



CS-202

C++ Classes – Midterm Recapitulation

C. Papachristos


Autonomous Robots Lab
University of Nevada, Reno



Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday	Sunday
			Lab (8 Sections)		
	RECAP CLASS		CLASS		
PASS Session	PASS Session	Project DEADLINE	MIDTERM	PASS Session	PASS Session



Your 6th Project Deadline is this Wednesday 3/13.

Your **Midterm** will be held this Thursday 3/14.

- A Midterm Sample has been announced since last week.
- Lectures, Labs, PASS sessions (with a Sunday extra), dedicated to recapitulation.
- Final recap, questions & Midterm Sample overview today!

Today's Topics

C++ Classes Cheatsheet

- Declaration
- Members, Methods, Interface
- Implementation – Resolution Operator (`::`)
- Instantiation – Objects
- Object Usage – Dot Operator (`.`)
- Object Pointer Usage – Arrow Operator (`->`)
- Classes as Function Parameters, Pass-by-Value, by-(`const`)-Reference, by-Address
- Protection Mechanisms – `const` Method signature
- Classes – Code File Structure
- Constructor(s), Initialization List(s), Destructor
- `static` Members – Variables / Functions
- Class `friend`(s)
- Keyword `this`
- Operator Overloading
- Class/Object Relationships – Composition, Aggregation,
- Inheritance – Rules, Method Overriding
- Polymorphism – Base Class Pointers (Abstract Data Structure(s) support)
- `virtual` Methods – Static vs Dynamic Binding
- Pure `virtual` Methods – Abstract Classes

Functions - Examples

Implement Helper Functions

```
const int STR_MAX = 256;
```

```
void strcpy(char * dst, const char * src)
{
    while (*dst++ = *src++);
}
```

```
int strlen(const char * str)
{
    const char * s = str;
    for ( ; *s; ++s);
    return s - str;
}
```

➤ For C-string types
(**char** arrays – i.e. **char ***)

```
int strcmp(const char * s1, const char * s2)
{
    while (*s1 == *s2++){
        if (!*s1++){
            return 0;
        }
    }
    return *s1 - *--s2;
}
```

Functions - Examples

Implement Helper Functions

➤ Similarly for any other type
(`float *`, `double *`, etc.)

```
void intcpy(int * dst, const int * src, int size){
    while (--size>=0){
        *dst++ = *src++;
    }
}
```

```
void intcmp(const int * arr1, const int * arr2, int size){
    while (--size>=0){
        int res = *arr1++ - *arr2++;
        if (res){ return res; }
    }
    return 0;
}
```

```
void intprint(std::ostream & os, const int * arr, int size){
    while (--size>=0){
        os << *arr++;
    }
}
```

Classes - Examples

Declare a Class, Implement some Methods

```
const char * BOOK_DEFAULT_TITLE = "notitle";
const size_t BOOK_ISBN_LEN = 13;
const int BOOK_DEFAULT_ISBN[BOOK_ISBN_LEN] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
const char * BOOK_DEFAULT_RENTER = "norenter";

class Book {

    friend std::ostream & operator<<(std::ostream & os, const Book & b);

public:

    Book();
    Book(const char * title, const int * isbn = BOOK_DEFAULT_ISBN,
          const char * renter = BOOK_DEFAULT_RENTER); //use default parameters in list
    Book(const Book& other);
    ~Book();

    Book & operator=(const Book & rhs);
    void setTitle(const char * title);      const char * getTitle() const;
    void setIsbn(const int * isbn);         const int * getIsbn() const;
    ...
}
```

Classes - Examples

Declare a Class, Implement some Methods

```
...
bool getAvailable() const ;
const char * getRenter() const ;

bool valid() const;
bool operator+(const char * renter);
void free();

static int getIdgen();

private:

    const size_t m_id;

    char m_title[STR_MAX];
    int m_isbn[BOOK_ISBN_LEN];
    bool m_available;
    char m_renter[STR_MAX];

    static size_t s_idgen;

};
```


Classes - Examples

Declare a Class, Implement some Methods

```
const char * BOOK_DEFAULT_TITLE = "notitle";
const size_t BOOK_ISBN_LEN = 13;
const int BOOK_DEFAULT_ISBN[BOOK_ISBN_LEN] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
const char * BOOK_DEFAULT_RENTER = "norenter";

class Book {

    friend std::ostream & operator<<(std::ostream & os, const Book & b);

public:

    Book();
    Book(const char * title, const int * isbn = BOOK_DEFAULT_ISBN,
          const char * renter = BOOK_DEFAULT_RENTER); //use default parameters in list
    Book(const Book& other);
    ~Book();

    Book & operator=(const Book & rhs);
    void setTitle(const char * title);
    void setIsbn(const int * isbn);
    ...
    const char * getTitle() const;
    const int * getIsbn() const;
```

➤ Global **constants** can be of
simple type,
array-type,
C-string type.

