

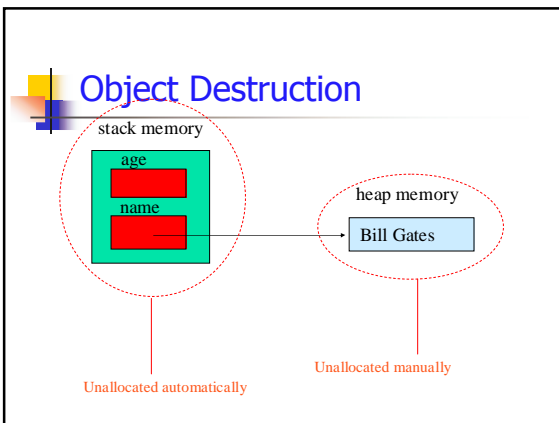
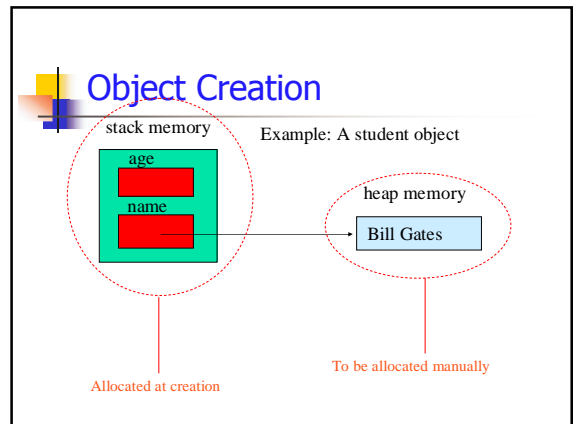
Object Oriented Programming

Constructors and destructors in C++

July 3, 2009 Dr. Mordo Shalom

- ### The need for a Constructor
- When an object is created, for instance `String s1;`
 - A block of memory is allocated for its non-static data members.
 - This area is un-initialized, therefore the variables contain garbage.
 - We can (sometimes) reset these values to meaningful values, by using appropriate methods – e.g. `s1.Assign("")`. This is:
 - not always possible
 - we might forget to do it (error prone)

- ### The need for a Destructor
- At the end of its lifetime the memory allocated for the object is discarded.
 - But other resources might still be allocated to it. For instance:
 - Heap memory that is pointed by its pointer data members.
 - Open files.
 - etc..
 - We have to free all these resource before the object "dies".



- ### The need for a Constructor
- When the object is created, for instance `Student s1;`
 - The values of the variables contain garbage
 - In particular, the pointers contain garbage.
 - No initialization of const vars.
 - No dynamic allocation
 - Unless we do all of this in the constructor.

Properties of c'tor

- Named by the name of the class.
- Almost always public.
- Can be overloaded.
- No return value (even not void).
- Only one invocation of constructor per object.
- A c'tor can not invoke other c'tor.

Example

point.h

```
class Point
{
public:
    Point(){} // default ctor
    Point(int a, int b); // ctor
    void init(){x=0;y=0;}
    int getX() const {return x;}
    int getY() const {return y;}
    void setX(int t) {if (t>=0 && t<=1024) x=t; else x=0;}
    void setY(int t) {if (t>=0 && t<=768) y=t; else y=0;}
    void setPoint(int t, int s);
    Point add(const Point& a)const;
    void print()const;
private:
    int x,y;
};
```

Example

point.cpp

```
Point::Point(int a, int b)
{
    if (a>=0 && a<=1024)
        x=a;
    else x=0;
    if (b>=0 && b<=768)
        y=b;
    else y=0;
}
```

Also possible

```
Point::Point(int a, int b)
{
    SetPoint(a,b);
}
```

C'tor invocation

- For an automatic variable (allocation in stack)
 - Point p1(1,-2);
- For a dynamic variable (allocation in heap)
 - Point *pp = new Point(1,-2);

Default Constructor

- Is a constructor with no parameters.
 - Point p1();
 - Point *pp = new Point();
- We can optionally omit the parentheses on invocation.
 - Point p1;
 - Point *pp = new Point;

Default Constructor-cont'd

- If no constructors are defined for a class, then a default c'tor **is supplied** by the compiler.
- If there is a non-default c'tor for a class but no default c'tor then a default c'tor **is not supplied** by the compiler.

Default Constructor-cont'd

- The default c'tor is used to create arrays.
- Non-default c'tors can not be used to initialize arrays.
- There is at most one default c'tor per class. Why ?

