Within a member function definition, there is an implicit use of *this* pointer for references to data members



CStr object
(*this)

Data member reference	Equivalent to
pData	this->pData
nLength	this->nLength

The "*this*" pointer example

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

class X {
private:
  int a;
public:
  void Set_a(int a) {

    // The 'this' pointer is used to retrieve 'xobj.a'
    // hidden by the automatic variable 'a'
    this->a = a;
  }
   void Print_a() { cout << "a = " << a << endl; }
};

int main() {
  X xobj;
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
  system("pause");
}
```

Chapter 6: C++ Dynamic Memory Allocation

- In C:

```
int* a = (int *) malloc(sizeof(int));
free(a);
```

- In C++:

```
int* a = new int;
delete a;
```

Chapter 7: Namespaces

•Namespaces allow us to group entities like classes, objects and functions under a name.

•The global scope can be divided into subscopes

•The format of namespaces is:

namespace identifier {
entities
}

•Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace.

Using Namespaces

```
namespace myNamespace {
        int a, b;
}
```

•In this case, the variables a and b are normal variables declared within a namespace called myNamespace.

•In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::.

•For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a;
myNamespace::b;
```

Namespaces: Example

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

namespace first {
  int var = 5;
}

namespace second {
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

Output:

5
3.1416

The keyword ‘using’ Declaration

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

namespace first {
  int x = 5;
  int y = 10;
}

namespace second {
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

x (without any name qualifier) refers to first::x
y refers to second::y,
exactly as our **using** declarations have specified.
We still have access to first::y and second::x
using their fully qualified names.

Output:
5
2.7183
10
3.1416

The keyword ‘using’ Directive

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

namespace first {
  int x = 5;
  int y = 10;
}

namespace second {
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}
```

Output:

5

10

3.1416

2.7183

The keyword ‘using’ with Scope

•using and using namespace have validity only in the same block in which they are stated
•If we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

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

namespace first {
  int x = 5;
}

namespace second {
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```

Output:
5
3.1416

Namespace alias

•We can declare alternate names for existing namespaces according to the following format:

namespace new_name = current_name;

Unnamed Namespace

- C++ allows you to access the members of an unnamed namespace.
- You can declare an unnamed namespace as a superior alternative to the use of global static variable declarations.
- Syntax:

```
namespace {
    members
}
```

Unnamed Namespace: Example

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

namespace {
        void func(void) {
                cout<<“::func”<<endl;
        }
}

int main (void) {
        ::func();
}
```

Output:
::func

Duplicate Namespaces

```
namespace x {
    func1() {}
}

namespace x {
    func2() {}
}
```

The same as:

```
namespace x {
    func1() {}
    func2() {}
}
```

Namespace std

- All the files in the C++ standard library declare all of its entities within the std namespace.
- That is why we have generally included the **using namespace std;** statement in all programs that used any entity defined in iostream.

Operator overloading

- Programmer can use some operator symbols to define special member functions of a class
- Provides convenient notations for object behaviors

Why Operator Overloading

```
int i, j, k;          // integers
float m, n, p;        // floats

k = i + j;
    // integer addition and assignment
p = m + n;
    // floating addition and assignment
```

The compiler overloads the + operator for built-in integer and float types by default, producing integer addition with i+j, and floating addition with m+n.

We can make object operation look like individual int variable operation, using operator functions

```
Complex a,b,c;
c = a + b;
```

Operator Overloading Syntax

- Syntax is:

operator@*(argument-list)*

--- operator is a function

@ is one of C++ operator symbols (+, -, =, etc..)