Classes - Examples

Declare a Class, Implement some Methods

➤ Strict methodology about overloading certain operators, managing access, etc.

```
const char * BOOK_DEFAULT_TITLE = "notitle";
const size_t BOOK_ISBN_LEN = 13;
const int BOOK_DEFAULT_ISBN[BOOK_ISBN_LEN] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
const char * BOOK_DEFAULT_RENTER = "norenter";

class Book {

    friend std::ostream & operator<<(std::ostream & os, const Book & b);

public:

    Book();
    Book(const char * title, const int * isbn = BOOK_DEFAULT_ISBN,
          const char * renter = BOOK_DEFAULT_RENTER); //use default parameters in list
    Book(const Book& other);
    ~Book();

    Book & operator=(const Book & rhs);
    void setTitle(const char * title);
    void setIsbn(const int * isbn);
    ...
    const char * getTitle() const;
    const int * getIsbn() const;
```

Classes - Examples

Declare a Class, Implement some Methods

➤ Default parameters
(if specified) appear only in
function declarations !

```
const char * BOOK_DEFAULT_TITLE = "notitle";
const size_t BOOK_ISBN_LEN = 13;
const int BOOK_DEFAULT_ISBN[BOOK_ISBN_LEN] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
const char * BOOK_DEFAULT_RENTER = "norenter";

class Book {

    friend std::ostream & operator<<(std::ostream & os, const Book & b);

public:

    Book();
    Book(const char * title, const int * isbn = BOOK_DEFAULT_ISBN,
          const char * renter = BOOK_DEFAULT_RENTER); //use default parameters in list
    Book(const Book& other);
    ~Book();

    Book & operator=(const Book & rhs);
    void setTitle(const char * title);      const char * getTitle() const;
    void setIsbn(const int * isbn);         const int * getIsbn() const;
    ...
}
```

Classes - Examples

Declare a Class, Implement some Methods

```
...  
bool getAvailable() const;  
[const char *] getRenter() const;  
  
bool valid() const;  
bool operator+(const char * renter);  
void free();  
  
static int getIdgen();  
  
private:  
  
const size_t m_id;  
  
char m_title[STR_MAX];  
int m_isbn[BOOK_ISBN_LEN];  
bool m_available;  
char m_renter[STR_MAX];  
  
static size_t s_idgen;  
  
};
```

➤ Getters/Setters/Helpers all should follow rules about allowing / restricting mutation if so specified !!!

➤ **static** member functions & data

➤ **const** data members !!!

Classes - Examples

Declare a Class, Implement some Methods

```
size_t Book::s_idgen = 0; //do not forget static member definition
```

```
Book::Book()  
: m_id( s_idgen++ ) //member is const, have to use initializer list to set it  
                  //in all constructors  
{  
    setTitle(BOOK_DEFAULT_TITLE); //code reuse - set the title to default  
    setIsbn(BOOK_DEFAULT_ISBN);  //code reuse - set the isbn to default  
    free();                       //code reuse - mark as free  
}
```

➤ Initializer Lists are only used with Constructors and appear only in implementations !

➤ Re-use functionalities that should be there according to specifications !

Classes - Examples

Declare a Class, Implement some Methods

```
Book::Book(const char * title, const int * isbn, const char * renter)
: m_id( s_idgen++ )    //member is const, have to use initializer list to set it
                      //in all constructors
{
    setTitle(title); //code reuse - set the title using passed parameter
    setIsbn(isbn);   //code reuse - set the isbn using passed parameter
    free();          //code reuse - mark as free first but then check whether the user put in
                      //an actual renter name or left everything to the default parameter list
    if ( strcmp(renter, BOOK_DEFAULT_RENTER) ){
        (*this) + renter; // note - cannot do: this->+(renter)
                          // the correct alternative is: this->operator+(renter)
    }
}
```

