Constructors and Other Tools

Computer Programming for Engineers (DASF003-41)

Instructor:

Sungjae Hwang (jason.sungjae.hwang@gmail.com)

This Week

- **Constructors (ctors) and destructors (dtors)**
  - Definitions
  - Calling conventions

- **Other Tools for classes**
  - `const` parameter modifier
  - Inline functions
  - Array decay

CONSTRUCTORS AND DESTRUCTORS

Constructors

- **A special sort of member function**
  - Automatically called when object declared
  - Generally, they're not allowed to call explicitly
- **Initialization of objects**
  - Initialize some or all member variables
  - Other many actions possible as well
  - Very useful tool: key principle of OOP

Constructor Definition

- **Constructors defined like any member function, except:**
  - Must have **same name** as class
  - Return type is not declared: cannot return a value, not even `void`
- **Constructor in public section**
  - It's called when objects are declared
  - Example:

```
class DayOfYear
{
public:
   // Constructor initializes month and day
   DayOfYear( int monthValue, int dayValue );

private:
   int month;
   int day;
};
```

Constructor Code

- **Constructor definition is similar to other member functions:**

```
DayOfYear::DayOfYear( int monthValue, int dayValue )
{
   month = monthValue;
   day = dayValue;
}
```

  - Identical names around the scope resolution operator (::)
    - Clearly identifies a constructor
  - No return type is declared

Initialization Section

Initialization Section (with ":")

- Previous definition equivalent to:

```
DayOfYear::DayOfYear(  int monthValue, int dayValue )
   : month(monthValue), day(dayValue)
{
   … // constructor body
}
```

- Second line called "**Initialization Section**"
- Check: Argument names can be the same as members' name.

Purposes of initialization section

- Default values
- Shorter version of a constructor body
- Initialization of **const** or **reference** member variables
  - Such members are not allowed to assign in the constructor body.

Constructor with initialization section


```
1 #include <iostream>
2 using namespace std;
3
4 class DayOfYear
5 {
6   public:
7     DayOfYear(int, int);
8     void output();
9   private:
10    int month = 1;
11    int day = 1;
12 };
13
14
15 DayOfYear::DayOfYear(int monthValue, int dayValue)
16   : month(monthValue), day(dayValue)
17 {
18   cout << "In the constructor DayOfYear(" << monthValue;
19   cout << "," << dayValue << ")" << endl;
20   month = monthValue;
21   day = dayValue;
22 }
23
24
25 /*
26 DayOfYear::DayOfYear(int month, int day)
27  // : month(month), day(day)
28 {
29   cout << "In the constructor DayOfYear(" << month;
30   cout << "," << day << ")" << endl;
31   //cout << this->month << endl;
32
33   cout << month << endl;
34   month = month;
35 } */
38 /* DayOfYear::DayOfYear(int monthValue, int dayValue)
40   : month(12), day(5){
42   month = monthValue;
43 }*/
46 void DayOfYear::output(){
47   cout << "day of ouput function" << endl;
48   cout << month << endl;
49 }
51 int main(){
52   DayOfYear christmas(12,25);
53   // What is the result? What does it mean?
54   christmas.output();
55   return 0;
56 }
```

Calling Constructors

Declare objects:

- The default look is similar to function calls.

```
DayOfYear date1(7, 4), date2(5, 5);
```

- Dynamic allocation with new also calls constructor

```
DayOfYear* p_date1 = new DayOfYear(7, 4);
DayOfYear* p_date2 = new DayOfYear(5, 5);
```

