Constructors and Other Tools

Computer Programming for Engineers (DSAF003-42)

Instructor:

Youngjoong Ko (nlp.skku.edu)

This Week

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

- **Other Tools for classes**
  - `const` parameter modifier
  - Inline functions
  - Static member data

CONSTRUCTORS AND DESTRUCTORS

Constructors

- A special sort of member function
  - Automatically called when object is 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

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 Equivalency: Illegal

- Consider:

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

  - CANNOT call constructors like other member functions!
  - It's always called **implicitly**.

constructor


```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
  cout << "In the constructor DayOfYear(" << monthValue;
  cout << "," << dayValue << ")" << endl;
  month = monthValue;
  day = dayValue;
}
int main()
{
  DayOfYear christmas(12,25);
  christmas.output();

  int month(5);
  int* day = new int(11);
  DayOfYear* birthday = new DayOfYear(month, *day);
  birthday->output();
}
```

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**" 초기화 섹션.
  - Check: Argument names can be the same as members' name. 인수이름 ≠ 멤버 이름.
- Purposes of initialization section
  - Default values 기본값 정하기.
  - Shorter version of a constructor body
  - Initialization of **const** or **reference** member variables
    - Such members are not allowed to assign in the constructor body.

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;
}
```

Constructor with initialization section


```
#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(2), day(2)
{
  cout << "In the constructor DayOfYear(" << monthValue;
  cout << "," << dayValue << ")" << endl;
  month = monthValue;
  day = dayValue;
}
```

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

int main()
{
  DayOfYear christmas(12,25);
  // What is the result? What does it mean?
  christmas.output();
}
```

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
  - 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 input( );
    void output( );
    int getMonthNumber( );
    // returns 1 for January, 2 for February, etc.
```

Class with Constructors Example

- Display 7.1 Class with Constructors (2 of 4)

```
    int getDay( );
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;
    date1 = DayOfYear(10, 31);
    cout << "date1 reset to the following:\n";
    date1.output( ); cout << endl;
    return 0;
}
```

<Definitions of the other member functions are the same as in prior examples>

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);
    }
}
```

```
Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31
```

Constructor with No Arguments

생성자는 인자 X!

- Standard functions with no arguments:
  - Called with syntax: `DayOfYear();`
  - Including empty parentheses

- Object declarations with no "initializers":

```
DayOfYear date1;      // Yes!
DayOfYear date();     // NO!
```

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 there is 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;
}
```

DayofYear A; 이때 여기로

DayofYear A(12,25); 이때 여기로

Copy Constructor

복사본 생성시 호출되는 생성자.

- Constructor can retrieve another object of the same class

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

의역해서 할당

  - 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

- Maybe automatically 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)
```

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

- 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
```

Constructor Delegation (C++11)

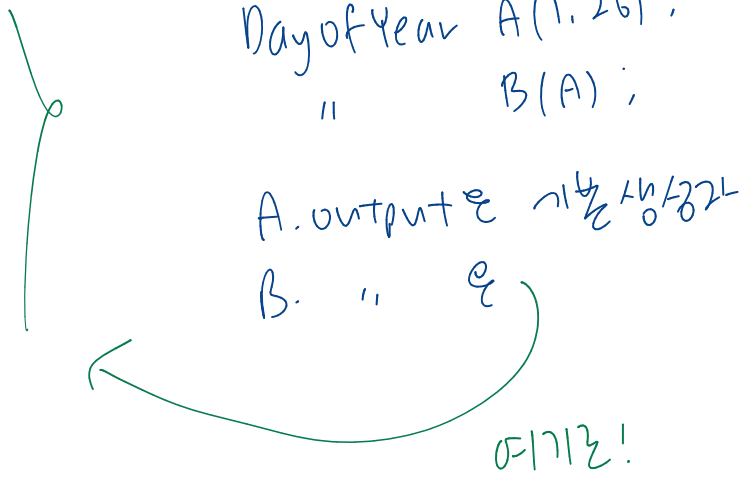
- C++11 allows one constructor to invoke another

```
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

Copy constructor


```
class DayOfYear
{
  public:
    DayOfYear();
    DayOfYear(int, int);

    DayOfYear( const DayOfYear& other )
    {
      cout << "In the copy constructor " ;
      this->month = other.month;
      this->day = other.day;
    }
    void output() const;
  private:
    int month;
    int day;
};
```



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.

객체가 X될 때 메모리 반납 필요해야.

- Answer: destructor!

Destructors

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

  ↳ 이렇게하면 동적 할당된 변수 제거 X.
- Can be defined similar to constructors, but need to add "~".