➤ Do NOT confuse with:

```
if (renter == BOOK_DEFAULT_RENTER) ...!
```

Classes - Examples

Declare a Class, Implement some Methods


```
Book::Book(const Book & other)
: m_id( s_idgen++ ) //member is const, have to use initializer list to set it
                  //in all constructors
{
    setTitle(other.m_title); //code reuse - set the title using other object data
    setIsbn(other.m_isbn);   //code reuse - set the isbn using other object data
    free(); //custom copy constructor will make book copy but mark new object as free
}

Book::~~Book()
{
    //--s_idgen; //not decrementing static to avoid duplication
}
```

Classes - Examples

Declare a Class, Implement some Methods

```
Book & Book::operator=(const Book & rhs){  
    if (this != &rhs){ //remember to check for self-assignment first  
        //cannot do anything to const int m_id  
        setTitle(rhs.m_title); //code reuse - set the title using rhs object data  
        setIsbn(rhs.m_isbn); //code reuse - set the isbn using rhs object data  
        if (rhs.m_available){  
            free();  
        }  
        else{  
            (*this) + rhs.m_renter; //code reuse - operator+ to add other object's renter  
        }  
    }  
    return *this;  
}
```



➤ Rules & Methodology expected to be followed *to-the-Letter* !!!

Classes - Examples

Declare a Class, Implement some Methods

```
void Book::setTitle(const char *title) {  
    strcpy(m_title, title);  
}  
  
const char *Book::getTitle() const {  
    return m_title;  
}  
  
void Book::setIsbn(const int *isbn) {  
    intcpy(m_isbn, isbn, BOOK_ISBN_LEN);  
}  
  
const int *Book::getIsbn() const {  
    return m_isbn;  
}  
  
bool Book::getAvailable() const {  
    return m_available;  
}  
  
const char *Book::getRenter() const {  
    return m_renter;  
}
```

➤ **const**—correctness
applies *at all times* !!!

Classes - Examples

Declare a Class, Implement some Methods

```
bool Book::operator+(const char * renter){ //const correctness
    if (!m_available ||
        !strcmp(renter, BOOK_DEFAULT_RENTER) //remember: strcmp match returns 0
    ){ //basic sanity check: is the book available?
        //are we giving a valid name for the renter?
        return false; //directly return false
    }
    else{
        strcpy(m_renter, renter);
        m_available = false; //remember: maintain the logic
    }
    return true;
}
```

➤ Do NOT confuse with:
m_renter = renter ...!

Classes - Examples

Declare a Class, Implement some Methods

```
void Book::free() {
    strcpy(m_renter, BOOK_DEFAULT_RENTER);
    m_available = true;           //remember: maintain the logic
}

bool Book::valid() const {       //use specified values that signify object state
    if (!strcmp(m_title, BOOK_DEFAULT_TITLE) ||
        !intcmp(m_isbn, BOOK_DEFAULT_ISBN, BOOK_ISBN_LEN)
    ) {
        return false;
    }
    return true;
}
```

➤ Do NOT confuse with:

```
if (m_isbn == BOOK_DEFAULT_ISBN) ...!
```

Classes - Examples

Declare a Class, Implement some Methods

```
int Book::getIdgen() { //static member function - but no static keyword in definition
    return s_idgen;
}
```

```
std::ostream & operator<<(std::ostream & os, const Book & b) { //not a member function
    os << b.m_title << " (" << b.m_id << ") ";
    int print(os, b.m_isbn, BOOK_ISBN_LEN);
    if (b.m_available){
        os << " Free for rent";
    }
    else{
        os << " Rented to: " << b.m_renter;
    }
    return os;
}
```

➤ Rules & Methodology expected to be followed *to-the-Letter* !!!