Objects are created here

- Constructor is called
- Values in parentheses passed as arguments to constructor
- Member variables month, day initialized:

date1.month → 7 date2.month → 5
date1.day → 4 date2.day → 5

constructor (heap and stack)


```
1 #include <iostream>
2 using namespace std;
3
4 class DayOfYear{
6   public:
7     DayOfYear(int, int);
8     void output();
9   private:
10    int month = 1;
11    int day = 1;
12 };
13
14 DayOfYear::DayOfYear(int monthValue, int dayValue)
15 {
16   cout << "In the constructor DayOfYear(" << monthValue;
17   cout << "," << dayValue << ")" << endl;
18   month = monthValue;
19   day = dayValue;
20 }
21
22 void DayOfYear::output(){
23    cout << "day of ouput function" << endl;
24    cout << month << endl;
25 }
26
27 int main()
28 {
32   DayOfYear obj1(5, 10);
33   DayOfYear* obj2 = new DayOfYear(2, 2);
34
35   cout << "obj1: " << &obj1 << endl;
36   cout << "obj2: " << obj2 << endl;
37
42   return 0;
43
44 }
45
```

Constructor Equivalency: Illegal

Consider:

```
DayOfYear date1, date2
date1.DayOfYear(7, 4); // ILLEGAL!
date2.DayOfYear(5, 5); // ILLEGAL!
            vs
DayOfYear date1(7, 4), date2(5, 5); // OK!
```

Seemingly OK…

- but, CANNOT call constructors like other member functions!

Explicit call to constructor

```
DayOfYear date1, date2
date1 = DayOfYear(7, 4); // OK!
```

Order of Class Member Initialization

- 1) Non-static member initializers
  - explained previously
- 2) Initialization section
- 3) Constructor body

```
class DayOfYear
{
public:
   DayOfYear( int monthValue, int dayValue )
   : month(2), day(2)
   {
     month = monthValue;
     day   = dayValue;
   }
private:
   int month = 1;
   int day = 1;
}
```

Additional Purpose of Constructor

Validate the data!

- Not just initialize data; body doesn't have to be empty in initializer version
- Ensure only appropriate data is assigned to class private member variables

```
class DayOfYear
{
public:
   DayOfYear( int monthValue, int dayValue )
   : month(2), day(2)
   {
     if ((month < 1) || (month > 12)){
       cout << "Illegal month value!\n";
       exit(1);
     }
   }
}
```

consturctor (validation)


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

class DayOfYear
{
  public:
    DayOfYear(int, int);
    void output();
  private:
    int month = 1;
    int day = 1;
};

DayOfYear::DayOfYear(int monthValue, int dayValue)
  : month(monthValue), day(dayValue)
{
  if ((month < 1) || (month > 12)){
    cout << "Illegal month value!" << endl;
    exit(1);
  }
  if ((day < 1) || (day > 31)){
    cout << "Illegal day value!" << endl;
    exit(1);
  }
}
void DayOfYear::output(){
  cout << "Month: " << month << " Day:" << day << endl;
}

int main(){
  DayOfYear christmas(12,25);
  christmas.output();
  return 0;
}
```

Constructor in Private Section (Advanced)

- **Constructor in private section is not common in general**
  - If it's private, it can never be used for declaring objects!

- **Used for special purposes**
  - e.g., restrict certain types of constructors
    - Such as copy constructor or particular parameter types
    - Ex) prevent calling `DayOfYear(1.1, 3.0)`

Private constructor


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

class DayOfYear
{
  public:
    DayOfYear(int, int) {}
    void output();

  private:
    // We cannot call private constructors
    DayOfYear(int, double) {}
    DayOfYear(double, int) {}
    DayOfYear(double, double) {}

    int month;
    int day;
};

int main(){
  //DayOfYear a(5.5, 5.5);
  DayOfYear s(5, 5);
  return 0;
}
```

Overloaded Constructors

- **Can overload constructors just like other functions**
  - Recall: a signature consists of:
    - Name of function
    - Parameter list
    - (Return type is not included)

- **Provide constructors for all possible argument-lists**
  - Particularly "how many"
  - Using default arguments can be effective in reducing the potential cases.

