Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

September 7, 2018

C++ Classes: Part III

- We have seen how to create (or instantiate) objects of a class.
- In this lecture we shall learn how to perform the other three operations:
  1. **Copy.**
  2. **Assign.**
  3. **Destroy.**
- The above three actions typically go together, in the sense that if it is necessary to override the default action of the compiler for any one of them, it is usually necessary to override all three.
- Some authors therefore refer to the above operations as the **Big Three.**

1 Class PStrings: for use in later examples

```
class PStrings {
public:

  PStrings(int n, string s[])
  {
    length = n;
    pstr = new string[length];              // dynamic memory allocation, operator new
    for (int i = 0; i < length; ++i)
      pstr[i] = s[i];
  }

  // destroy
  // copy
  // assign

  void print() const {
    cout << "length = " << length << endl;
    cout << "pstr: " << endl;
    for (int i = 0; i < length; ++i)
      cout << pstr[i] << endl;
  }

private:
  int length;
  string *pstr;
};
```

2 Destructor, part 1

- The **destructor** of a class is invoked when an object goes out of scope.
- As with the default constructor, the compiler will automatically generate a destructor of we do not write one.
- Also as with the default constructor, the automatically generated destructor may not do exactly what we want.
- The automatically generated destructor typically does nothing.
- Consider the class `PStrings` in Sec. 1.
  1. This class has the feature that memory is dynamically allocated in the constructor, to hold a dynamically allocated array of strings, using `operator new`.
  2. Recall that memory that is dynamically allocated using `operator new` is only released when `operator delete` is called (in this case `operator delete []`).
  3. The automatically generated destructor by the compiler does not call `operator delete []`.
  4. *How could it know to do such a thing?*
- **Hence when an object of the class `PStrings` goes out of scope, there will be a memory leak.**
- The following program has a memory leak, when the object `ps` goes out of scope.

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

class PStrings {
  // etc
};

int main()
{
  int n = 3;
  string s[] = {"a", "b", "c"};
  PStrings ps(n, s);
  ps.print();
  return 0;
}                                   // object goes out of scope, memory leak
```

- We must write an explicit destructor and deallocate the memory correctly.

3 Destructor, part 2

- We write a destructor for the class `PStrings`.

```
class PStrings {
public:

  // destroy
  ~PStrings()
  {
    delete [] pstr;
  }

  // etc
};
```

- The destructor has the following syntax.
  1. **The name of the destructor is "∼" followed by the name of the class.**
  2. **There is no return type.**
  3. **The destructor has no input arguments.**
- **There is only one destructor for a C++ class.**
- Unlike constructors, there are no "non-default" destructors.
- When the destructor is invoked, the dynamically allocated memory is released correctly.
- If the above destructor is written for the class `PStrings`, the program displayed in Sec. 2 no longer leaks memory.

4 Class PStringsD: Class PStrings with destructor

- To avoid confusion, we write a class `PStringsD`.
- It is the same as `PStrings`, but we write an explicit destructor.

```
class PStringsD {
public:

  PStringsD(int n, string s[])
  {
    length = n;
    pstr = new string[length];
    for (int i = 0; i < length; ++i)
      pstr[i] = s[i];
  }

  // destroy
  ~PStringsD()
  {
    delete [] pstr;
  }

  void print() const {
    cout << "length = " << length << endl;
    cout << "pstr: " << endl;
    for (int i = 0; i < length; ++i)
      cout << pstr[i] << endl;
  }

private:
  int length;
  string *pstr;
};
```

5 Operators `new` and `delete` with destructor

- Operator `new` invokes a suitable default or non-default constructor to instantiate an object.
- **Operator `delete` invokes the destructor.**
- Here is the same main program displayed in Sec. 2:
  (i) using the class `PStringsD`,
  (ii) calling operators `new` and `delete`.

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

class PStringsD {
  // etc
};

int main()
{
  int n = 3;
  string s[] = {"a", "b", "c"};
  PStringsD psd(n, s);
  psd.print();

  PStringsD *ppsd = new PStringsD(n, s);  // operator new invokes constructor
  ppsd->print();
  delete ppsd;                            // operator delete invokes destructor
  return 0;
}                 // object "psd" goes out of scope, memory released correctly
```