Classes - Examples

Work with an Aggregate Class

```
const size_t LIBRARY_N_BOOKS = 1000;

class Library {
    friend std::ostream& operator<<(std::ostream& os,
                                    const Library& l);

public:
    Library(const char * name);

    void setName(const char * name);
    const char * getName() const;

    Book * findOpenSpot();
    Book * operator[](const char * title);
    Book & operator[](size_t index);

    bool rentBook(size_t index, const char * name);
    bool operator+(const Book & book);

private:
    char m_name[STR_MAX];
    Book m_inventory[LIBRARY_N_BOOKS];
};
```

```
class Book {
    friend std::ostream & operator<<
        (std::ostream &, const Book & b);

public:
    Book();
    Book(const char *t, const int *isbn=...,
          const char * renter=...);
    Book(const Book & other);
    ~Book();
    Book& operator=(const Book & rhs);
    ... set/get...(c... ...);
    bool valid() const;
    bool operator+(const char * renter);
    void free();
    static int getIdgen();

private:
    const size_t m_id;
    char m_title[STR_MAX];
    int m_isbn[BOOK_ISBN_LEN];
    bool m_available;
    char m_renter[STR_MAX];
    static int s_idgen;
};
```

Classes - Examples

Work with an Aggregate Class

```
Library::Library(const char * name)
{
    //Book objects array m_inventory automatically instantiated - default ctor based
    setName(name);           //code reuse
}

void Library::setName(const char * name){
    strcpy(m_name, name);    //code reuse - cstring manipulation
}

const char * Library::getName() const{    //const correctness
    return m_name;
}
```

Classes - Examples

Work with an Aggregate Class

```
Book * Library::findOpenSpot() {  
    Book * m_inventory_pt = m_inventory;  
    for (size_t i=0; i<LIBRARY_N_BOOKS; ++i){  
        if ( !m_inventory_pt->valid() ){ //code reuse: if the object at that index  
                                        //is not valid, it can be considered  
                                        //as "open-to-assign"  
            return m_inventory_pt; //found one  
        }  
        ++m_inventory_pt;  
    }  
    return NULL; //found none  
}
```

- Index by-Member-Value (here internal state with interface `valid`) and return Pointer-to-Data !

Classes - Examples

Work with an Aggregate Class

```
Book * Library::operator[](const char * title){
    Book * m_inventory_pt = m_inventory;
    for (size_t i=0; i<LIBRARY_N_BOOKS; ++i){
        if ( !strcmp(m_inventory_pt->getTitle(), title) ){ //code reuse: if check for
                                                            //specific title

            return m_inventory_pt;
        }
        ++m_inventory_pt;
    }
    return NULL;
}

Book & Library::operator[](size_t index){
    return m_inventory[index];
}
```

➤ Index by-Array-Index
and return Reference-to-Data !

➤ Index by-Member-Value (here C-string data member **m_title**
through interface **getValid**) and return Pointer-to-Data !

Classes - Examples

Work with an Aggregate Class

```
bool Library::rentBook(size_t index, const char * name){
    return m_inventory[index] + name; //code reuse: class Book operator+
                                     //function returns bool on success/fail
}

bool Library::operator+(const Book & book){
    if ( book.valid() ){ //code reuse: first check that passed object is valid
        Book * open_book_pt = findOpenSpot(); //code reuse: then find an open spot
        if ( open_book_pt ){ //code reuse: check not NULL-pointer
                               //if findOpenSpot() succeeded
            *open_book_pt = book; //dereference and assign-to
            return true;
        }
    }
    return false;
}
```

Classes - Examples

Work with an Aggregate Class

```
std::ostream & operator<<(std::ostream & os, const Library & l){
    const Book * m_inventory_pt = l.m_inventory;
    for (size_t i=0; i<LIBRARY_N_BOOKS; ++i){
        if ( m_inventory_pt->valid() ){ //code reuse: check that output object is valid
            //call insertion operator on ostream os and pass Book object
            //have to dereference m_inventory_pt
            os << "Index: " << i << ", Book: " << *m_inventory_pt << endl;
        }
        ++m_inventory_pt;
    }

    // Alternative implementation: code reuse of operator[]
    // Compiler will optimize away extra function call - treat l[i] as direct indexing
    // for (size_t i=0; i<N_BOOKS; ++i)
    //     if ( l[i]->valid() )
    //         os << "Index: " << i << ", Book: " << l[i] << endl;

    return os; //remember: always return 1st argument for operator cascading
}
```