Class with Constructors Example

- Display 7.1 Class with Constructors (1 of 4)

```
#include <iostream>
#include <cstdlib> //for exit
using namespace std;

class DayOfYear
{
public:
   DayOfYear( int monthValue, int dayValue );
   // initializes the month and day to arguments
   DayOfYear( int monthValue );
   // initializes the date to the first of the given month
   DayOfYear( );
   // initializes the date to January 1
   void output( );
```

Class with Constructors Example

- Display 7.1 Class with Constructors (2 of 4)

```
private:
   int month;
   int day;
   void testDate( );
};
int main( )
{
   DayOfYear date1(2, 21), date2(5), date3;
   cout << "Initialized dates:\n";
   date1.output( ); cout << endl;
   date2.output( ); cout << endl;
   date3.output( ); cout << endl;
   return 0;
}
```

Class with Constructors Example

Display 7.1 Class with Constructors (3 of 4)

```
DayOfYear::DayOfYear( int monthValue, int dayValue )
   : month(monthValue), day(dayValue)
{
   testDate( );
}
DayOfYear::DayOfYear( int monthValue )
   : month(monthValue), day(1)
{
   testDate( );
}
DayOfYear::DayOfYear( ) : month(1), day(1)
{/*Body intentionally empty.*/}
```

Class with Constructors Example

Display 7.1 Class with Constructors (4 of 4)

```
// uses iostream and cstdlib:
void DayOfYear::testDate( )
{
   if((month < 1) || (month > 12))
   {
     cout << "Illegal month value!\n";
     exit(1);
   }
   if((day < 1) || (day > 31))
   {
     cout << "Illegal day value!\n";
     exit(1);
   }
}
```

Constructor with No Arguments

Standard functions with no arguments:

- Called with syntax: `func_name();`
  - `e.x. testFunc();`
- Including empty parentheses

Object declarations with no "initializers":

```
DayOfYear date1;      // Yes!
DayOfYear date();     // NO!
date1 = DayOfYear(); // Yes! creates an anonymous instance
```

Anonymous objects

- A value that has no name
  - `cout << 1 + 2 << endl; // 3 is placed in an anonymous object.`

Default Constructor

- **One constructor should always be defined**
- **Default constructor**
  - Defined as: constructor with no arguments
  - Also, does nothing in the body
- **Auto-Generated? Yes and No**
  - If no constructors AT ALL are defined → Yes
  - If any constructors are defined → No
- **If no default constructor with other constructors:**
  - Cannot declare with no initializers
  - `MyClass myObject; // NO!`

Default constructor


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

class DayOfYear
{
  public:
    /*
    DayOfYear() {
      cout << "1 In the constructor DayOfYear" << endl;
      month = 1;
      day = 1;
    }
    */

   // DayOfYear(int, int);

    void output();
  private:
    int month;
    int day;
};
/*
DayOfYear::DayOfYear(int monthValue, int dayValue)
  : month(monthValue), day(dayValue)
{
  cout << "2 In the constructor DayOfYear" << endl;
}
*/

void DayOfYear::output(){
  cout << "Month: " << month << " Day:" << day << endl;
}

int main(){
  DayOfYear obj1;
  //DayOfYear obj2(2,3);
  obj1.output();
  //obj2.output();
}
```

Copy Constructor

- **Constructor can retrieve another object of the same class**

```
DayOfYear holiday = DayOfYear(7, 4);
```

  - `DayOfYear(7,4)` returns "anonymous object", which can then be assigned

- **Copy constructor**
  - A special constructor, having a single parameter of const CLASS&.

```
class DayOfYear
{
public:
   DayOfYear( const DayOfYear& other )
   {
   this->month = other.month;
   this->day = other.day;
   }
};
```

Copy Constructor

DEMO

Copy constructor is called when:

- Class object declared and initialized to other object

```
DayOfYear new_year(1, 1);
DayOfYear holiday = new_year; // calls DayOfYear(new_year)
```

- When argument of class type is "plugged in" as actual argument to call-by-value parameter