6 Copy constructor, part 1

- We have written a destructor for the class `PStringsD`, but that is not enough.
- **The following program will compile, but it will crash when executed.**

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

class PStringsD {
  // etc
};

void print(PStringsD psd_copy)
{
  psd_copy.print();
}

int main()
{
  int n = 3;
  string s[] = {"a", "b", "c"};
  PStringsD psd(n, s);

  print(psd);
  psd.print();                    // object psd corrupted
  return 0;
}
```

- **See next page(s).**

7 Shallow copy and deep copy

- Because of call by value, the object `psd_copy` is a copy of `psd` in the main program.
- We have not specified *how to make a copy of an object of the class* `PStringsD`.
- The automatically generated copy by the compiler **simply copies all the data members.**

```
psd_copy.length = psd.length;
psd_copy.pstr   = psd.pstr;      // pstr in copy points to same memory address
                                 // as pstr in object in main program
```

- **Therefore the pointer `pstr` in the copy points to the same memory address as `pstr` in the original object in the main program.**
- The copy object goes out of scope at the end of the function call.
  1. The destructor is invoked for the copy object.
  2. **The destructor *deallocates the memory of the object in the main program.***

     ```
     delete [] pstr;          // *** releases memory in object in main program ***
     ```

  3. **The object `psd` in the main program is now corrupted.**
  4. **The memory in the object `psd` in the main program has been wrongly deallocated.**
  5. A run-time fault occurs when the statement "`psd.print()`" is executed in the main program.
- The default action by the compiler is known as a **shallow copy.**
- **A shallow copy copies the values of all the data members, *including pointers.***
- The correct action we really want is for the copy object to ***dynamically allocate its own memory*** and copy the **contents of the pointer.**
- This is different from simply copying the pointer.
- Such an action is called a **deep copy.**
- We shall see examples of deep copies below.

8 Copy constructor, part 2

- The function to make a copy is called the **copy constructor.**
- We write a copy constructor for the class `PStringsD`.
- The copy constructor implements a deep copy.

```
class PStringsD {
public:

  // copy
  PStringsD(const PStringsD &orig)
  {
    length = orig.length;
    pstr = new string[length];              // allocate fresh memory
    for (int i = 0; i < length; ++i)
      pstr[i] = orig.pstr[i];
  }

  // etc
};
```

- The copy constructor has the following syntax.
  1. **The name of the constructor is the name of the class.**
  2. **There is no return type.**
  3. **There is one input argument, a const reference to an object of the same class.**
- The first two items identify the function as a constructor.
- The third item identifies the function as a copy constructor.
- Else it would be a non-default constructor.
- We perform the deep copy as follows:
  1. We allocate fresh memory for the copy object: `pstr = new string[length];`
  2. After that we copy the **contents of the array.**
  3. See the loop: we copy `pstr[i]` for $i = 0, 1, \ldots$.
  4. When the destructor of the copy object is invoked, *the memory in the copy is deallocated.*
  5. **The memory in the original object is not affected.**
- If the above copy constructor is written for the class `PStringsD`, the program displayed in Sec. 6 will not crash.

9 Class PStringsDC: Class PStringsD with copy constructor

9.1 Class declaration

- To avoid confusion, we write a class `PStringsDC`.
- It is the same as `PStringsD`, but with a copy constructor.
  1. We also add a public method "`set`" to set the value of an element in the array.

     ```
     void set(int n, string s) {
       if ((n>= 0) && (n < length)) pstr[n] = s;
     }
     ```

  2. We test if the input value of n is valid. If true, we set the array element.
- The full list of function bodies has become too long to fit into one page, hence only the class declaration is given below.
- **All the class methods and constructors and destructor are written non-inline.**
- See in particular the non-inline syntax for the destructor.

