

CS390-CPP

C++ programming

Gustavo Rodriguez-Rivera

General Information

- Web Page:
<http://www.cs.purdue.edu/homes/cs390lang/cpp>
- Office: LWSN1210
- E-mail: grr@cs.purdue.edu
- Textbook:
 - C++ Programming with Design Patterns Revealed
Tomasz Muldner
Adison Wesley
ISBN 0-201-72231-3
 - Buying the textbook is not required

Grading

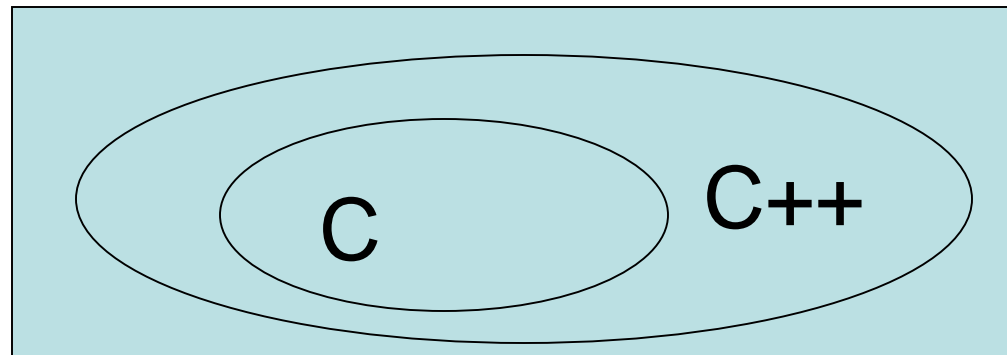
- Grade allocation
 - Final Exam: 40%
 - Final Project: 48%
 - Homework 2%
 - Quiz and Attendance 10%
- Exams also include questions about the projects.
- Post your questions in Piazza

Course Organization

- Basic Types
- Simple I/O
- Reference Data Types
- Function Overloading,
- Default Arguments Constructors and Destructors
- Constant and Static Features
- Exception Handling
- Namespaces
- Inheritance
- Overloaded Operators
- Templates
- Standard Template Library

C++ Introduction

- C++ was designed by Bjarne Stroustrup of AT&T Bell Labs in the early 1980s
- Now C++ is widely accepted as an object-oriented low-level computer language.
- C++ is a superset of C, that is, code written in C can be compiled by a C++ compiler.



C++ and Java

- James Gosling created the Java Language
- He used C++ as base to design the Java syntax
- In this way it was easy for C++ programmers to learn Java.
- You will find Java and C++ to be very similar

Example of a C++ Program: Stack

- A C++ class is divided into two files:
 - The Header file
 - It ends with .h, or .hh
 - Example: Stack.h
 - An implementation file
 - It ends with .cc or .cpp
 - Example: Stack.cpp
- Programs that use the C++ class should include the header file
 - #include “Stack.h”

Stack.h

```
// Make sure that this file is included only once
#ifndef Stack_h
#define Stack_h

// Definition of a Stack class to store double values

class Stack {
private:
    int maxStack;    // Max number of elements
    int top;        // Index to top of the stack
    double * stack; // Stack array
public:
    // Constructor
    Stack(int maxStack);

    // Push a value into the stack.
    // Return false if max is reached.
    bool push(double value);
};
```

Stack.h (cont)

```
// Pop a value from the stack.  
// Return false if stack is empty  
bool pop(double & value);  
  
// Return number of values in the stack  
int getTop() const;  
  
// Return max number of values in stack  
int getMaxStack() const;  
  
// Prints the stack contents  
void print() const;  
  
// Destructor  
~Stack();  
};  
  
#endif
```

Stack.cpp

```
// Stack Implementation

// Used for cout << "string"
#include <iostream>

using namespace std;

#include "Stack.h"

// Constructor
Stack::Stack(int maxStack) {
    this->maxStack = maxStack;
    stack = new double[maxStack];
    top = 0;
}
```

Stack.cpp (cont.)

```
// Push a value into the stack.  
// Return false if max is reached.  
bool  
Stack::push(double value) {  
    if (top == maxStack) {  
        return false;  
    }  
  
    stack[top]=value;  
    top++;  
    return true;  
}
```

Stack.cpp (cont.)

```
// Pop a value from the stack.  
// Return false if stack is empty  
bool  
Stack::pop(double & value) {  
    if (top == 0) {  
        return false;  
    }  
  
    top--;  
  
    // Copy top of stack to variable value  
    // passed by reference  
    value = stack[top];  
    return true;  
}
```

Stack.cpp (cont.)

```
// Return number of values in the stack
int
Stack::getTop() const {
    return top;
}
```

```
// Return max number of values in stack
int
Stack::getMaxStack() const {
    return maxStack;
}
```

Stack.cpp (cont.)

```
// Prints the stack contents
void
Stack::print() const {
    cout << "Stack:" << endl;
    if (top==0) {
        cout << "Empty" << endl;
    }
    for (int i = 0; i < top; i++) {
        cout << i << ":" << stack[i] << endl;
    }
    cout << "-----" << endl;
}
// Destructor
Stack::~Stack() {
    delete [] stack;
}
```

TestStack.cpp

```
// Example program to test Stack class
#include <iostream>
#include "Stack.h"

using namespace std;

int
main(int argc, char **argv) {
    Stack * stack = new Stack(10);
    stack->push(40);
    stack->push(50);
    stack->push(60);
    stack->push(70);
    stack->push(80);
}
```

TestStack.cpp (cont.)

```
stack->print();
```

```
double val;
```

```
stack->pop(val);
```

```
cout << "val=" << val << endl;
```

```
stack->print();
```

```
delete stack;
```

```
}
```

Makefile

```
all: TestStack
```

```
TestStack: TestStack.cpp Stack.cpp Stack.h  
    g++ -o TestStack TestStack.cpp Stack.cpp
```

```
clean:
```

```
    rm -f TestStack core
```

Output

```
bash-4.1$ make
g++ -o TestStack TestStack.cpp Stack.cpp
.bash-4.1$ ./TestStack
Stack:
0:40
1:50
2:60
3:70
4:80
-----
val=80
Stack:
0:40
1:50
2:60
3:70
-----
```

Reference Data Types

- It is used to create aliases of variables and to pass by references in functions.

```
int i = 1;
int & r = i; // Now r is an alias for i
r++;        // Increments i
```

- References have pointer semantics.