```
void print_day(DayOfYear day){ /* print */ }
int main(){
   DayOfYear new_year(1, 1);
   print_day(new_year); // pass DayOfYear(new_year)
}
```

- When a function returns copy of the object

Copy constructor


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

class CPE
{
private:
    int studentNo;
public:
    CPE(int no) {
      cout << "In Normal Constructor" << endl;
      studentNo = no;
    }

    // Copy constructor
    CPE(const CPE &p1) {
      cout << "In Copy Constructor" << endl;
      studentNo = p1.studentNo;
    }

    int getStudentNo() {  return studentNo; }
};

int main()
{
    // Normal constructor is called
    CPE obj1(52);

    // Copy constructor is called
    CPE obj2 = obj1;

    cout << "obj1 student No: " << obj1.getStudentNo() << endl;
    cout << "obj2 student No: " << obj2.getStudentNo() << endl;

    return 0;
}
```

Default Copy Constructor

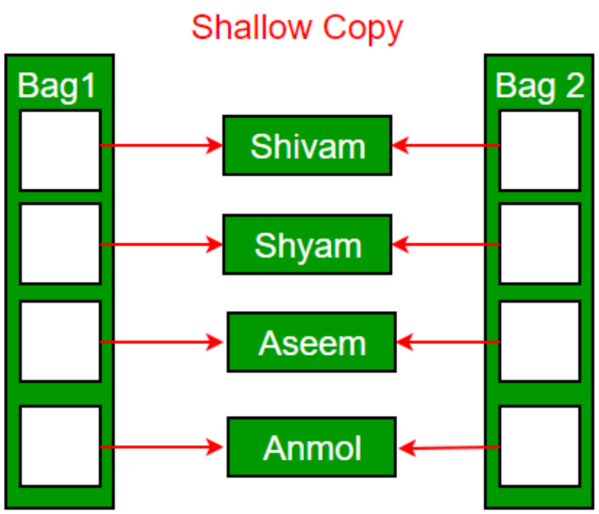
Default copy constructor

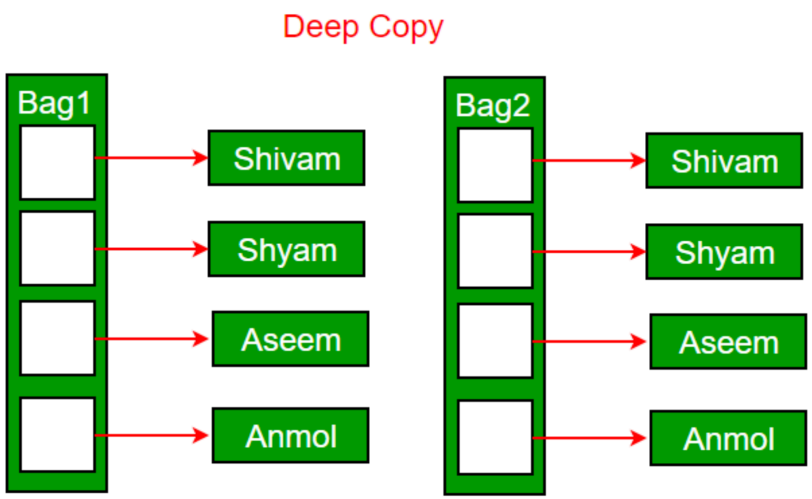
- Like default "=", performs member-wise copy
  - This is **shallow copy**, where pointer addresses are only copied.
  - The values of the pointers are shared with the other objects.

- For pointers, to copy the value
  - We need to allocate memory and then copy the value to the new addresses.
  - For this, write your own copy constructor for **deep copy**!

See the next page for shallow vs. deep copy

Deep Copy vs. Shallow Copy





https://www.geeksforgeeks.org/copy-constructor-in-cpp/

Deep Copy vs. Shallow Copy

- Given a class new_int,

```
class new_int{
   int* ptr;
   new_int( const new_int& ); } // copy constructor
```

- Shallow copy
  - copies pointer itself without copying the value of int

```
new_int::new_int( const new_int& other ){
   ptr = other.ptr; } // content of ptr is shared with other
```

  - However, we actually want to copy the value.
- Deep copy
  - allocates a new int, and copy the content like this:

```
new_int::new_int( const new_int& other ){
   ptr = new int;
   *ptr = *other.ptr; } // content is copied from other
```