Examples:

operator+

operator-

operator*

operator/

Example of Operator Overloading

```
class CStr
{
    char *pData;
    int nLength;
  public:
    // …
    void cat(char *s);
    // …
     CStr operator+(CStr str1, CStr str2);
     CStr operator+(CStr str, char *s);
     CStr operator+(char *s, CStr str);

     //accessors
    char* get_Data();
    int get_Len();

};
```

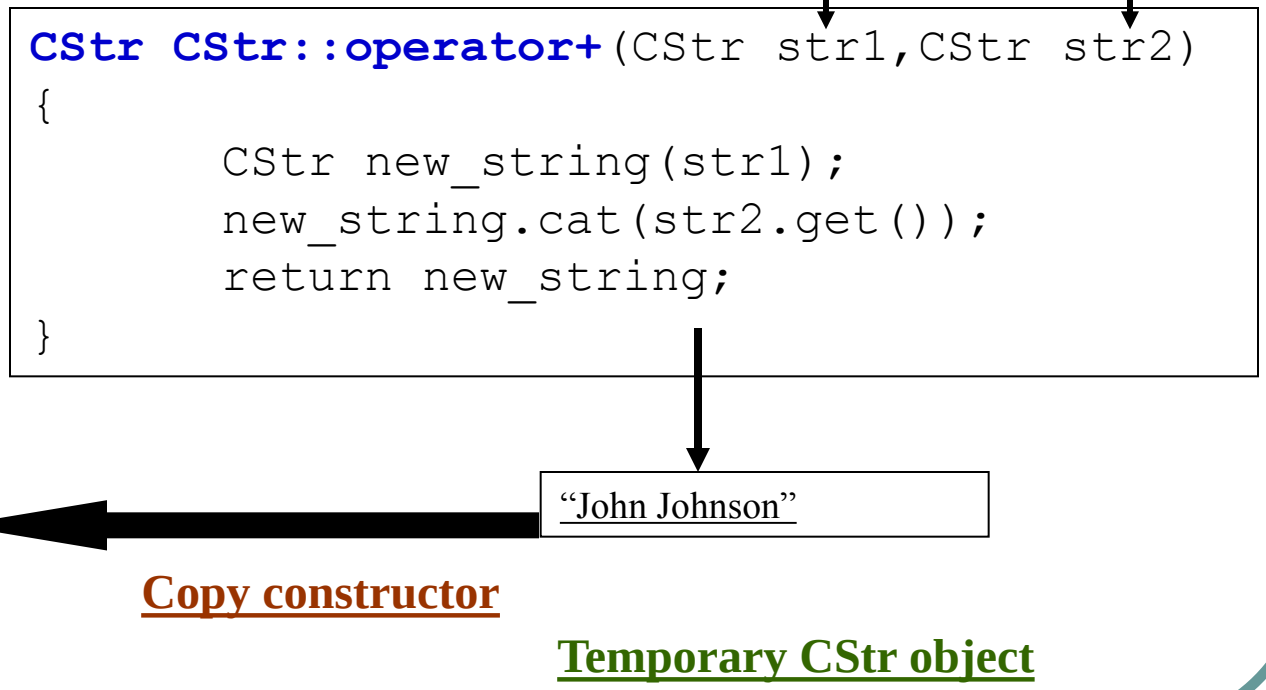
```
// function to append a string
void CStr::cat(char *s)
{
   int n;
   char *pTemp;
   n=strlen(s);
   if (n==0) return;

   pTemp=new char[n+nLength+1];
   if (pData)
      strcpy(pTemp,pData);

   strcat(pTemp,s);
   pData=pTemp;
   nLength+=n;
}
```

The Addition (+) Operator

```
CStr CStr::operator+(CStr str1, CStr str2)
{
   CStr new_string(str1);
            //call the copy constructor to initialize an
            //entirely new CStr object with the first
            //operand
   new_string.cat(str2.get_Data());
            //concatenate the second operand onto the
            //end of new_string
   return new_string;
            //call copy constructor to create a copy of
            //the return value new_string
}
```

How does it work?

```
CStr first("John");
CStr last("Johnson");
CStr name(first+last);
```