Passing by reference

- If a parameter type of a function is a reference and the parameter is modified, then it will modify the variable passed as parameter.

```
void swap(int & a, int & b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}
```

```
main () {  
    int x = 5;  
    int y = 6;  
    swap(x, y);  
    // Now x ==6 and y ==5  
}
```

- **Notice that no pointer operators * are not necessary so it looks simpler.**

Passing by Reference

- If we want a similar behavior using pure C we will need to write swap as follows:

```
void swap(int * pa, int * pb) {  
    int tmp = *pa;  
    *pa = *pb;  
    *pb = tmp;  
}
```

```
main () {  
    int x = 5;  
    int y = 6;  
    swap(&x, &y);  
    // Now x ==6 and y ==5  
}
```

- References is a “syntax sugar” for pointers, that is, the code generated is the same in both cases.

Constants

- You can define variables as constant.
- If you try to modify them in your code, the compiler will generate an error.
- In this way the compiler will detect misuses of variables.
- Early detection of errors is important.
- Example:

```
const double pi = 3.14;  
pi = 5; // Compiler Error!
```

Constant Parameters

- Also parameters to functions can be defined as constant.
- If the parameter is modified in the function then the compiler will generate an error.
- Example:

```
printInt(const int & i) {  
    i = 9; // Compiler Error!  
    . . .  
}
```

Default Parameters

- You can set default parameters.
- The default parameters should be the last ones in a function declaration.
- Example:

```
void drawRect(int x, int y,  
              int width=100, int height=100);
```

```
drawRect(45, 67); // Use defaults.
```

```
drawRect(78, 96, 45); // Use default height only.
```

- Only the declaration includes the default parameters and not the implementation.

Overloading Functions

- Also in C++ you can have multiple functions with the same name as long as the types of the arguments are different.
- When calling a function C++ will use the function that best matches the types.

Overloading Functions

- Example:

```
print(int a) { ... }
```

```
print(const char * s) { ... }
```

```
print(double d) { ... }
```

```
...
```

```
int x = 9;
```

```
double y = 89.78;
```

```
const char * z = "Hello world";
```

```
print(x); // Uses print(int a);
```

```
print(y); // Uses print(double d)
```

```
print(z); // Uses print(const char * s)
```

Operator Overloading

- You can also define your own operators in C++.
- For example, if you have a class

```
class A {  
    ...  
}
```

Then you can define an operator

```
A operator + (const A &s1, const A &s2) {  
    ...  
}
```

And use it as:

```
A a;  
A b;  
A c = a + b;
```

We will see more of this later.

Classes

- Remember that classes in C++ use two files.
- `<class>.h` (or `.hh`) has the class declaration with the names of the methods and variables.
- `<class>.cpp` (or `.cc` or `.cxx`) has the implementation.

private:/protected:/public:

- Variables inside a class can be defined private, protected or public.
- **private:**
 - It means that every method or variable after a ***private:*** declaration can be used only by the class that defines it.
- **protected:**
 - It means that every method or variable after a ***protected:*** declaration can be used only by the class or subclass that defines it.
- **public:**
 - It means that every method or variable after a ***public:*** declaration can be used by anybody.
- See Stack.h

friends

- In some cases you would like to allow some special classes to access the private method or variables of a class.
- For this you use friends.
- Example:

```
class ListNode {
    friend class List; // Class list can access
    char * val;        // anything in ListNode
    ListNode * next;
}
class List {
    ...
}
```

Inline Functions

- You can define functions as inline.
- This will be a hint for the compiler to “inline” or expand the function instead of calling it.
- This may be faster in some cases.
- Example:

```
inline int min(int a, int b) {  
    return (b<a)?b:a;  
}
```
- The inline function may appear in a header file.

Creating an Instance of an Object

- You can create an instance of an object dynamically by calling “new”. Example:
`Stack * stack = new Stack();`
- There is no garbage collector in C++, therefore you need to “delete” an object when it is no longer needed. Example:
`delete stack;`
- Not calling delete may cause your program to use more and more memory. This is called a “memory leak”.
- Be careful not to delete an object while it is still in use. This is called a “premature free”.

Objects as Local Variables

- You may also create an object as a local variable.
Example:

```
void pushMany() {  
    Stack stack;  
    stack.push(78.9);  
    stack.push(89.7);  
    stack.print();  
}
```

- The object will be deleted automatically when the function returns. No explicit call to “delete” is needed.
- The destructor will be called before returning.
- Try to define objects as local variables when possible.

TestStackWithLocalVars.cpp

```
// Example program to test Stack class
#include <iostream>
#include "Stack.h"

using namespace std;

int
main(int argc, char **argv) {
    Stack stack(10);
    stack.push(40);
    stack.push(50);
    stack.push(60);
    stack.push(70);
    stack.push(80);
```

TestStackWithLocalVars.cpp(cont.)

```
stack.print();
```

```
double val;
```

```
stack.pop(val);
```

```
cout << "val=" << val << endl;
```

```
stack.print();
```

```
}
```

Objects as Global Variables

- Alternatively, you can define an object as a global variable.
- The variable will be created and the constructor called before `main()` starts.
- These constructors called before `main` starts are called “static constructors”.

Objects as Global Variables

```
// Example program to test Stack class
#include <iostream>
#include "Stack.h"

using namespace std;

// Create stack before main starts
Stack stack(10);

int
main(int argc, char **argv) {
    stack.push(40);
    stack.push(50);
    stack.push(60);
    stack.push(70);
    stack.push(80);
}
```

Constructors and Destructors

- When an object is created, either with `new` or as a local variable, the constructor is called.
- The constructor is a method with the same name as the class of the object that contains initialization code.
- The destructor is a method in the class that starts with “~” and the name of the class that is called when the object is deleted.

Constructors and Destructors

Stack.h

```
class Stack {
private:
    int maxStack;    // Max number of elements
    int top;         // Index to top of the stack
    double * stack; // Stack array
public:
    // Constructor
    Stack(int maxStack);
    ~Stack();
...
}
```

Constructors and Destructors

`Stack.cpp:`

```
// Constructor
Stack::Stack(int maxStack) {
    this->maxStack = maxStack;
    stack = new double[maxStack];
    top = 0;
}

// Destructor
Stack::~~Stack() {
    // delete the array
    delete [] stack;
}
```

Constructors and Destructors

```
int foo() {  
    Stack * stack = new Stack(10);  
    ...  
    // Prevent memory leak  
    delete stack;  
}
```

Allocating Arrays Dynamically

- To allocate arrays dynamically you can use `new`.

```
double array = new double[100];
```

- To delete an array call “`delete []`”

```
delete [] array;
```

Copy Constructor

- Copy constructors are called when:

1. Initializing an object from another

```
Stack s1(50);
```

```
s1.push(78);
```

```
Stack s2 = s1;
```

```
// Now s1 and s2 are two
```

```
// different objects where s1 and
```

```
// s2 have the same contents.
```