Deep Copy & Shallow Copy


```
1 #include<iostream>
2 #include <cstring>
3 using namespace std;
4
5 class CPE
6 {
7 private:
8     int studentNo;
9     char *instructor;
10 public:
11     CPE(int no, const char *name) {
12       cout << "In Normal Constructor" << endl;
13       studentNo = no;
14       int size = strlen(name);
15       instructor = new char[size+1];
16       strcpy(instructor, name);
17     }
18
19
20     // Copy constructor : Shallow Copy
21     CPE(const CPE &p1) {
22       cout << "In Copy Constructor" << endl;
23       studentNo = p1.studentNo;
24       instructor = p1.instructor;
25     }
26 /* // Copy constructor : Deep Copy
29     CPE(const CPE &p1) {
30       cout << "In Copy Constructor" << endl;
31       studentNo = p1.studentNo;
33       instructor = new char[strlen(p1.instructor)+1];
34       strcpy(instructor, p1.instructor);
35     }*/
38     void changeIns(const char *name) {
39       strcpy(instructor, name); }
42     int getStudentNo() {  return studentNo; }
43     void getInsName() {  cout << instructor << endl; }
44 };
46 int main(){
48     // Normal constructor is called
49     CPE obj1(52, "HSJ");
51     // Copy constructor is called
52     CPE obj2 = obj1;
54     obj1.getInsName();
55     obj2.getInsName();
57     obj1.changeIns("AAA");
59     obj1.getInsName();
60     obj2.getInsName();
61
62     return 0;
63 }
```

Constructor Delegation (C++11)

- C++11 allows one constructor to invoke another

```
Coordinate::Coordinate(int xval, int yval) : x(xval), y(yval)
{ }

Coordinate::Coordinate() : Coordinate(99,99)
{ }
```

  - The default constructor invokes constructor to initialize x and y to 99,99

Need for Destructor

- **Dynamically-allocated variables**
  - Do not go away until "deleted“.
- **If pointers are members**
  - They are dynamically allocated with "real" data.
  - Must have ways to "deallocate" when object is destroyed.

- **Answer: destructor!**

Destructors

- **Opposite of constructor**
  - Automatically called when an object become out-of-scope.
  - Default version does not remove dynamically allocated variables.

- **Can be defined similar to constructors, but need to add "~".**

```
class DayOfYear
{
public:
   ~DayOfYear()
   {
   // when necessary, deallocate pointers
   // do other clean-up
   }
};
```

Destructor


```
1 #include <iostream>
2 using namespace std;
3
4
5 class DayOfYear
6 {
7   public:
8     DayOfYear() {month = 1; day = 1;}
9     DayOfYear(int, int);
10    ~DayOfYear();
11
12  private:
13    int month;
14    int day;
15 };
16
17 DayOfYear::DayOfYear(int a, int b)
18 {
19   cout << "constructing " << endl;
20   month = a;
21   day = b;
22 }
23
24 DayOfYear::~DayOfYear()
25 {
26   cout << "destructing " << endl;
27 }
28
29 void test(){
30   cout << "in test" << endl;
31 }
32
33 int main(){
34   cout << "object is created" << endl;
35   DayOfYear a(5,5);
36   return 0;
37 }
```

Summary

- Constructor
- Default Constructor
- Shallow & Deep Copy
- Constructor Overload
- Destructor

OTHER TOOLS IN CLASS & ARRAY DECAY

Parameter Passing Methods

Efficiency of parameter passing

- Call-by-value
  - Requires copy be made → Overhead
- Call-by-reference
  - Placeholder for actual argument
  - Most efficient method
- Negligible difference for simple types
- For class types → clear advantage

Call-by-reference desirable

- Especially for "large" data, like class types

The `const` Parameter Modifier

Large data types (typically classes/structures)

- Desirable to use pass-by-reference
- Even if function will not make modifications

Protect argument

- Place keyword **const** before type :
  - Also called constant call-by-reference parameter
- Makes parameter "read-only"
- Attempt to modify parameter results in compiler error

```
//constant call-by-reference parameter
bool isLarger(const BankAccount& account1)
{ }
```

Inline Functions

- **Compiler attempts to insert the code in place of call**
  - Eliminates overhead of function call
  - More efficient, but not always guaranteed to be inserted.