```
class PStringsDC {
public:
  PStringsDC(int n, string s[]);             // construct
  ~PStringsDC();                             // destroy
  PStringsDC(const PStringsDC &orig);        // copy

  void set(int n, string s);
  void print() const;

private:
  int length;
  string *pstr;
};
```

9.2 Non-inline function bodies

- **All the class methods and constructors and destructor are written non-inline.**
- See in particular the non-inline syntax for the destructor.

```
// construct
PStringsDC::PStringsDC(int n, string s[])
{
  length = n;
  pstr = new string[length];
  for (int i = 0; i < length; ++i)
    pstr[i] = s[i];
}

// destroy
PStringsDC::~PStringsDC()
{
  delete [] pstr;
}

// copy
PStringsDC::PStringsDC(const PStringsDC &orig)
{
  length = orig.length;
  pstr = new string[length];
  for (int i = 0; i < length; ++i)
    pstr[i] = orig.pstr[i];
}

// public method
void PStringsDC::set(int n, string s)
{
  if ((n>= 0) && (n < length)) pstr[n] = s;
}

// public method
void PStringsDC::print() const
{
  cout << "length = " << length << endl;
  cout << "pstr: " << endl;
  for (int i = 0; i < length; ++i)
    cout << pstr[i] << endl;
}
```

10 Copy of object in main program

- **A function call is not the only occasion to invoke the copy constructor.**
  1. Suppose we have an already constructed object in the application.
  2. We wish to instantiate a second object which is mostly the same as the first object.
  3. It is simpler to make a copy, and then update the values of the items we wish to change.
- We invoke the copy constructor as follows.

```
// psdc1 already exists
PStringsDC psdc2(psdc1);          // copy
```

- **In this respect, a copy constructor is a non-default constructor where the input is a reference to an object of the same class.**
- The program below is the same as that displayed in Sec. 6 but using the class `PStringsDC`.
- The object `psdc2` is a copy of the object `psdc1`.

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

class PStringsDC {
  // etc
};

void print(PStringsDC psdc_copy)
{
  psdc_copy.print();
}

int main()
{
  int n = 3;
  string s[] = {"a", "b", "c"};
  string alpha("alpha");
  PStringsDC psdc1(n, s);
  print(psdc1);
  psdc1.print();

  PStringsDC psdc2(psdc1);          // copy
  psdc2.set(0, alpha);              // does not affect original object
  psdc2.print();
  psdc1.print();                    // original object not affected
  return 0;
}
```

11 Assignment operator, part 1

- We have written a destructor and copy constructor for the class `PStringsDC`.
- However, that is not enough.
- **The following program will compile, but it will crash when executed.**

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

class PStringsDC {
  // etc
};

int main()
{
  int n1 = 3;
  int n2 = 2;
  string s1[] = {"a", "b", "c"};
  string s2[] = {"alpha", "beta"};
  PStringsDC ps1(n1, s1);
  PStringsDC ps2(n2, s2);

  ps1 = ps2;                          // shallow copy of pointer
  ps1.print();
  ps2.print();
  return 0;
}                                     // bad memory deallocation
```

- **The problem is again the shallow copy of the pointer in the assignment.**
- The automatically generated assignment by the compiler performs a shallow copy.

