OPERATOR OVERLOADING

(2017)

OPERATOR OVERLOADING

- Operator overloading is just “syntactic sugar”, which means it is simply another way for you to make a function call.

- The difference is that the arguments for this function don’t appear inside parentheses, but instead they surround or are next to characters you’ve always thought of as immutable operators.

OPERATOR OVERLOADING (CONST…)

- In C++ it is possible to define new operators that work with classes.

- This definition is just like an ordinary function definition except that the name of the function consists of the keyword operator followed by the operator.

(I) Number of arguments

The number of arguments in the overloaded operator's argument list depends on two factors:

1. Whether it is an unary operator (one argument) or a binary operator (two arguments).

2. Whether the operator is defined as a global function (one argument for unary, two for binary) or a member function (zero arguments for unary, one for binary – the object becomes the left-hand argument).

(II) What you cannot do in operator overloading

- **Although you can overload almost all the operators available in C++, you <u>cannot</u>:**
  - combine operators that currently have no meaning in C++ (such as ** to represent exponentiation) or make up new operators that aren't currently in the set.
  - change the evaluation precedence of operators.
  - change the number of arguments required by an operator.
  - overload | :: | . | .* | ?: |

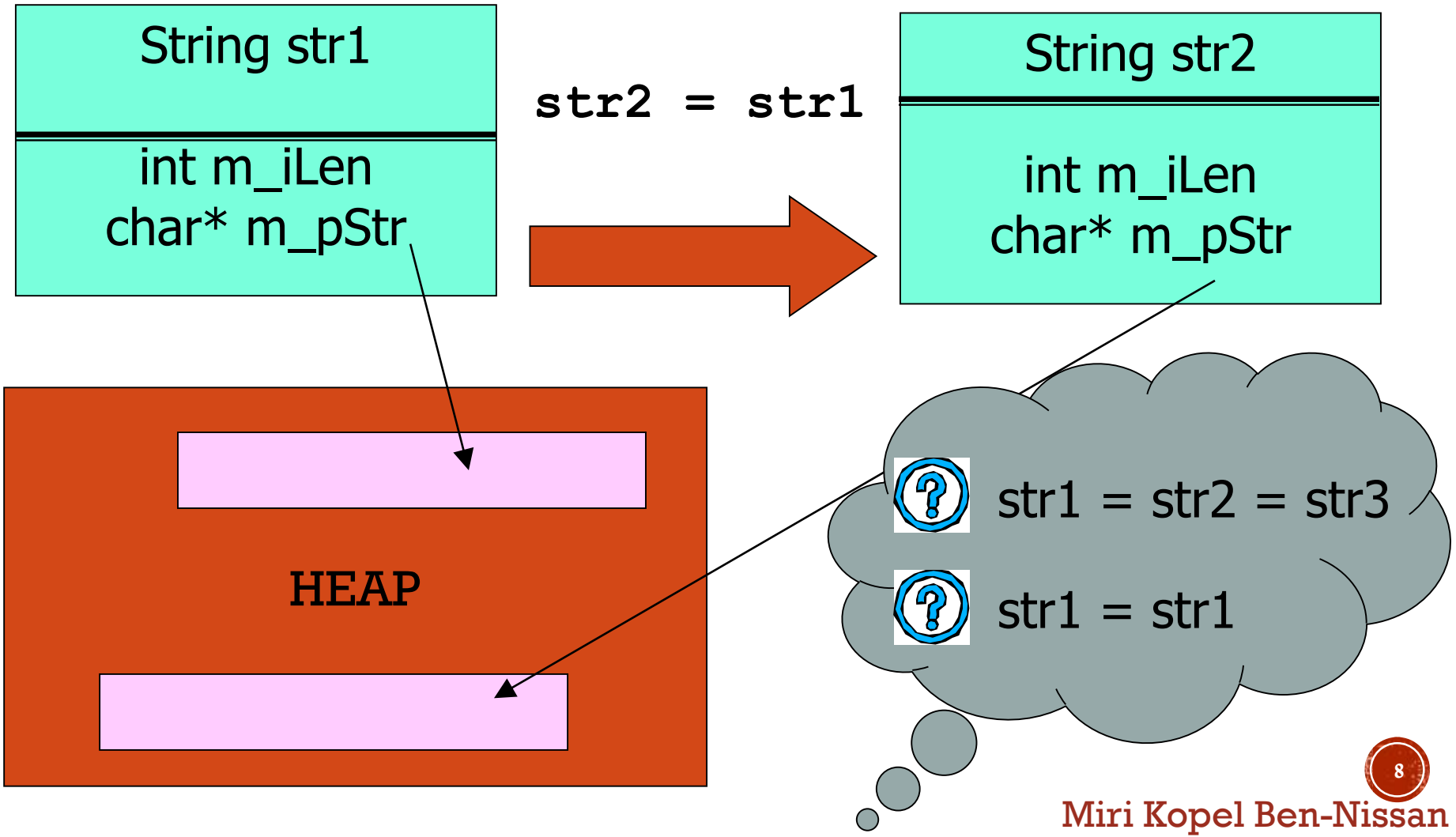
(III) Arguments and Return Values

1. If you only need to read from the argument and not change it, default to passing it as a const reference.
2. If the effect of the operator is to produce a new value, you will need to generate a new object as the return value. This object is returned by value as a const, so the result cannot be modified as an l-value.
3. The return value for all of the assignment operators should be a non-const reference.
4. For logical operators, return a bool type.

(IV) RETURN BY VALUE AS CONST

- **If you use the binary operator+ in an expression such as f(a+b), the result of a+b becomes a temporary object that is used in the call of f( ).**
  - Because it is temporary, it's automatically const.
  - to prevent the user from storing potentially valuable information in an object that will most likely to be lost, like in (a+b).g( ), you should make the return value const.

(V) OVERLOADING ASSIGNMENT



(VI) OPERATOR= VS. COPY CONSTRUCTOR

- **The copy constructor initializes uninitialized memory.**
  The assignment operator must correctly deal with a well-constructed object.
  - The general strategy for an assignment operator: protect against self-assignment, delete old elements, initialize, and copy-in new elements.

(VII) AUTOMATIC TYPE CONVERSION

- In C++, if the compiler sees an expression or function-call using a type that isn't quite the one it needs, it can often perform an automatic type conversion from the type it has to the type it wants.

- You can achieve this same effect for user-defined types by defining automatic type conversion functions.