Implementing Operator Overloading

- Two ways:
  - Implemented as member functions
  - Implemented as non-member or Friend functions
    - the operator function may need to be declared as a friend if it requires access to protected or private data
- Expression *obj1@obj2* translates into a function call
  - *obj1.operator@(obj2)*, if this function is defined within class obj1
  - *operator@(obj1,obj2)*, if this function is defined outside the class obj1

Implementing Operator Overloading

1. Defined as a member function

```
class Complex {
   ...
 public:
   ...
   Complex operator +(const Complex &op)
   {
     double real   = _real   + op._real,
                 imag = _imag + op._imag;
     return(Complex(real, imag));
   }
   ...
 };
```

c = a+b;

c = a.operator+ (b);

Implementing Operator Overloading

2. Defined as a non-member function

```
class Complex {
   ...
 public:
   ...
  double real() { return _real; }
    //need access functions
  double imag() { return _imag; }
   ...
 };
```

c = a+b;

c = operator+ (a, b);

```
Complex operator +(Complex &op1, Complex &op2)
{
   double real   = op1.real()   + op2.real(),
                  imag = op1.imag() + op2.imag();
   return(Complex(real, imag));
}
```

What is 'Friend'?

- Friend declarations introduce extra coupling between classes
  - Once an object is declared as a friend, it has access to all non-public members as if they were public
- Access is unidirectional
  - If B is designated as friend of A, B can access A's non-public members; A cannot access B's
- A friend function of a class is defined outside of that class's scope

Implementing Operator Overloading

3. Defined as a friend function

```
class Complex {
   ...
 public:
   ...
  friend Complex operator +(
     const Complex &,
     const Complex &
   );
   ...
};
```

c = a+b;

c = operator+ (a, b);

```
Complex operator +(Complex &op1, Complex &op2)
{
   double real   = op1._real   + op2._real,
                 imag = op1._imag + op2._imag;
   return(Complex(real, imag));
}
```

More about 'Friend'

- The major use of friends is
  - to provide more efficient access to data members than the function call
  - to accommodate operator functions with easy access to private data members
- Friends can have access to everything, which defeats data hiding, so use them carefully
- Friends have permission to change the internal state from outside the class.
- Always use member functions instead of friends to change state

Assignment Operator

- Assignment between objects of the same type is always supported
  - the compiler supplies a hidden assignment function if you don't write your own
  - same problem as with the copy constructor - the member by member copying
  - Syntax:

```
class& class::operator=(const class &arg)
{
   //…
}
```

Example: Assignment for CStr class

Assignment operator for CStr:

CStr& CStr::operator=(const CStr & source)

Return type - a reference to (address of) a CStr object

Argument type - a reference to a CStr object (since it is const, the function cannot modify it)

```
CStr& CStr::operator=(const CStr &source){
//... Do the copying
return *this;
}
```

Assignment function is called as a member function of the left operand =>Return the object itself

str1=str2;

str1.operator=(str2)

Overloading stream-insertion and stream-extraction operators

- cout<< or cin>> are operator overloading built in C++ standard lib of iostream.h, using operator "<<" and ">>"
- cout and cin are the objects of ostream and istream classes, respectively
- We can add a friend function which overloads the operator <<

friend ostream& operator<< (ostream &os, const Date &d);

```
ostream& operator<<(ostream &os, const Date &d)
{
   os<<d.month<<"/"<<d.day<<"/"<<d.year;
   return os;
}
…
cout<< d1;  //overloaded operator
```

Overloading stream-insertion and stream-extraction operators

- We can also add a friend function which overloads the operator >>

friend istream& operator>> (istream &in, Date &d);

```
istream& operator>> (istream &in, Date &d)
{
  char mmddyy[9];
  in >> mmddyy;
  // check if valid data entered
  if (d.set(mmddyy))   return in;
  cout<< "Invalid date format: "<<d<<endl;
  exit(-1);
 }
```

cin ---- object of istream

cin >> d1; //Overloaded Operator

The End!

Based on: http://www.cplusplus.com/doc/tutorial/namespaces.html
and slides by Dr. Neelam Soundarajan