```
ps1.length = ps2.length;
ps1.pstr   = ps2.pstr;              // shallow copy of pointer
```

- When both objects `ps1` and `ps2` go out of scope at the end of the main program, the destructors of both objects **call `operator delete []` on the same memory address.**
- It is a serious error to call `operator delete` on memory which has already been deallocated.
- Hence there is a run time fault at the end of the main program.
- We must perform a deep copy in the assignment statement `ps1 = ps2`.

12 Assignment operator, part 2

- The function to perform the assignment is called the **assignment operator.**
- It is the **operator "="** in the statement `ps1 = ps2`.
- The assignment operator has many similarities to the copy constructor.
- Both make copies of an input object.
- However, there are some notable differences.
  1. As its name suggests, the "assignment operator" is an operator, not a constructor.
  2. Its name is "**operator=**" and not the name of the class.
  3. *The assignment operator has a return type*, which a constructor does not.
  4. Although the final result is to make a copy of `ps2`, **note that the destination object `ps1` already exists.**
  5. For the copy constructor, the destination is a brand new object which does not exist yet.
  6. Because the destination object `ps1` already exists, ***we must deallocate its existing memory, before we allocate fresh memory for the pointer*** `pstr` ***in*** `ps1`**.**
  7. The above step is not required for the copy constructor because the destination object does not exist yet, hence there is no "previously allocated memory" to release.
  8. Because the assignment operator must deallocate the existing memory in the destination object, the assignment operator also shares some features in common with the destructor.
- However, there is more to writing the assignment operator than deallocating the existing memory and allocating fresh memory to perform a deep copy.
- We must learn more formalism about C++ to understand how to formulate the code for the assignment operator.
- We shall do so below.

13 Assignment operator, part 3

- The signature of the assignment operator for the class `PStringsDC` is as follows.

  1. Both the inline and non-line versions are displayed.

     ```
     PStringsDC& operator= (const PStringsDC &rhs);                  // inline
     PStringsDC& PStringsDC::operator= (const PStringsDC &rhs);      // non-inline
     ```

  2. There is a return type, which is a reference to an object of the same class.
  3. As already stated, the name is "`operator=`" to signify it is an operator.
  4. As with the copy constructor, the input argument is a reference to an object of the same class. This should be obvious.

- The input object is labelled "`rhs`" because an assignment statement "`=`" has an object on the left and the right. The input object is on the **right hand side**, hence `rhs`.

- Here is the code for the assignment operator for the class `PStringsDC`.

```
PStringsDC& operator= (const PStringsDC &rhs)
{
  if (this == &rhs) return *this;

  length = rhs.length;
  delete [] pstr;
  pstr = new string[length];
  for (int i = 0; i < length; ++i) {
    pstr[i] = rhs.pstr[i];
  }
  return *this;
}
```

- However, we need to understand several things to make sense of the above code.

- The most obvious question is: **what is the thing called "this" in the above code?**

- The second obvious question is: **what is the meaning of the test "`if (this == &rhs)`" in the top line of the function body?**

14 The "this" pointer

- C++ automatically generates a pointer named "**this**" for every object.
  1. The "**this**" pointer is a pointer to the object itself (pointer to "this object").
  2. The "**this**" pointer can only be accessed internally inside the class methods.
  3. The "**this**" pointer is not accessible by external calling applications.
  4. The "**this**" pointer is available internally in both public and private class methods.
- **The "this" pointer cannot be reassigned.** It always points to the current object.
- The statement "`return *this`" therefore returns a reference to the current object.
- *Why is such a statement required?*
- To answer that question, we must understand the return type of the assignment operator.

15 Chained assignments

- The reason for the return type is that C++ permits "chained assignments" for primitive data types such as `int` and `double`, etc.