AUTOMATIC TYPE CONVERSION (CONT...)

1. **Define a constructor that takes an object (or reference) of another type as its single argument. That constructor allows the compiler to perform an automatic type conversion.**
   - **drawback**: the cost is the hidden constructor call, which may matter if you're concerned about the efficiency of calls to this function.
   - **solution**: make the constructor explicit.

AUTOMATIC TYPE CONVERSION (CONT...)

2. **create a member function that takes the current type and converts it to the desired type, using the operator keyword followed by the type you want to convert to.**
   - the return type is the same name of the operator you're overloading.

```
class One
{
   int m_i;
public:
   One(int i=0) : m_i(i){}
};
```

```
class Two
{
   int m_x;
public:
   Two(int x) : m_x(x) {}
   operator One() const {return One(m_x);}
};
```

```
void func(const One&
                one)
{
}
```

```
void main()
{
   Two t(1);//int→Two
   func(t); //Two→One
   func(1); //int→One
}
```

AUTOMATIC TYPE CONVERSION (CONT…)

- With the constructor technique, the destination class is performing the conversion, but with operators, the source class performs the conversion.

- There's no way to use the constructor in order to convert a user-defined type into a built-in type; this is possible only with operator overloading.

EXPLICIT CONSTRUCTOR

- **By default, a single argument constructor defines an implicit conversion.**
  - For example: `Complex z = 2;` initializes `z` with `Complex(2);`
- **There are cases in which conversion is undesirable and error-prone.**
  - For example: if we have `String(int size)`, than one would write: `String s = 'a'` , and create `s` with `'a'` places.
- **Implicit conversion can be suppressed by declaring a constructor *explicit*.**

AUTOMATIC TYPE CONVERSION (CONT...)

```
class String
{
public:
   //pre-allocates n bytes for the string.
   explicit String(int n);

   //initial value is pStr.
   String(const char* pStr);

   //...
};

void f(String); //global function
```

```
int main()
{

   String s1 = 'a';
    //error: no implicit char->String conversion.
   String s2(10);
    //OK: String with space for 10 chars.
   String s3 = "testing";
    //OK: conversion of char*->String.
   f(10);
    //error: no implicit int->String conversion.
   f(String(10));
    //OK.
}
```

FRIENDS

- **The *friend* keyword allows a function or class to gain access to the private and protected members of a class.**

- **A friend function is a function that is not a member of a class but has access to the class's private and protected members.**
  - A friend function is declared by the class that is granting access. The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.
  - the friend keyword does not appear in the function definition.

FRIENDS (CONT...)

- **A friend class is a class all of whose member functions are friend functions of a class, that is, whose member functions have access to the other class's private and protected members.**
  - Friendship is not mutual unless explicitly specified as such.