Implied default c'tor

```
class Time {
public:
    Time( int = 0, int = 0, int = 0); // default constructor
    void setTime( int, int, int ); // set hour, minute, second
    void printUniversal(); // print universal-time format
    void printStandard(); // print standard-time format
private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
}; // end class Time

// Time constructor initializes each data member to zero;
// ensures all Time objects start in a consistent state
Time::Time( int hr, int min, int sec )
{
    setTime( hr, min, sec ); // validate and set time
} // end Time constructor
```

Another example

```
class MyStack
{
public:
    MyStack(int size=DEFAULTSIZE); // default c'tor
    void Reset(){top=EMPTY;}
    void Push(char c);
    void Read(char *);
    char Pop(){return charArray[top--];}
    char TopOf() const {return charArray[top];}
    bool Empty() const {return (bool)(top == EMPTY);}
    bool Full() const {return (bool)(top == maxLen-1);}
    int GetLen() const {return maxLen;}
    int NumItems() const { return top+1;}
private:
    enum{EMPTY = -1};
    enum{DEFAULTSIZE = 10};
    char *charArray;
    int maxLen;
    int top;
};
```

Implementation

```
MyStack::MyStack(int size)
{
    charArray = new char[size];
    maxLen = size;
    top = EMPTY;
}

void MyStack::Read(char *t)
{
    while(*t!='\0' && !Full())
        Push(*t++);
}

void MyStack::Push(char c)
{
    if(!Full())
        charArray[++top] = c;
    else
        cout<<"Error!! Trying to push into a full stack\n";
}
```

The Singleton Pattern

- Goal: A class that has at most one instance.
- Howto:
 - A static pointer, pointing to the only instance.
 - The public method getInstance() returns this pointer. If it is NULL it creates a new instance.
 - The c'tor is private.

The Singleton Pattern

```
class Singleton
{
public:
    static Singleton* getInstance();
    //interface goes here
private:
    Singleton();
    static Singleton* instance;
};

Singleton *Singleton::_instance=NULL;
Singleton *Singleton::getInstance()
{
    if (!instance)
        instance = new Singleton;
    return _instance;
}

int main()
{
    Singleton *UniqueInstance = Singleton::getInstance();
    // ...
}
```

Member variable initializer

- There is a very basic difference between an assignment statement and a variable initialization.
 - `int i=0;` // initialization
 - `int j;`
 - `j=0;` // assignment
- Remember const variables.

Member variable initialization

- If an assignment statement in a c'tor is not considered as an initialization (as opposed to Java) it is an assignment statement.

```
Point::Point(int a, int b)
{
    if (a>=0 && a<=1024)
        x=a;
    else x=0;
    if (b>=0 && b<=768)
        y=b;
    else y=0;
}
```

Member variable initialization

- Sometimes we have to initialize variables:
 - consts
 - References
- A member variable can be an object by itself. Then we have to invoke its constructor.

Solution: constructor initializer

```
Point::Point(int a, int b)
:x(a),y(b)
{
    if (!(a>=0 && a<=1024)) x=0;
    if (!(b>=0 && b<=768)) y=0;
}
```

constructor initializer – example 2

```
class MyStack
{
public:
    MyStack(int size=DEFAULTSIZE); // default c'tor
    ...
private:
    enum {EMPTY = -1};
    enum {DEFAULTSIZE = 10};
    const int maxLen;
    char *s;
    int top;
};
```

```
MyStack::MyStack(int size)
: maxLen(size), top(EMPTY), s(new char[maxLen])
{ }
```

constructor initializer – example 3

```
class Student
{
public:
    Student():age(0){} // default c'tor
    Student(const char *theName, int theAge);
    Student(const String &theName, int theAge);
    // no need for d'tor
    void SetName(const String& newName);
    void SetName(const char* newName);
    String GetName() const { return name; }
    void SetAge(int newAge) { age = newAge; }
    int GetAge() const { return age; }
    void Print() const;
private:
    String name;
    int age;
};
```

String object

```

Student::Student(const char *theName, int theAge)
: name(theName), age(theAge)
{}

Student::Student(const String &theName, int theAge)
: name(theName), age(theAge)
{}

```

Invocation of the String c'tor

```

class Student
{
public:
    Student(): age(0) {} // default c'tor
    ...
private:
    String name;
    int age;
};

```

No invocation of string c'tor →
Default c'tor used.

More on c'tor initializers

- As a result of initializer the body of the c'tor may remain empty.
- The c'tor initializer is part of the implementation of the c'tor and not part of the prototype.
- The order the c'tor initializers are invoked, is not the order they are written. It is the order of the variables in the class. A good practice is to keep the same order to avoid confusion.