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

Destructor


```
class DayOfYear
{
  public:
    DayOfYear() {month = 1; day = 1;}
    DayOfYear(int, int);
    ~DayOfYear();

    void output();
  private:
    int month;
    int day;
};

DayOfYear::~DayOfYear()
{
  cout << "Destructing ";
  output();
}
```

OTHER TOOLS

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

Call-by-reference for class


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

class Nara
{
public :
char name[20]; // 공개 멤버 변수
char soodo[20]; // 공개 멤버 변수
int ingoo; // 공개 멤버 변수
};

void Showdata(const Nara &s) // 전달되는 인자를 참조자로 받고 있음.
{
cout << " *** 국가 정보 *** " << endl;
cout << " 국가 이름 : " << s.name << endl;
cout << " 수도 : " << s.soodo << endl;
cout << " 인구 : " << s.ingoo << endl;
}
```

```
int main(void)
{
Nara country;

cout << " 국가 이름 : ";
cin >> country.name;

cout << " 수도 : " ;
cin >> country.soodo;

cout << " 인구 : ";
cin >> country.ingoo;

Showdata(country);

return 0;
}
```

The `const` Parameter Modifier

- Large data types (typically classes/structures)
  - Desirable to use pass-by-reference
  - function will not make modifications 바꾸지 않음

- 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

Inline Member Functions

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

클래스 안의 정의하는 거

```
class DayOfYear
{
public:
   DayOfYear(){ ... } // 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

Static Members

- Static member variables
  - Place keyword `static` before type
  - All objects of class "share" one copy
  - Useful for "tracking": e.g., class instance counter
- Out-of-class definition (instancing a static variable)
  - We also need to declare their definitions outside the class definition.

```
// in Server.h
class Server
{
private:
   static int turn; // this is just a declaration
};

// in Server.cpp
int Server::turn = 0; // now, Server::turn is allocated
```

Static Functions

static으로 선언하면 클래스의 특정 개체에 독립적으로 만들 수 있.

- Member functions can be static
  - By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.
  - Can then be called outside class
    - From non-class objects (e.g., `Server::getTurn();`)
    - As well as via class objects (e.g., `my_server.getTurn();`)

- They can use only static data and functions!

정적(data, func)이 아니면 멤버 함수에서 접근 X

  - Members that are not static(data, functions) are not accessible inside the static member functions.
    - Because there's no way of knowing from which class instances the non-static members come.

Static Members Example

- Display 7.6 Static Members (1 of 4)

```
#include <iostream>
using namespace std;
class Server
{
public:
    Server(char letterName);
    static int getTurn( );
    void serveOne( );
    static bool stillOpen( );
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};
int Server::turn = 0;
int Server::lastServed = 0;
bool Server::nowOpen = true;
```

Static Members Example

- Display 7.6 Static Members (2 of 4)

```
int main( )
{
    Server s1('A'), s2('B');
    int number, count;
    do
    {
        cout << "How many in your group? ";
        cin >> number;
        cout << "Your turns are: ";
        for (count = 0; count < number; count++)
            cout << Server::getTurn( ) << ' ';
        cout << endl;
        s1.serveOne( );
        s2.serveOne( );
    } while (Server::stillOpen( ));
    cout << "Now closing service.\n";
    return 0;
}
```

Static Members Example

- Display 7.6 Static Members (3 of 4)

```
Server::Server(char letterName) : name(letterName)
{/*Intentionally empty*/}

//Since getTurn and stillOpen is static, only static members
//can be referenced in here.
int Server::getTurn( ) { turn++; return turn; }

bool Server::stillOpen( ) { return nowOpen; }

void Server::serveOne( )
{
    if (nowOpen && lastServed < turn)
    {
        lastServed++;
        cout << "Server " << name
             << " now serving " << lastServed << endl;
    }
    if (lastServed >= turn) //Everyone served
        nowOpen = false;
}
```

Static Members Example

- Display 7.6 Static Members (4 of 4)

```
How many in your group? 3
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? 2
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? 0
Your turns are:
Server A now serving 5
Now closing service.
```

Static members


```
class Obj {
  public:
    Obj():name("not set") {count++;}
    Obj(string name):name(name) {count++;}
    static void printCounter();
    void printInfo();
  private:
    string name;
    static int count;
};

int Obj::count = 0;

void Obj::printInfo() {
  cout << name << ": " << count << endl;
}

void Obj::printCounter() {
  //cout << name << ": " << count << endl;
  cout << count << endl;
}
```

이경우 name에 접근이 X.