EXAMPLE: STRING CLASS

```
// String.h: interface for the String class.
//////////////////////////////////////////////////////////////////
#ifndef __STRING_H__
#define __STRING_H__
#include <iostream.h>

class String
{
    enum{MAXSIZE=1024};

public:
    //-- constructors/destructor --//
    String(const char* str=NULL);
    String(const String&);
    ~String();
```

```
//-- operators --//
String& operator=(const String&);
String& operator+=(const String&);

friend String operator+(const String&,const String&);

//logical operators
friend bool operator==(const String& , const String&);
friend bool operator!=(const String& , const String&);
friend bool operator<(const String& , const String&);
friend bool operator>(const String& , const String&);
friend bool operator<=(const String& , const String&);
friend bool operator>=(const String& , const String&);

//I/O operator
friend ostream& operator<<(ostream& , const String&);
friend istream& operator>>(istream& , String&);

//indexing operator
char& operator[](int) const;
```

```
   //casting operator
   operator char*() const;

   //-- methods --//
   int  Length() const;
   void Insert(int index_pos,const String&);
   int  Remove(int index_pos,int length);
   //finds a substring in the current string.
   int  Find(const String&) const;
   //returns count characters String starting at index_pos.
   String Substr(int index_pos,int iCount) const;
   //replaces substring s with substring t.
   int  Replace(const String& s, const String& t);

private:
   //-- data members --//
   char* m_str; //pointer to dynamically allocated data.
   int   m_iLen;//current length of the string.
   int   m_iSize;//maximum size of the string.
};
```

```
//////////////////// inline functions ////////////////////

inline int String::Length() const
{
  return m_iLen;
}

inline String::operator char*() const
{
  return m_str;
}

#endif //__STRING_H__
```

```
// String.cpp: implementation of the String class.
//////////////////////////////////////////////////////////////////////
#include "String.h"
#include <string.h>
#include <assert.h>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// constructor/default constructor
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
String::String(const char* str/*=NULL*/)
{
    if(str)  cout<<"::converting char* to String::\n";
    else     cout<<"::default constructor::\n";
    if(str){
       m_iLen = strlen(str);
       m_str = new char[m_iLen + 1];
       strcpy(m_str,str);
    }
    else{
       m_str = NULL;
       m_iLen = 0;
    }
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// copy constructor
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
String::String(const String& str)
{
    cout<<"::copy constructor::\n";
    m_iLen = str.m_iLen;
    m_str = new char[m_iLen + 1];
    strcpy(m_str,str.m_str);
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// destructor
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
String::~String()
{
    cout<<"::destructor::\n";
    if(m_str)
    {
        delete[] m_str;
        m_str = NULL;
    }
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// operator =
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
String& String::operator = (const String& str)
{
    //checking self-assignments
    if(this != &str)
    {
        //check if there's memory to release
        if(m_str){
            delete[] m_str;
            m_str = NULL;
        }
        //copy str to currect string
        m_iLen = str.m_iLen;
        m_str = new char[m_iLen+1];
        strcpy(m_str , str.m_str);
    }

    return *this;
}
```

```
// operator +=
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
String& String::operator += (const String& str)
{
    if(m_str){
        int iNewLen = m_iLen + str.m_iLen;
        char* sNewStr = new char[iNewLen+1];
        strcpy(sNewStr,m_str);
        //concatenate the two strings
        strcat(sNewStr,str.m_str);
        delete[] m_str;
        m_str = NULL;
        //update the member attributes:
        m_str = sNewStr;
        m_iLen = iNewLen;
    }
    else{ //m_str is empty
        m_iLen = str.m_iLen;
        m_str = new char[m_iLen+1];
        strcpy(m_str,str.m_str);
    }
    return *this;
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// operator []
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
char& String::operator [] (int iIndex) const
{
   assert(iIndex>=0 && iIndex<m_iLen);
   return m_str[iIndex];
}

////////////////////////////////////////////////////////////
// Friend Functions
////////////////////////////////////////////////////////////
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// operator +
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
String operator + (const String& s1, const String& s2)