C'tor initializer – order of invocation

```

class MyStack
{
public:
    MyStack(int size=DEFAULTSIZE); // default c'tor
    ...
private:
    enum{EMPTY = -1};
    enum{DEFAULTSIZE = 10};
    char *s;
    const int maxLen;
    int top;
};

```

This will probably lead to a runtime error. Why ?

```

MyStack::MyStack(int size)
: maxLen(size), top(EMPTY), s(new char[maxLen])
{}

```

conversion c'tor

```

String::String(const char *p) // conversion c'tor
: s(new char[strlen(p)+1])
{
    cout<<"constructing string from char *"<<endl;
    strcpy(s,p);
}

```

Invoked automatically in type conversions:

- Type casting
- Parameter passing to functions

conversion c'tor

```

class String
{
public:
    String(): s(new char[1]) {s[0]=0;} // default c'tor
    String(const char *); // conversion c'tor

    void Assign(const String&);
    void Assign(const char *p);
    int Length() const;
    void Print() const;
private:
    char *s;
};

```

conversion c'tor

```
int main()
{
    char tempName[40];
    cout<<"Enter name: ";
    cin>>tempName;
    String tempString = tempName;
    tempString.Print();

    return 0;
}
```

Call to conversion ctor

conversion c'tor

```
class Student
{
public:
    Student():age(0){} // default c'tor
    Student(const String &theName, int theAge);

    void SetName(const String& newName);
    void SetName(const char* newName);
    String GetName() const { return name;}
    void SetAge (int newAge) {age = newAge;}
    int GetAge() const { return age;}
    void Print() const;

private:
    String name;
    int age;
};
```

String& parameter

conversion c'tor

```
void CheckName(const String& name)
{
    cout<<"checking name"<<endl;
}

int main()
{
    char tempName[40];
    int i, tempAge;
    cout<<"Enter student name: ";
    cin>>tempName;
    CheckName(tempName);
    cout<<"Enter age: ";
    cin>>tempAge;
    Student myStud(tempName,tempAge);
    myStud.Print();

    return 0;
}
```

String& parameter

char* argument
Conversion c'tor
invoked

Explicit invocation of conversion c'tor

This invocation is natural:

```
String my_string = "Shine on you crazy diamond";
```

This one is less natural:

```
MyStack theStack = 20;
```

*

Explicit – cont'd

```
void CheckStack(const MyStack& theStack)
{
    cout<<"Checking stack"<<endl;
}
```

```
MyStack theStack(20);
...
CheckStack(theStack);
```

This one is not natural at all:

```
CheckStack(20);
```

*

explicit

To prevent such usages we define the c'tor as explicit

```
explicit MyStack(int size=DEFAULTSIZE); //default c'tor
```

Then the following gets a compilation error

```
MyStack theStack = 20;
```

*

Copy c'tor

- A very basic operation that we do with basic types is to initialize a variable to the value of another variable of the same type.

```
int a = 5;
int b = a; // same type initialization
```

- The same can be done for complex types too:

```
Point p(1, -1);
Point q = p;           Point q(p);
```

The default copy constructor

- Invokes the copy constructors of each one of the data members.
- The default c'tor of the Point class is equivalent to:

```
Point::Point(const Point& rhs)
: x(rhs.x), y(rhs.y)
{ }
```

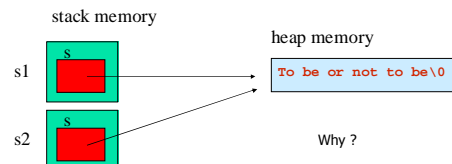
Invocation of the copy c'tor

- Done automatically when:
- An object is initialized to another object of the same type.
- A parameter is passed by value to a function.
- A value will be returned from a function

copy constructor

```
String s1("To be or not to be");
String s2 = s1; // call to copy ctor
```

The result is:



Copy c'tor-another example

```
int CountAlphaInStack(MyStack st) //copy c'tor is needed
{
    int count = 0;
    while(!st.Empty())
        if (isalpha(st.Pop()))
            count++;
    return count;
}
```

The parameter has to be passed by value. Why ?

Copy C'tor

- In both cases above the default copy c'tor is used.
- What it does is a "shallow copy" which is not what we need.
- Therefore we have to provide a copy constructor.

copy constructor

```
String::String(const String& st) // copy c'tor
: s(new char[st.Length()+1])
{
    strcpy(s,st.s);
}
```

```
MyStack::MyStack(const MyStack& st) //copy c'tor
:maxLen(st.maxLen), top(st.top),charArray(new char[maxLen])
{
    memcpy(charArray, st.charArray, maxLen);
}
```

Why not strcpy ?

copy constructor

In the following, we do not need to provide a copy c'tor for the Student Class. Why ?

```
1 Student a("Izzet",20);
2 Student b = a;
```

Destructors

- When an object is destroyed (either by going out of scope or by delete operation) two things happen.
 - The d'tor is invoked.
 - The memory is freed.
- If we do not provide a d'tor the compiler supplies a default d'tor.
- The default d'tor invokes the d'tors of the contained object.

destructors

```
int g()
{
    int n;
    Student st("Izzet", 20);
    ...
    return n;
}
```

call to d'tor of Student

destructors

```
Student *studentsList= new Student[10];
...
delete [] studentsList;
```

call to dtor of Student

Properties of D'tor

- In every class at most one d'tor
- Does not get parameters (even not void)
- Does not return a value (even not void)
- If there are contained objects they are destroyed **after** the containing object.
- We use d'tor mostly to
 - delete dynamically allocated objects
 - close files.