Copy Constructor (cont)

2. A parameter of a class type is passed as a value

```
Stack s1(50);
```

```
s1.push(78);
```

```
foo(s1);
```

```
...
```

```
void foo(Stack s) {
```

```
    // s and s1 are two different
```

```
    // objects with the same contents.
```

Copy Constructor (cont)

3. When a value of a class type is returned in a function.

```
Stack s2 = getStack();
```

...

```
Stack getStack() {
```

```
    Stack s1(50);
```

```
    s1.push(78);
```

```
    return s1;
```

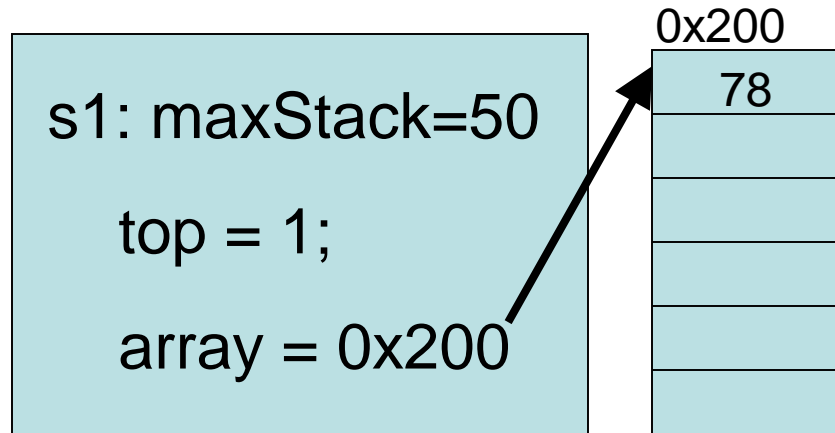
```
} // The content of s1 is copied into s2
```

Copy Constructor (cont.)

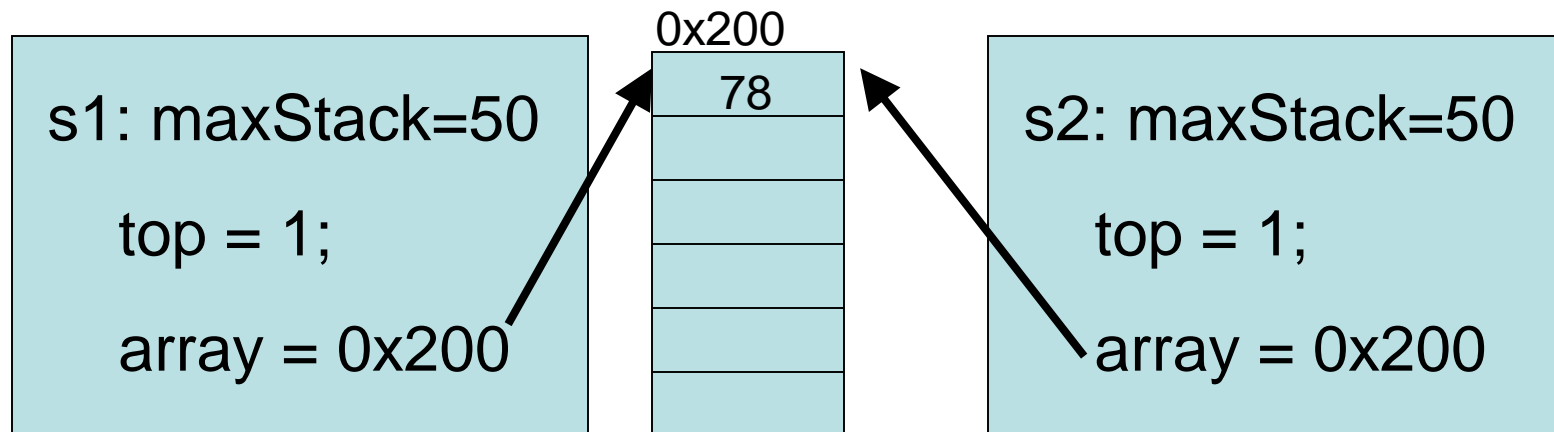
- By default the compiler will generate a copy constructor that will allocate a new instance and copy element member from the old to the new object (shallow copy).
- This is OK for simple classes but not for the classes that manage their own resources.
- Assume:

```
Stack s1(50);  
s1.push(78);  
Stack s2 = s1;
```

Copy Constructor (cont.)



Stack `s2 = s1;`



Copy Constructor (cont.)

- This is wrong because s2 should have its own copy of the stack array.
- This is solved by defining in the Stack class a “copy constructor” that will make a “deep copy” of the old object.

Copy Constructor (cont.)

Stack.h

```
class Stack {  
    ...  
    public:  
        Stack ();  
        Stack (Stack &s) ; // Copy constructor  
    ...  
};
```

Copy Constructor (cont.)