{
   String sResult = s1;
   sResult+=s2;
   return sResult;
}
```

```
////////////////////// logical operators //////////////////////

bool operator == (const String& s1, const String& s2)
{
    return strcmp(s1.m_str , s2.m_str) == 0;
}
bool operator != (const String& s1, const String& s2)
{
    return strcmp(s1.m_str , s2.m_str) != 0;
}
bool operator < (const String& s1, const String& s2)
{
    return strcmp(s1.m_str , s2.m_str) < 0;
}
bool operator > (const String& s1, const String& s2)
{
    return strcmp(s1.m_str , s2.m_str) > 0;
}
bool operator <= (const String& s1, const String& s2)
{
    return strcmp(s1.m_str , s2.m_str) <= 0;
}
bool operator >= (const String& s1, const String& s2)
{
    return strcmp(s1.m_str , s2.m_str) >= 0;
}
```

```
//////////////////////// I/O operators ////////////////////////
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// operator <<
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ostream& operator << (ostream& out, const String& str)
{
   //verify that string isn't empty to avoid access violation
   if(str.m_str){
    out<<str.m_str;
   }
   return out;
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// operator >>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
istream& operator >> (istream& in, String& str)
{
   char sBuffer[String::MAXSIZE];
   in.getline(sBuffer,String::MAXSIZE);
   str = sBuffer;
   return in;
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Insert
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void String::Insert(int index_pos,const String& str)
{
  assert((index_pos >= 0) && (index_pos <= m_iLen));

  int iNewLen = m_iLen + str.m_iLen;
  char *pStr = new char[iNewLen + 1];

  strncpy(pStr, m_str, index_pos);

  strcpy(pStr + index_pos, str.m_str);
  strcpy(pStr + index_pos + str.m_iLen, m_str +
index_pos);

  delete[] m_str;

  m_str = pStr;
  m_iLen = iNewLen;
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Remove
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
int String::Remove(int index_pos,int length)
{
  assert((index_pos >= 0) && (index_pos <= m_iLen));
  assert(length <= m_iLen - index_pos);
  int iNewLen = m_iLen - length;
  if (iNewLen == 0){
     delete[] m_str;
     m_str = NULL;
  }
  else{
     char *pStr = new char[iNewLen + 1];
     strncpy(pStr, m_str, index_pos);
     strcpy(pStr + index_pos, m_str + index_pos + length);
     delete[] m_str;
     m_str = pStr;
  }
  m_iLen = iNewLen;
  return 0;
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Find
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
int String::Find(const String& str) const
{
   int i, j;
   for (i=0; m_str[i]; ++i)
   {
      for (j=0; m_str[i+j] && str.m_str[j] &&
              (m_str[i+j] == str.m_str[j]); ++j);

      if (!str.m_str[j])
        return i;
   }
   return -1;
}
```

```cpp
// main.cpp: the use of the String class.
////////////////////////////////////////////////////////

#include "String.h"

void byValue(String str)
{
    cout<<"::by value::\n";
}

int main()
{
    String s1("object");
    String s2;

    //-- copy constructor --//
    byValue(s1);

    //-- assignment --//
    s2 = s1;
```

```
//-- comparison --//
if(s1==s2){
   cout<<":: s1 == s2 ::\n";
}

s2 = "oriented";

//-- comparison --//
if(s1!=s2){
   cout<<":: s1 != s2 ::\n";
}

//-- operator "less then" --//
if(s1 < s2){
   cout<<":: s1 < s2 ::\n";
}
else{
   cout<<":: s1 !< s2 ::\n";
}
```

```
//-- operator "bigger than" --//
if(s1 > s2){
 cout<<":: s1 > s2 ::\n";
}
else{
 cout<<":: s1 !> s2 ::\n";
}

//-- copy constructor and operator plus --//
String s3 = s1 + " " + s2;

//-- output --//
cout<<"s3="<<s3<<endl;

//-- operator [] --//
cout<<"::first letter of s3 is \""<<s3[0]<<"\"::\n";
cout<<"The letters of s1 are: ";
for(int i=0; i<s1.Length(); ++i)
 cout<<s1[i]<<" ";
cout<<endl;
```

```
//-- assignment - first delete the old value of s2 --//
s2 = s3;

//-- operator casting --//
cout<<"s2="<<(char*)s2<<endl;

//-- insert method --//
s2.Insert(6," -");
cout<<"s2="<<s2<<endl;

//-- find method --//
int iPos = s2.Find("jec");
cout<<"\"jec\" is in s2 at possition "<<iPos<<endl;

cout<<":: Last line in the main ::\n";
}
```

THE OUTPUT:

=-=-=-=-=-=-=-=-=

::converting char* to String::
::default constructor::
::copy constructor::
::by value::
::destructor::
:: s1 == s2 ::
::converting char* to String::
::destructor::
:: s1 != s2 ::
:: s1 < s2 ::
:: s1 !> s2 ::

```
::converting char* to String::
::copy constructor::
::copy constructor::
::destructor::
::copy constructor::
::copy constructor::
::destructor::
::destructor::
::destructor::
object oriented
::first letter of s3 is "o"::
o b j e c t
object oriented
:: Last line in the main ::
::destructor::
::destructor::
::destructor::
```