```
i = j = k;
x = y = z;
```

- Hence for user-defined classes, the assignment operator must also support chained assignments of the form `ps1 = ps2 = ps3 = ps4`.

  1. Unlike most other operators, the "`operator=`" parses from **right to left.**
  2. This is because the object on the right is copied into the object on the left.
  3. Hence the compiler evaluates the above assignment in the following order: `ps1 = (ps2 = (ps3 = ps4))`.
  4. First the compiler computes the assignment `ps3 = ps4`.
  5. The result is stored in `ps3`, which is passed as an input to the assignment `ps2 = ps3`.
  6. The compiler computes the assignment `ps2 = ps3` and stores the result in `ps2`, which is passed as an input to the assignment `ps1 = ps2`.
  7. The compiler computes the assignment `ps1 = ps2` and stores the result in `ps1`.
  8. The return value `ps1` is not used because the above chain terminates, but the return value exists and could be used to extend the chain.

- Hence at every step of the chain, the return value of the assignment operation is a reference to the object on the left side of the assignment operation.
- For a user-defined C++ class, the required reference is `*this`.
- Therefore the return type is a reference `PStringsDC&` and the return value is `*this`.
- **The "`this`" pointer is rarely used in programming.**
- **The assignment operator is almost the only occasion where the "`this`" pointer is required.**

16 Self-assignment

- The test "`if (this == &rhs)`" tests for **self-assignment.**
- **Self-assignment means the objects on the left and right are the same object.**
- It is perfectly possible to write the statement `ps1 = ps1`.
- Then if we "deallocated the existing memory" in the left hand object `ps1` we would deallocate in the right hand object also, and the result would obviously cause problems.
  1. Hence the self-assignment test "`if (this == &rhs)`" is a check if the memory addresses of the objects on the left and right side of the assignment are equal.
  2. If true, then the left and right objects are the same object and there is nothing to do.
  3. Just `return *this` and exit.

16.1 Self-assignment: a personal comment

- Self-assignment is not as stupid as it sounds.
- It is not the result of careless or bad programming.
- It actually happened and caused a bug in a financial software system at one of my jobs.
- Admittedly, one does not typically see coding statements such as "$x = x$" but in a large project, with complicated code, what *did* happen, *and it was in a company where I was employed, so I speak from personal experience*, was to copy the contents of two pointers:

```
*p1 = *p2;
```

- By the time the program execution reached the above line, there was no telling what `p1` and `p2` were pointing to. Their addresses had been set in far away parts of the code.
- And it indeed happened, in one case, that `p1` and `p2` pointed to the same memory address.
- This caused a bug because it had been forgotten to check for self-assignment.
- The memory in `*p1` was deallocated, which of course also deallocated `*p2`, and the assignment operation then caused a memory fault.
- **The test for self-assignment is essential.**

17 Assignment operator, part 4

- Hence the procedure to write the assignment operator is as follows.

  1. Test for self-assignment.
  2. If true, `return *this` and exit.
  3. Else deallocate the existing memory in the destination object.
  4. Allocate fresh memory to match the memory in the right hand ("source") object.
  5. Perform the deep copy.
  6. At the end, `return *this` and exit.

- Here is the code for the assignment operator for the class `PStringsDC` again, with comments.

```
PStringsDC& operator= (const PStringsDC &rhs)
{
  if (this == &rhs) return *this;       // test for self-assignment

  length = rhs.length;
  delete [] pstr;                       // release existing memory
  pstr = new string[length];
  for (int i = 0; i < length; ++i) {
    pstr[i] = rhs.pstr[i];              // deep copy
  }
  return *this;                         // return reference to destination object
}
```

- If the above assignment operator is written for the class `PStringsDC`, the program displayed in Sec. 11 will not crash.

18 Syntactic sugar: copy or assign?

- Consider the instantiations below.

```
int n = 3;
string s[] = {"a", "b", "c"};
PStringsDC ps1(n, s);
PStringsDC ps2(ps1);           // copy
PStringsDC ps3 = ps1;          // copy or assign ?
```

- Clearly `ps2` is a copy of `ps1`. The copy constructor is invoked.
- ***What about*** `ps3`***?***
- Is the statement "`ps3 = ps1`" an assignment?
  1. *No.* Despite its appearance, it is not an assignment.
  2. The object `ps3` is a **brand new object which is being instantiated.**
  3. **Therefore the statement "`ps3 = ps1`" invokes the copy constructor.**
- The statement "`ps3 = ps1`" is an example of **syntactic sugar.**
- **Syntactic sugar** is just a convenience, an equivalent expression provided by the language.
- The following two statements are equivalent ways to invoke the copy constructor.