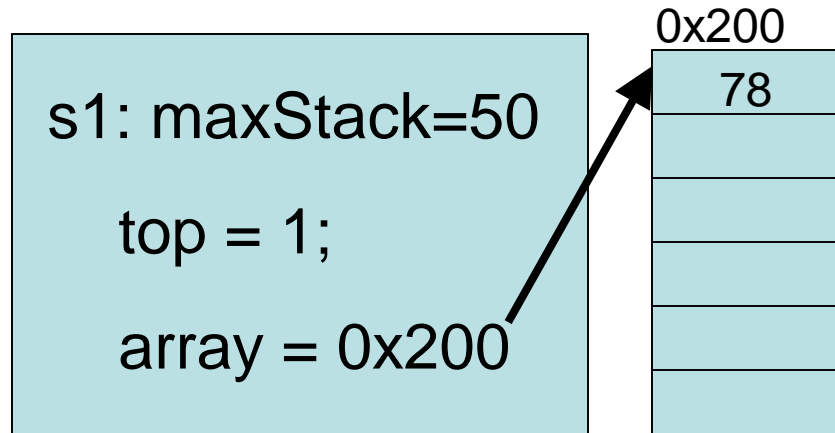
Stack.cpp

```
Stack::Stack(Stack &s) {
    top = s.top;
    maxStack = s.maxStack;

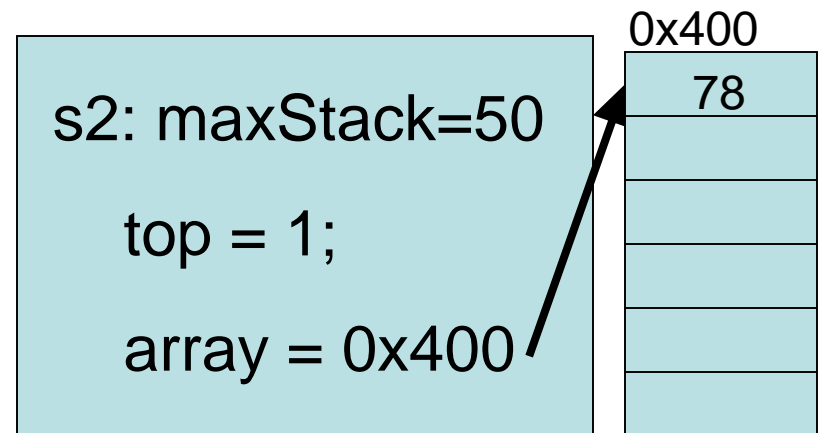
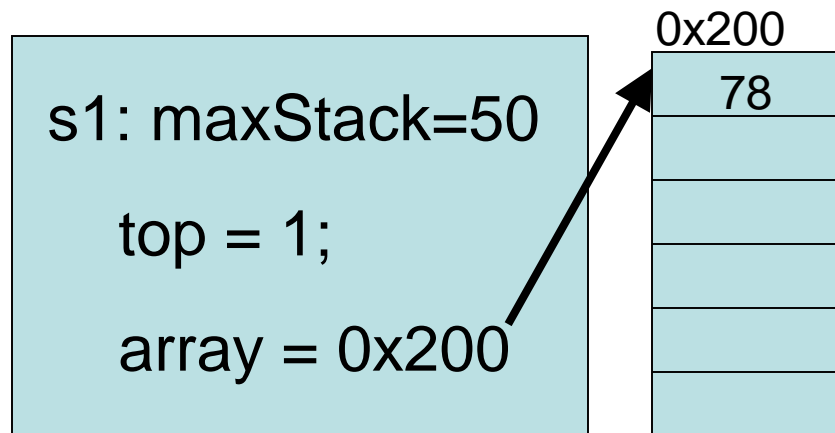
    // Create a new stack array
    stack = new double[maxStack];

    // Copy the entries in the array.
    for (int i = 0; i < top; i++) {
        stack[i] = s.stack[i];
    }
}
```

Copy Constructor (cont.)



Stack s2; s2 = s1;



Assignment Operator with Classes

- By default assigning an object to another will copy the elements of one object to another (shallow copy)

```
Stack a(20); // Create stack a as a local var
```

```
a.push(4);
```

```
a.push(7);
```

```
Stack b(10); // Create stack b as a local var
```

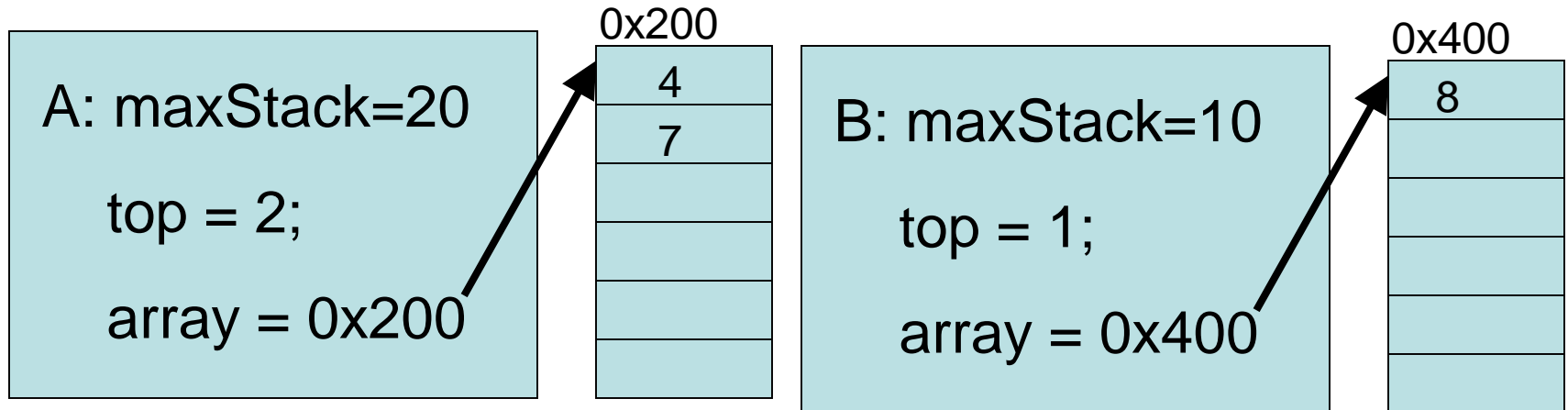
```
b.push(8);
```

```
b = a; // Assign content of a to b
```

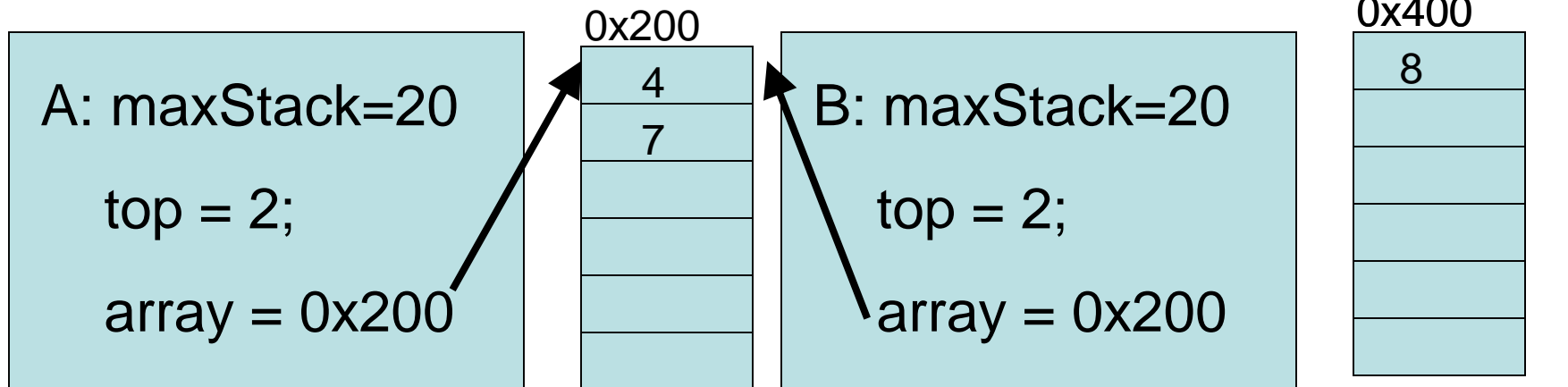
Assignment Operator with Classes

- This is wrong because the content of **b** will be exactly the same as **a**.
- The member variable “stack” will point to the same array in both a and b.
- The array pointed by **b** will be overwritten and leaked.
- To fix this problem if you need to assign objects to objects you have to define your own assignment operator.