Classes - Examples

➤ Write a function that performs I/O and uses the previous

Usage

```
void importBooks(Library & library){ //object to update is passed by-Reference
    ifstream fin("LibraryIndex.txt");
    while (!fin.eof()){
        char title[STR_MAX]; fin >> title;
        char isbn_char[BOOK_ISBN_LEN]; fin >> isbn_char;
        const char * isbn_char_pt = isbn_char;
        int isbn[BOOK_ISBN_LEN];
        int * isbn_pt = isbn;
        for (int i=0, int* isbn_pt=isbn; i<BOOK_ISBN_LEN; ++i, ++isbn_pt, ++isbn_char_pt){
            *isbn_pt = *isbn_char_pt-'0'; //or use atoi
        }
        char renter[STR_MAX]; fin >> renter;
        if (fin.eof()){ break; }

        Book book(title, isbn, renter);

        library + book; //code reuse (Library's operator+ overload)
    }
    fin.close();
}
```

Classes - Examples

Usage

```
void importBooks(Library & library){ //object to update is passed by-Reference
[ifstream fin("LibraryIndex.txt");
while (!fin.eof()) {
    char title[STR_MAX]; fin >> title;
    char isbn_char[BOOK_ISBN_LEN]; fin >> isbn_char;
    const char * isbn_char_pt = isbn_char;
    int isbn[BOOK_ISBN_LEN];
    int * isbn_pt = isbn;
    for (int i=0, int* isbn_pt=isbn; i<BOOK_ISBN_LEN; ++i, ++isbn_pt, ++isbn_char_pt){
        *isbn_pt = *isbn_char_pt-'0'; //or use atoi
    }
    char renter[STR_MAX]; fin >> renter;
[if (fin.eof()){ break; }
    Book book(title, isbn, renter);

    library + book; //code reuse
                    //(Library's operator+ overload)
}
[fin.close();
}
```

➤ File I/O & Parsing

Classes - Examples

Usage

```
void importBooks(Library & library){  
    ifstream fin("LibraryIndex.txt");  
    while (!fin.eof()) {  
        char title[STR_MAX]; fin >> title;  
        char isbn_char[BOOK_ISBN_LEN]; fin >> isbn_char;  
        const char * isbn_char_pt = isbn_char;  
        int isbn[BOOK_ISBN_LEN];  
        int * isbn_pt = isbn;  
        for (int i=0, int* isbn_pt=isbn; i<BOOK_ISBN_LEN; ++i, ++isbn_pt, ++isbn_char_pt){  
            *isbn_pt = *isbn_char_pt-'0'; //or use atoi  
        }  
        char renter[STR_MAX]; fin >> renter;  
        if (fin.eof()){ break; }  
        Book book(title, isbn, renter);  
        library + book; //code reuse  
                        //(Library's operator+ overload)  
    }  
    fin.close();  
}
```

//object to update is passed by-Reference

➤ Re-use functionalities that should be there according to specifications !

Classes - Examples

➤ Write a function that performs I/O and uses the previous

Usage

```
void exportBooks(Library& library){ //parameter is passed by-Reference
    ofstream fout("LibraryIndexPost.txt");

    fout << library; //code reuse (operator<< overload for Library objects)
    fout.close();
}
```


Classes - Examples

Usage

```
void exportBooks(Library& library){ //parameter is passed by-Reference
    ofstream fout("LibraryIndexPost.txt");

    fout << library; //code reuse (operator<< overload for Library objects)
    fout.close();
}
```

➤ Write a function that performs I/O and uses the previous

➤ Re-use functionalities that should be there according to specifications !

Classes - Examples

Usage

- Trace output of a sample program
- Do it at the end, based on the code specifications you read !

```
int main(){
    Library delamare("DeLaMare Science and Engineering Library");

    importBooks(delamare);
    cout << delamare;

    int bookIndex;
    cout << endl << "What book index will you rent?" << endl;
    cin >> bookIndex;
    char renterName[STR_MAX];
    cout << "What is your name?" << endl;
    cin >> renterName;

    if ( !delamare.rentBook(bookIndex, renterName) ){
        cout << "Could not reserve book based on index, is it available?" << endl;
    }

    exportBooks(delamare);

    return 0;
}
```

Midterm Sample

Question 1

```
void printArray(int arr[], size_t size) {
    for (size_t i=0; i<size; ++i)
    {   cout << arr[i] << " "; }
    cout << endl;
}


void fillArrayAscending(int arr[], size_t size) {
    for (size_t i=0; i<size; ++i)
    { arr[i] = i; }
}

const size_t ARRAYSIZE = 10;
struct MyStruct{
    int intArray[ARRAYSIZE];
};

void fillStructArrayAscending(MyStruct st_in) {
    fillArrayAscending( st_in.intArray , ARRAYSIZE);
}

void printStructArray(MyStruct st_in) {
    printArray(st_in.intArray, ARRAYSIZE);
}
```