D'tor

- The d'tors functionality corresponds to the functionality of the c'tor s.
 - If c'tor allocated memory, d'tor de-allocates,
 - If c'tor opens file, d'tor closes it,
 - etc...

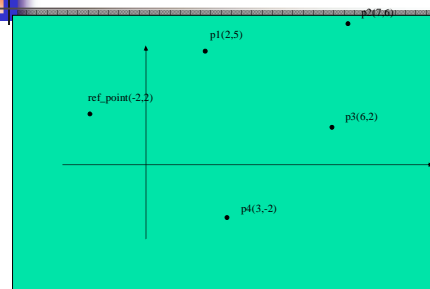
Example: String

```
class String
{
public:
    String(): s(new char[1]) {s[0]=0;} // default c'tor
    String(const char *); // conversion c'tor
    String(const String&); // copy c'tor
    ~String() {delete []s; } // d'tor
    void Assign(const String&);
    void Assign(const char *p);
    int Length() const;
    void Print() const;
private:
    char *s;
};
```

In this case we can delete [] s in the d'tor:

```
String(): s(NULL) {} // default c'tor
```

A case study: TaggedPoint



TaggedPoint

```
class TaggedPoint
{
public:
    TaggedPoint(); //default c'tor
    TaggedPoint(char *name); // default coordinates
    TaggedPoint(int a, int b); // default tag
    TaggedPoint(int a, int b, char *name);
    TaggedPoint(const TaggedPoint&); // copy c'tor
    ~TaggedPoint();
    void Init(){x=0;y=0;}
    int GetX() const {return x;}
    int GetY() const {return y;}
    void SetX(int t) {if (t>=0 && t<=1024) x=t; else x=0;}
    void SetY(int t) {if (t>=0 && t<=768) y=t; else y=0;}
    void SetTag(char *t);
    void SetPoint(int a,int b);
    void SetPoint(int a,int b, char *t);
    void Print()const;
    static int GetCount() {return count;}
private:
    static int defaultTag; // counts all points with default tag
    static int count; // counts all existing instances of TaggedPoint
    int x,y; // the coordinates
    char *tag; // the name of the point
    void CreateDefaultTag();
};
```

TaggedPoint

```
int TaggedPoint::defaultTag=0;
int TaggedPoint::count=0;
// default ctor
TaggedPoint::TaggedPoint()
: x(0), y(0)
{
    CreateDefaultTag();
    count++;
}
// ctor with tag
TaggedPoint::TaggedPoint(char *name)
: x(0), y(0), tag(NULL)
{
    SetTag(name);
    count++;
}
```

Could it be done otherwise ?

TaggedPoint

```
// ctor with coordinates
TaggedPoint::TaggedPoint(int a, int b)
{
    CreateDefaultTag();
    SetPoint(a,b);
    count++;
}
// ctor with coordinates and tag
TaggedPoint::TaggedPoint(int a, int b, char *name)
: tag(NULL)
{
    SetTag(name);
    SetPoint(a,b);
    count++;
}
```

TaggedPoint – copy c'tor

```
// copy c'tor copies the coordinates,
// but sets a new default tag
TaggedPoint::TaggedPoint(const TaggedPoint& p)
:x(p.x), y(p.y), tag(NULL)
{
    CreateDefaultTag();
    count++;
}
void TaggedPoint::CreateDefaultTag()
{
    tag=new char[5];
    *tag='p';
    defaultTag++;
    itoa(TaggedPoint.defaultTag,tag+1,10);
}
```

CreateDefaultTag

```
// creates a default tag for a point
// when the tag is not a ctor parameter
void TaggedPoint::CreateDefaultTag()
{
    tag=new char[5];
    *tag='p';
    defaultTag++;
    itoa(TaggedPoint.defaultTag,tag+1,10);
}
```

TaggedPoint – d'tor

```
// d'tor
TaggedPoint::~TaggedPoint()
{
    delete []tag;
    count--;
}
```

```
void TaggedPoint::SetTag(char *t)
{
    delete []tag;
    tag = new char[strlen(t) + 1];
    strcpy(tag,t);
}
void TaggedPoint::SetPoint(int t, int s)
{
    if (t>=0 && t<=1024) x=t; else x=0;
    if (s>=0 && s<=768) y=s; else y=0;
}
void TaggedPoint::SetPoint(int t, int s, char *newTag)
{
    SetPoint(t,s);
    SetTag(newTag);
}
void TaggedPoint::Print() const
{
    cout<<tag<<": ("<<x<<","<<y<<")\n";
}
```