Assignment Operator with Classes



b = a;



Assignment Operator with Classes

- The default assignment operator that implements a shallow copy is fine for most classes.
- However, it is not fine for classes that manage their own resources.
- For those classes we need to define an assignment operator.

Assignment Operator with Classes

Stack.h

```
class Stack {  
    ...  
    public:  
        Stack ();  
  
        // Copy constructor  
        Stack (Stack &s) ;  
  
        // Assignment Operator  
        Stack & operator=(const Stack & s) ;  
    ...  
};
```

Assignment Operator with Classes

Stack.cpp

```
// Assignment Operator
Stack &
Stack::operator=(const Stack & s) {
    // Check for self assignment. E.g. a=a;
    if (this == &s) {
        return *this;
    }
    // deallocate old array
    delete [] stack;
```

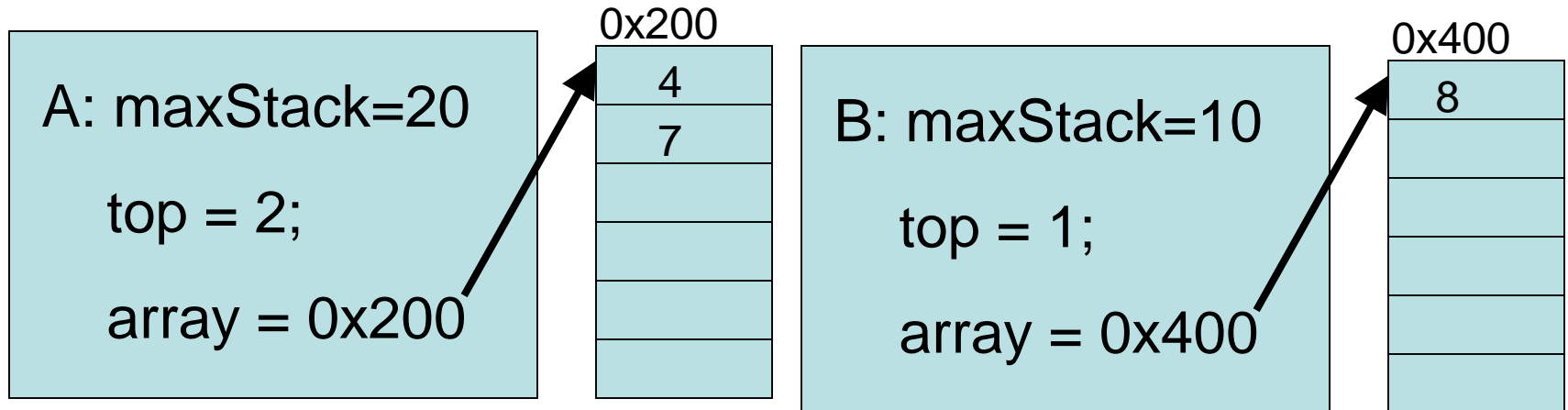
Assignment Operator with Classes

```
// Copy members
top = s.top;
maxStack = s.maxStack;

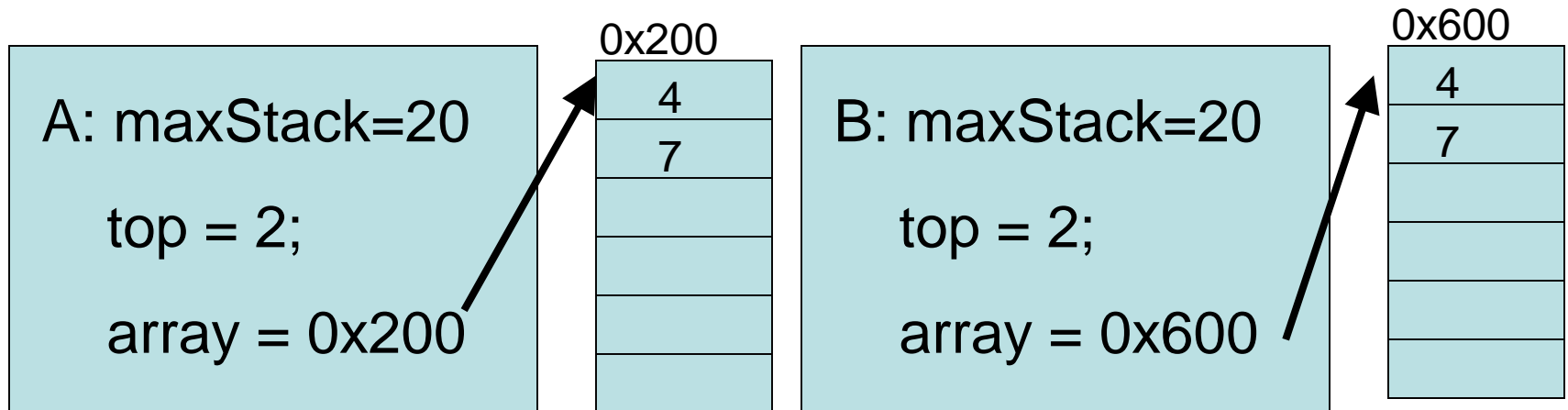
// Create a new stack array
stack = new double[maxStack];

// Copy the entries in the array.
for (int i = 0; i < top; i++) {
    stack[i] = s.stack[i];
}
return *this;
}
```

Assignment Operator with Classes



b = a;



Exception Handling

- In C++ there are exceptions like in Java
- The exceptions thrown derive from the class “exception”.
- Other standard exceptions are defined in the include `<stdexcept>`.
- Exception types are:
 - `exception`, `logic_error`, `invalid_argument`, `runtime_error`, `range_error`, `overflow_error`, `underflow_error`, etc.

Exception Handling

`Stack.h:`

```
#include <stdexcept>
class Stack {
    double pop()
    throw (logic_error);
    ...
}
```

Exception Handling

Stack.cpp:

```
#include "Stack.h"
double Stack::pop() throw(logic_error)
{
    if (top==0) {
        throw logic_error("Stack Empty");
    }
    top--;
    return stack[top];
}
```

Exception Handling

```
int main() {  
    Stack * stack = new Stack();  
    try {  
        stack->push();  
    }  
    catch (logic_error &e) {  
        cout << e.what() << endl;  
    }  
}
```

Namespaces

- Namespaces are used to help encapsulation and modularity.
- They also are used to avoid name conflicts.
- In Java namespaces are called packages.
- The syntax is similar to the classes but without the public/protected/private.