```
// p1 already constructed
PStringsDC p2(p1);          // equivalent
PStringsDC p2 = p1;         // equivalent
```

19 Private copy constructor and assignment operator

- **We can make the copy constructor and assignment operator private.**
- We can make anything private.
- Typically, they go together. If the copy constructor is private, the assignment operator should also be private.
    1. The consequence is that outside calling applications cannot make copies of objects.
    2. This is important way to control who/what can instantiate objects of a class.
    3. Many projects I have worked on did this.
    4. I have written private copy constructors and assignment operators myself.
- In many projects, an object contains sensitive data.
    1. By using the `private` keyword, we can prevent unauthorized access to that data by external applications.
    2. By disabling copies of objects, we can go one step further and prevent external applications from making unauthorized copies of objects.
- It was previously stated that there can be many objects for a given class.
    1. It was also stated that a C++ class is similar in many respects to a standard data type such as `char`, `int`, `double`, etc.
    2. Here we see how a user-defined class is different from the standard data types such as `char`, `int`, `double`, etc.
    3. We can create as many copies of an `int` (or `double`, etc.) as we like.
    4. However, for a C++ class, we can control the copying of objects.

20 Private destructor

- **We can make the destructor private.**
- As stated above, we can make anything private.
- C++ allows us to make the destructor private.
- Admittedly this is a curious thing to do.
- I have never seen a private destructor in any project on which I have worked.

21 A personal comment on the Big Three

- According to the textbooks, the Big Three of C++ classes are the **copy constructor**, the **assignment operator** and the **destructor**.
- All three are automatically generated by the compiler for a user-defined C++ class.
- The claim by the experts is that if any one of three needs to be explicitly written for a C++ class, then all three should be written for the class, i.e. they go together.
- This is also called the **Rule of Three.**
- Personally, it has never really occurred to me in my own career that the three went together.
- That probably demonstrates that I am not an expert.
- All of my C++ programming career has been to write software for financial companies, and this fact has undoubtedly influenced my expertise with C++.

  1. *The copy constructor and assignment operator do go together.* That much is obvious. They both exist to make copies of objects.
  2. However, in my own programming career, the destructor has played a different role.
  3. I have found the destructor to be far more important than the copy constructor and assignment operator.
  4. In fact, in many if not most of my projects, the creation of copies was disabled.
  5. For a trading system in a financial company, there is only one trading system, and it is internal to the company. There is no need for copies.
  6. Many financial business products contain sensitive data, and access to them is restricted.
  7. If copies are allowed, the person/application which created the object loses control of it.
  8. We do not want copies of objects with sensitive data floating around beyond our control, which others can edit without permission.
  9. Instead pointers are passed to calling applications, and more often than not they are `const` pointers: outside applications can read the non-private data, but cannot modify the contents without our permission.
  10. Classes in financial applications can be HUGE. (This is probably true in most real life applications.)
  11. Making copies, *especially deep copies*, is time consuming, and in financial trading a high speed of execution is critical. Making copies of objects is avoided unless necessary.
  12. Furthermore, writing a deep copy is a real pain.
  13. In large projects with multiple software developers, new data members (and methods) get added to classes all the time, and keeping the deep copy up to date is a headache.
  14. It is simpler to disable the making of copies altogether.
  15. Of course, the destructor must be kept up to date, but that is only one out of three.

- Furthermore, think about this:
  1. If a new data member is added to a class, and it *does not require dynamic memory allocation*, e.g. a `double` or `int`, *then the destructor does not need to be updated.*
  2. What would be the update to the destructor anyway?
  3. However, the deep copy in the copy constructor and assignment operator *must be updated*.
  4. Else the copied objects would be erroneous.
  5. The destructor (mostly) only needs to deal with dynamically allocated memory.
  6. The deep copy needs to be updated for *everything.*
- Hence keeping the destructor up to date is simpler than keeping the deep copy of the copy constructor and assignment operator up to date.
- Ultimately, the textbooks contain buzzwords such as the "Big Three" and the "Rule of Three" and you have to learn them and recite them in job interviews.
- However, the reality in your actual career may be very different.