- Trace output
- Go line-by-line on the **main** !



```
int main() {
    MyStruct my_struct;
    printStructArray(my_struct);
    fillStructArrayAscending(my_struct);
    printStructArray(my_struct);
    return 0;
}
```

Midterm Sample

Question 1

```
void printArray(int arr[], size_t size) {
    for (size_t i=0; i<size; ++i)
    {   cout << arr[i] << " "; }
    cout << endl;
}

void fillArrayAscending(int arr[], size_t size) {
    for (size_t i=0; i<size; ++i)
    { arr[i] = i; }
}

const size_t ARRAYSIZE = 10;
struct MyStruct{
    int intArray[ARRAYSIZE];
};

void fillStructArrayAscending(MyStruct st_in) {
    fillArrayAscending( st_in.intArray , ARRAYSIZE );
}

void printStructArray(MyStruct st_in) {
    printArray(st_in.intArray, ARRAYSIZE);
}
```

Call-by-Value implementation of **fillStructArrayAscending** performs actions on local copy of **my_struct**. Both calls to **printStructArray** will print out the same (uninitialized values of **my_struct**).

```
int main() {
    MyStruct my_struct;
    printStructArray(my_struct);
    fillStructArrayAscending(my_struct);
    printStructArray(my_struct);
    return 0;
}
```

Midterm Sample

Question 2

- Check compilation correctness
- Check every function for known rules (e.g. access specifications, type correctness, overload resolution, etc.)

```
struct MyStruct{  
    void printIntVar(){  
        cout << intVar;  
    }  
    int intVar;  
};
```

```
int main(){  
    MyStruct ms;  
    ms.intVar = 1;  
    ms.printIntVar();  
    return 0;  
}
```

```
class MyClass{  
    public:  
        void setIntVar(int v){  
            m_intVar = v;  
        }  
        void printIntVar(){  
            cout << m_intVar;  
        }  
    private:  
        int m_intVar;  
};
```

```
int main(){  
    MyClass mc;  
    mc.setIntVar(1);  
    mc.printIntVar();  
    return 0;  
}
```

Midterm Sample

Question 2

```
struct MyStruct{  
    void printIntVar() {  
        cout << intVar;  
    }  
    int intVar;  
};
```

```
int main() {  
    MyStruct ms;  
    ms.intVar = 1;  
    ms.printIntVar();  
    return 0;  
}
```

```
class MyClass{  
    public:  
        void setIntVar(int v) {  
            m_intVar = v;  
        }  
        void printIntVar() {  
            cout << m_intVar;  
        }  
    private:  
        int m_intVar;  
};
```

```
int main() {  
    MyClass mc;  
    mc.setIntVar(1);  
    mc.printIntVar();  
    return 0;  
}
```

All clear.

We can have in a **struct** :

- a) Member functions
- b) Constructors
- c) Destructor

and **struct** members default to **public**.

Midterm Sample

Question 3

```
class TestClass{
    TestClass(){
        cout << m_intTest;
    }
    TestClass(int intTest){
        m_intTest = intTest;
        cout << m_intTest;
    }
private:
    int m_intTest;
};
```

- Check compilation correctness
- Check every function for known rules (e.g. **access specifications**, type correctness, overload resolution, etc.)
- Trace output

```
int main(){
    TestClass tc(1000);
    return 0;
}
```


Midterm Sample

Question 3

```
class TestClass{  
    TestClass(){  
        cout << m_intTest;  
    }  
    TestClass(int intTest){  
        m_intTest = intTest;  
        cout << m_intTest;  
    }  
private:  
    int m_intTest;  
};
```

No **public** access specifier for class Constructors.
Class members default to **private**.

```
int main(){  
    TestClass tc(1000);  
    return 0;  
}
```

Midterm Sample

- Trace output
- Go line-by-line on the **main** !