Namespace Definition

Figure.h

```
namespace MyDraw {  
    class Figure {  
        ....  
    };  
}
```

Line.h

```
namespace MyDraw {  
    class Line {  
        ....  
    };  
}
```

Namespace Usage

- To refer to the classes in the namespace from outside the namespace you need to pre-append the namespace.
- Example:

MyClass.h

```
#include "Figure.h"
```

```
class MyClass {
```

```
    MyDraw::Figure * figure; // Need to
```

```
                                //pre-append MyDraw
```

```
...
```

```
};
```


Standard Namespaces

- The standard library that defines strings, streams etc. are in the name space “std”.
- To refer to the elements of “std” you need to pre-append std::.

```
std::cout << "Hello world";
```

- If you don't want to pre-append “std” every time you use the standard library use the “using” statement.

```
using namespace std;
```

```
...
```

```
cout << "Hello world";
```

Public Inheritance

- It is used when you want a derived class to inherit all the public interface of the parent class.
- Assume the following Figure parent class.

Figure.h

```
class Figure {
public:
    enum FigureType { Line, Rectangle,
                     Circle, Text };

    Figure(FigureType figureType);

    FigureType getFigureType();
    void select(bool selected);
    bool isSelected();
    ...
};
```

Public Inheritance

- Now assume the following Line class that is subclass of Figure. Notice the keywords “public Figure”.

Line.h

```
class Line : public Figure {
    public:
        Line(int x0, int y0,
            int x1, int y1);
        int getX0();
        ...
}
```

- Line will inherit all the public methods of Figure, such as the select() and isSelected() methods.

Creating Objects of a Subclass

- The constructor of a subclass needs to take care first of constructing the attributes of the base class.
- In C++ there is no “super” like in Java, so the initialization of the base class is the same as the initialization of members.

```
// Constructor/destructor for a line  
Line::Line(int x0, int y0, int x1, int y1)  
:Figure(Figure::FigureType::Line)  
{  
    controlPoints.push_back(  
        new ControlPoint(this, x0, y0));  
    controlPoints.push_back(  
        new ControlPoint(this, x1, y1));  
}
```

Dynamic Cast

- You can assign a pointer to Line to a variable that is a pointer to Figure.

```
Figure * figure = new Line(x0, y0, x1, y1);
```

- You can use *figure* to call any public method of Figure in the instance of the Line object.

```
bool selected = figure->isSelected();
```

- However you cannot call a method that only belongs to Line.

```
int x0 = figure->getX0();  
    // Compiler error!! getX0() is  
    // not a method of Figure.
```

Dynamic Cast

- You can use a dynamic cast to go from a pointer to a base class to a pointer to a subclass.

```
Line * line =  
    dynamic_cast<Line>(figure);  
if (line != NULL) {  
    // Yeah! figure was a line.  
    int x0 = line->getX0();  
  
    ...  
}
```

- The dynamic cast will return NULL if the object *figure* points to is not a *Line*.

Virtual Methods

- A virtual method is a method that can be overwritten by a subclass.

Figure.h

```
class Figure {
    public:
        ...
        Figure(FigureType figureType);

        FigureType getFigureType();

        virtual void draw(CDC * pDC);
        ...
};
```

Virtual Methods

- The subclass may overwrite the virtual method or may keep the definition of the parent class.

Line.h

```
class Line : public Figure {
    public:
        Line(int x0, int y0,
            int x1, int y1);
        void draw(CDC * pDC);
        ...
};
```

- In this case Line overwrites the definition of the draw class.

Virtual Methods

- Other subclasses may also overwrite the draw method for other geometric shapes.

Rectangle.h

```
class Rectangle : public Figure {
public:
    Rectangle(int x0, int y0,
              int x1, int y1);
    void draw(CDC * pDC);
    ...
};
```

Calling Virtual Functions

- Another class like ***Drawing*** may have a vector of pointer to ***Figure*** to represent a collection of shapes.
- To draw all the figures in the vector it just needs to iterate over all the pointers and call draw().

```
void
Drawing::draw(CDC* pDC)
{
    // For each figure in vector "figures" draw the figure with
    // control points.
    for (unsigned i = 0; i < this->figures.size(); i++) {
        Figure * f = figures.at(i);
        f->draw(pDC);
    }
}
```

- The draw() method called will be the one redefined by the subclasses and not the one defined by Figure.

Abstract Classes

- An abstract class is one that can serve as a base class but that cannot be directly instantiated.
- It can be instantiated only through a subclass.
- An abstract class may declare methods without implementing them.

Abstract Classes

- Example

Figure.h

```
class Figure {
public:
    ...
    Figure(FigureType figureType);

    FigureType getFigureType();

    virtual void draw(CDC * pDC) = 0;
    ...
};
```

- The “=0” means that the base class Figure does not implement this method but subclasses should implement it.
- Trying to create an object of type Figure will give a compiler error.

```
Figure * f = new Figure(Figure::FigureType::Line);
// Compiler error
Figure * l = new Line(x0, y0, x1, y1); // OK
```

Virtual Destructors

- When calling delete on a subclass, the destructor of the subclass is called before the destructor of the base class.

A.h

```
class A {  
    X * x;  
public:  
    A();  
    ~A();  
};
```

A.cpp

```
A() {  
    x = new X;  
}  
~A() {  
    delete x;  
}
```

Virtual Destructors

B.h

```
class B : public A {  
    Y * y;  
public:  
    B();  
    ~B();  
};
```

B.cpp

```
B::B() {  
    y = new Y;  
}  
B::~~B() {  
    delete y;  
}
```

Virtual Destructors

```
B * b = new B();
```

```
...
```

```
delete b; // It will call ~B() and then ~A()
```

```
A * a = new A();
```

```
..
```

```
delete a; // It will call ~A()
```

However,

```
A* a = new B();
```

```
..
```

```
delete a; // It will call ~A() only!!!
```

Virtual Destructors

- To make sure that `~B` destructor is called, you need to define the destructor in `A` as virtual.

A.h

```
class A {  
    X * x;  
public:  
    A();  
    virtual ~A();  
};
```

- Now

```
A* a = new B();
```

```
..
```

```
delete a; // It will call ~B() and ~A() that is what we want.
```

Private Inheritance

- We have seen public inheritance where all the public methods of the parent class are public in the subclass.
- In private inheritance, the public methods of the parent class are private in the subclass.

```
class A {
    public:
        void xx();
        void yy();
};

class B : private A {
    public:
        using A::xx(); // Makes xx() public.
                       // Only xx() is accessible in B.
                       // yy() is private.
};
```

Protected Inheritance

- In protected inheritance, the public methods of the parent class are protected in the subclass.

```
class A {  
    public:  
        void xx();  
        void yy();  
};  
class B : protected A {  
    public:  
        using A::xx(); // Makes xx() made public.  
                        // Only xx() is public in B.  
                        // yy() is protected so it  
                        //can be inherited in a subclass.  
};
```

Multiple Inheritance

- In C++ you have multiple inheritance, that is a subclass can inherit from two or more parent classes.

```
class A {  
    public:  
        void xx();  
}  
class B {  
    public:  
        void yy();  
}  
class C: public A, public B {  
}; // C inherits from both A and B so xx() and yy() are  
public.
```

- Multiple inheritance is discouraged since adds extra complexity that is not needed.
- Java uses single inheritance but a class may implement multiple interfaces.