- **Usage:**
  - Use keyword inline in function declaration and function heading
  - Use for short functions in general

```
inline int add( int a, int b ){ return a+b; }
```

Inline Member Functions

Member function definitions

- Typically defined separately, in different file (*.h and *.cpp)
- **In-class definition** makes function **"inline"** by default

```
class DayOfYear
{
public:
   int test(){ return 1+2; } // this function is made inline
   void NoInlineFunction();
};

void DayOfYear::NoInlineFunction(){} // no inline
```

Use it for very short functions only

- More efficient: call stacks are not generated.
- If too long → actually less efficient!
  - All inline functions are include in binary

Array Decay

Loss of type and dimensions of an array

- When we pass array as pointer in function call
- First address to the array is passed

```
void f1(int *arr){
  // array decay here
  // array information is lost
  // e.x size of array
}

int main(){
  int arr[5] = {1,2,3,4,5};
  f1(arr);
}
```

Array Decay


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

void decayFunc(int *p)
{
    cout << "Size in decayFunc : ";
    cout << sizeof(p) << endl;
}

int main()
{
    int a[5] = {1, 2, 3, 4, 5};

    cout << "Actual size: ";
    cout << sizeof(a) <<endl;

    decayFunc(a);

    return 0;
}

```

Prevent Array Decay

- Two methods to avoid array decay
  - Pass size of array
  - Pass array by reference

```
void f1(int *arr, int num){
   for(int i=0; i<num; i++)
      cout << arr[i] << endl;
}

void f2(int (&arr)[5]){
   for(int a : arr)
      cout << a << endl;
}

int main(){
  int arr[5] = {1,2,3,4,5};
  f1(arr,5);
  f2(arr);
}
```

Array Decay : range-based for loop (1)

Loss of type and dimensions of an array

- When we pass array as pointer in function call
- First address to the array is passed

```
void arrayModify(int *arr, int num){
     for(int a : arr)
       std::cout << a << std::endl;
}

int main(){
  int arr[5] = {1,2,3,4,5};
  arrayModify(arr,5);
}
```

Compiler error :
no viable 'begin' function available

Array Decay : range-based for loop (2)

cppreference.com Create account Search

Page Discussion View Edit History

C++ C++ language Statements

Range-based for loop (since C++11)

Executes a for loop over a range.

Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.

Syntax

attr(optional) **for** (*init-statement*(optional) *range-declaration* **:** *range-expression*)
loop-statement

range-expression - any expression that represents a suitable sequence (either an array or an object for which begin and end member functions or free functions are defined, see below) or a braced-init-list.

https://en.cppreference.com/w/cpp/language/range-for

Array Decay : range-based for loop (3)

cppreference.com Create account array Search

Page Discussion View Edit History

C++ Containers library `std::vector`

std::vector

Iterators

`begin` `cbegin` (C++11)	returns an iterator to the beginning (public member function)
`end` `cend` (C++11)	returns an iterator to the end (public member function)

cppreference.com Create account Search

Page Discussion View Edit History

C++ Containers library `std::array`

std::array

Iterators

`begin` `cbegin` (C++11)	returns an iterator to the beginning (public member function)
`end` `cend` (C++11)	returns an iterator to the end (public member function)

Array Decay : range-based for loop (4)

- **We know the size of the array!**
  - Cast pointer into array pointer or array reference

```
void arrayModify(int *arr, int num){
      //array pointer
      for(int a : *(int(*)[10])arr)
         std::cout << a << std::endl;
                   //or
      //array reference
      for(int& a : (int(&)[num])*arr)
         std::cout << a << std::endl;
}

int main(){
  int arr[5] = {1,2,3,4,5};
  arrayModify(arr,5);
}
```

Summary

- Parameter Passing Methods
- Inline member functions
- Array Decay