Question 4

```
class StaticClass{
public:
    static size_t count;
    StaticClass(){
        m_count = 0;
        count++;
    }
    StaticClass(int count_in){
        m_count = count_in;
        count++;
    }
    void countUp(){ m_count++; }
    int getCount(){ return m_count; }
private:
    int m_count;
};

int StaticClass::count = 0;
```

```
int main(){
    StaticClass sc_a;
    sc_a.countUp();
    StaticClass sc_b(sc_a.count);
    sc_b.countUp();
    StaticClass sc_c(sc_b);
    sc_c.countUp();

    cout << sc_a.getCount() <<" " <<
         sc_b.getCount() <<" " <<
         sc_c.getCount() <<" " <<
         StaticClass::count << endl;
    return 0;
}
```

Midterm Sample

Question 4

```
class StaticClass{
public:
    static size_t count;
    StaticClass(){
        m_count = 0;
        count++;
    }
    StaticClass(int count_in){
        m_count = count_in;
        count++;
    }
    void countUp(){ m_count++; }
    int getCount(){ return m_count; }
private:
    int m_count;
};

int StaticClass::count = 0;
```

No *Copy*-Constructor overload manipulating the **m_count** static, like the other Constructors do. Carefully mind the sequence of actions!

```
int main(){
    StaticClass sc_a;
    sc_a.countUp();
    StaticClass sc_b(sc_a.count);
    sc_b.countUp();
    StaticClass sc_c(sc_b);
    sc_c.countUp();

    cout << sc_a.getCount() <<" " <<
         sc_b.getCount() <<" " <<
         sc_c.getCount() <<" " <<
         StaticClass::count << endl;

    return 0;
}
```

Midterm Sample

Question 5

```
class BaseClass{
    public:
        void setIntVar(int i){ m_intVar = i; }
        int getIntVar(){ return m_intVar; }
    private:
        int m_intVar;
};

class DerivedClass : public BaseClass{
    public:
        void setDoubleVar(double d){
            m_doubleVar = d * m_intVar;
        }
        double getDoubleVar(){
            return m_doubleVar;
        }
    private:
        double m_doubleVar;
};
```

- Check compilation correctness
- Check every function for known rules (e.g. **access specifications**, type correctness, **overload resolution**, etc.)

```
int main(){
    BaseClass b_result;

    BaseClass b1;
    b1.setIntVar(10);
    DerivedClass d2;
    d2.setDoubleVar(2.5);

    b_result.setDoubleVar((double)b1.getIntVar() + d2.getDoubleVar());
    cout << b_result.getDoubleVar();

    return 0;
}
```

Midterm Sample

Question 5

```
class BaseClass{
public:
    void setIntVar(int i){ m_intVar = i; }
    int getIntVar(){ return m_intVar; }
private:
    int m_intVar;
};

class DerivedClass : public BaseClass{
public:
    void setDoubleVar(double d){
        m_doubleVar = d * m_intVar;
    }
    double getDoubleVar(){
        return m_doubleVar;
    }
private:
    double m_doubleVar;
};
```

- Derived-Class Methods called on Base-Class Object.
- Access of **private** (not **protected** in Derived Class)

```
int main(){
    BaseClass b_result;

    BaseClass b1;
    b1.setIntVar(10);
    DerivedClass d2;
    d2.setDoubleVar(2.5);

    b_result.setDoubleVar((double)b1.getIntVar() + d2.getDoubleVar());
    cout << b_result.getDoubleVar();

    return 0;
}
```

Midterm Sample

Question 6

```
class Parent{
    public:
        virtual void setValue(int value){ m_value = value; }
        virtual int getValue(){ return m_value; }
    protected:
        int m_value;
};

class Child : public Parent{
    public:
        virtual void setValue(int value){
            m_precisionValue = value;
        }
        virtual double getValue(){
            return m_precisionValue;
        }
    private:
        double m_precisionValue;
};
```

- Check compilation correctness
- Check every function for known rules (e.g. access specifications, **type correctness**, overload resolution, etc.)

```
int main(){
    Child c;

    c.setValue(1);
    cout << c.getValue()/2 << endl;

    return 0;
}
```

Midterm Sample

Question 6

```
class Parent{
public:
    virtual void setValue(int value){ m_value = value; }
    virtual int getValue(){ return m_value; }
protected:
    int m_value;
};

class Child : public Parent{
public:
    virtual void setValue(int value){
        m_precisionValue = value;
    }
    virtual double getValue(){
        return m_precisionValue;
    }
private:
    double m_precisionValue;
};
```

Overriding a **virtual** Method with a non-*Covariant* type **returning** function.

```
int main(){
    Child c;

    c.setValue(1);
    cout << c.getValue()/2 << endl;

    return 0;
}
```




CS-202

Time for Questions !