Parameterized Types

- In C++ we have three kind of types:
 - Concrete Type:
 - It is a user defined class that is tied to a unique implementation. Example: an int or a simple class.
 - Abstract Type:
 - It is user-defined class that is not tied to a particular implementation. Example Figure is an abstract class where draw can be Line::draw, Rectangle::draw(). It uses virtual methods and subclassing.
 - Parameterized type:
 - It is a type that takes as parameter another type. Example: Stack<int> creates a stack of type int, Stack<Figure *> will build a stack of entries of type Figure *. This is the base for “Templates”.

Templates

- They are parameterized types.
- They allow to implement data structures for different types using the same code, for example :
 - `Stack<int>` - Stack of type `int`
 - `Stack<double>`, Stack of type `double`
 - `Stack<Figure>`, Stack of type `Figure`.

Templates

- A generic class starts with the template definition:
 `template <typename T>`
- `typename T` indicates that `T` is a type parameter.
- There can be also compile time constants or functions
 `template <typename T, int SIZE>`

Writing a Template

- Before writing a template it is recommended to write the code of the class without the parameters using a concrete type.
- For example, if you want to write a List template for any type, write first a List class for “int”s (ListInt).
- Once that you compile, test and debug ListInt, then write the template by substituting the “int” by “Data” (the parameter type).
- Also add `template <typename Data>` before every class, function, and struct.

A ListInt Class

```
ListInt.h
```

```
// Each list entry stores int  
struct ListEntryInt {  
    int _data;  
    ListEntryInt * _next;  
};
```

A ListInt Class

```
class ListInt {  
    public:  
        ListEntryInt * _head;  
  
        ListInt();  
        void insert(int data);  
        bool remove(int &data);  
};
```

A ListInt Class

```
ListInt::ListInt()  
{  
    _head = NULL;  
}
```

A ListInt Class

```
void ListInt::insert(int data)
{
    ListEntryInt * e = new ListEntryInt;
    e->_data = data;
    e->_next = _head;
    _head = e;
}
```

A ListInt Class

```
bool ListInt::remove(int &data)
{
    if (_head==NULL) {
        return false;
    }

    ListEntryInt * e = _head;
    data = e->_data;
    _head = e->_next;
    delete e;
    return true;
}
```

A ListGeneric Template

- To implement the ListGeneric Template that can be used for any type we start with ListInt.
- Copy ListInt.h to ListGeneric.h.
- Add “template <typename Data> “ before any class, struct or function.
- Substitute “int” by ”Data”
- Where “ListEntryInt” is used, use “ListEntry<Data>” instead.
- Where “ListInt” is used, use “ListGeneric<Data>” instead.

A ListGeneric Template

ListGeneric.h

```
// Each list entry stores data
template <typename Data>
struct ListEntry {
    Data _data;
    ListEntry * _next;
};
```

A ListGeneric Template

```
template <typename Data>
class ListGeneric {
public:
    ListEntry<Data> * _head;

    ListGeneric();
    void insert(Data data);
    bool remove(Data &data);
};
```

A ListGeneric Template

```
template <typename Data>  
ListGeneric<Data>::ListGeneric()  
{  
    _head = NULL;  
}
```

A ListGeneric Template

```
template <typename Data>
void ListGeneric<Data>::insert(Data data)
{
    ListEntry<Data> * e = new ListEntry<Data>;
    e->_data = data;
    e->_next = _head;
    _head = e;
}
```

A ListGeneric Template

```
template <typename Data>
bool ListGeneric<Data>::remove(Data &data)
{
    if (_head==NULL) {
        return false;
    }

    ListEntry<Data> * e = _head;
    data = e->_data;
    _head = e->_next;
    delete e;
    return true;
}
```

Using the Template

- To use the template include “ListGeneric.h”
#include “ListGeneric .h”
- To instantiate the ListGeneric :

```
//List of int's  
ListGeneric<int> * listInt =  
    new ListGeneric<int>();
```

```
//List of strings  
ListGeneric<const char *> * listString =  
    new ListGeneric<const char *>();
```

Or as local/global vars

```
ListGeneric<int> listInt; // List of int's  
ListGeneric<const char *> listString; // list of strings
```

A test for GenericList

```
#include <stdio.h>
#include <assert.h>
#include "ListGeneric.h"

int
main(int argc, char **argv)
{
    //////////////////////////////////////
    // testing lists for ints

    ListGeneric<int> * listInt = new ListGeneric<int>();

    listInt->insert(8);
    listInt->insert(9);

    int val;
    bool e;
    e = listInt->remove(val);
    assert(e); assert(val==9);

    e = listInt->remove(val);
    assert(e);
    assert(val==8);
```

Using the Template

```
////////////////////////////////
```

```
// testing lists for strings
```

```
ListGeneric<const char *> * listString = new ListGeneric<const char *>();
```

```
listString->insert("hello");  
listString->insert("world");
```

```
const char * s;  
e = listString->remove(s);  
assert(e);  
assert(!strcmp(s, "world"));
```

```
e = listString->remove(s);  
assert(e);  
assert(!strcmp(s, "hello"));
```

```
}
```

Iterator Template

- An iterator is a class that allows us to iterate over a data structure.
- It keeps the state of the position of the current element in the iteration.

```
template <typename Data>
class ListGenericIterator {
    ListEntry<Data> *_currentEntry; // Points to the
    current node
    ListGeneric<Data> * _list;
public:
    ListGenericIterator(ListGeneric<Data> * list);
    bool next(Data & data);
};
```

Iterator Template

```
template <typename Data>
ListGenericIterator<Data>::ListGenericIterator(ListGeneric<Data> * list)
{
    _list = list;
    _currentEntry = _list->_head;
}
```

```
template <typename Data>
bool ListGenericIterator<Data>::next(Data & data)
{
    if (_currentEntry == NULL) {
        return false;
    }

    data = _currentEntry->_data;
    _currentEntry = _currentEntry->_next;

    return true;
}
```

Iterator Template

```
void testIterator() {
    ListGeneric<const char *> * listString = new ListGeneric<const char *>();
    const char * (array[]) = {"one", "two", "three", "four", "five", "six"};
    int n = sizeof(array)/sizeof(const char*);

    int i;
    for (i=0;i<n;i++) {
        listString->insert(array[i]);
    }

    const char * s;
    ListGenericIterator<const char *> iterator(listString);
    while (iterator.next(s)) {
        printf(">>%s\n", s);
        i--;
        assert(!strcmp(s, array[i]));
    }

    printf("Tests passed!\n");
}
```

Default Template Parameters

- You can provide default values to templates. Example:

Stack.h

```
template <typename T = int, int n = 20>
class Stack {
    T array[n];
    ...
};
```

- At instantiation time:

```
Stack stack1; // Stack of type int of size 20 (default)
Stack<double> stack2; // Stack of type double of size 20
Stack<Figure, 100> stack3; // Stack of type Figure of size 100
```

Function Templates

- Also functions can be parameterized.

```
template <typename T>
void swap(T &a, T &b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

...

```
int x = 3; int y = 4;
swap(x,y); // Swaps int vars x, y
double z1 = 3.567; double z2 = 56;
swap(z1, z2); // Swaps double vars z1, z2
```

- The compiler will generate instances of the swap function for double and int.

C++ Input/Output Library

- Three types of I/O
 - Interactive I/O
 - File I/O
 - Stream I/O
- Use `istream` for input and `ostream` for output.
- We have the following predefined streams:
`istream cin; // Standard input`
`ostream cout; // Standard output`

Output Operations

- To output a variable use the << operator.
cout << 7; // Prints a 7 in standard output.
- Also you can use modifiers like:
 - flush – Flush output buffer
 - endl - Sends end-of-line and the flush.
 - dec - Display ints in base 10
 - hex - Displays ints in hex format
 - oct - Displays ints in octal format.
 - setw(i) – Specify field width.
 - left - Specify left justification
 - right – Specify right justification
 - setprecision(i) – Set total number of digits in a real value.

Examples

```
cout << setw(6) << left << 45;
```

```
// Prints "45bbbb" where b is space char
```

```
cout << setw(6) << right << 45;
```

```
// Print "bbbb45"
```

```
cout << setprecision(3) << 45.68;
```

```
// Prints 46.7
```

Input Operations

- By default input operator >> skips whitespace chars.

```
int i;  
double f;  
cin >> i; // Read an int value i  
cin >> f; // Read double f
```

- To check error conditions

```
If (!cin) {  
    // operation failed  
}
```

- You can also read a whole line

```
char line[200];  
cin.getline(line, 200);
```

Input state

- To check the state of the input you can use:
 - `cin.eof()` – EOF
 - `cin.bad()` – In error state
 - `cin.fail()` – Return true if last operation failed.

File Streams

- To write to a file you use `<fstream>`
`ifstream` – input stream
`ofstream` – output stream

For example:

```
ifstream myinput("file.txt"); // Open file for reading.
int i;
myinput >> i; // Read integer i.
if (!myinput) {
    // Error
}
myinput.close(); // Close file.
```

File Streams

```
#include <fstream>

...

// Open file for writing.
ofstream myoutput("file.txt");
int i;
myoutput << i; // Write integer i.
if (! myoutput) {
    // Error
}
myoutput.close(); // Close file.
```

Standard Template Library (STL)

- Created by Alex Stepanov in 1997.
- Intended to provide standard tool for programmers for vectors, lists, sorting, strings, numerical algorithms etc.
- It is part of the ANSI/ISO C++ standard.
- It has three components:
 - Containers: Contain collections of values.
 - Algorithms: Perform operations on the containers.
 - Iterators: Iterate over the containers.

Containers

- Containers store elements of the same type.
- There exist two kind of containers: Sequential and associative.
 - Sequential Containers:
 - used to represent sequences

`vector<ElementType>` - vectors

`deque<ElementType>` - queues with operations at either end.

`list<ElementType>` - Lists

Containers

– Associative Containers:

- Used to represent sorting collections.
- They associate a key to a data value.

`set<KeyType>` - Sets with unique keys.

`map<KeyType, ElementType>` - Maps with unique keys.

`multiset<KeyType>` - Sets with duplicate keys.

`multimaps<KeyType, ElementType>` - Maps with duplicate keys.

Iterators

- Iterators are used to refer to a value in a container.
- Every container provides its own iterator.
Example:

```
vector<int> v;
```

```
. . .
```

```
for (vector<int>::iterator it = v.begin();  
     it!=v.end(); ++it) {  
    cout << *it << endl;  
}
```

Iterators

- An iterator supports at least the following operations:
 - *iter** refers to the value the iterator points to.
 - ++iter** increments iter to point to the next value in the container.
 - iter1 == iter2** Used to compare two iterators.
- Also a container like `vector<int> v`; provides the functions:
 - `v.begin()` – Returns an iterator that points to the beginning of the container.
 - `v.end()` – Returns an iterator that points to the end of the container.

Vectors

- Use

```
#include <vector>
```
- Represents a resizable array.

```
vector<int> v; // Vector of ints.  
vector<Figure *> figures; // Vector of pointer to Figures.
```
- ***v.capacity()*** represents the maximum number of elements before resizing.
- ***v.size()*** represents the number of elements used.
- ***v.push_back(e)*** adds an element to the end.
- ***e=v.pop_back()*** Remove last element.
- ***e=v[i]* or *e=v.at(i)*** Get a reference to the *i*th element

Vectors and Iterators

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;
main()
{
    vector<string> table;
    table.push_back("Hello");
    table.push_back("World");
    table.push_back("cs390cpp");

    // Iterating with i
    for (int i = 0; i < table.size(); i++) {
        cout << table[i] << endl;
    }
}
```

Vectors and Iterators

```
// Iterating with an iterator
vector<string>::iterator it;

for (it=table.begin(); it<table.end(); it++) {
    cout << *it << endl;
}

// Reverse iterator
vector<string>::reverse_iterator rit;
for (rit=table.rbegin(); rit<table.rend(); rit++)
{
    cout << *rit << endl;
}
}
```

Lists

- Use

```
#include <list>
```
- Represents a list.

```
list<int> l; // List of ints.  
list<Figure *> figures; // List of pointer to Figures.
```
- Optimized for insertions and deletions. No random access operators such as [] or at() are provided.
- ***l.size()*** represents the number of elements used.
- ***l.push_back(e)*** adds an element to the end.
- ***e=v.pop_back()*** Remove last element.
- ***l.sort()*** sorts list.

Maps

- They are templates that represent a table that relate a key to its data.
- `Std::map <key_type, data_type, [comparison_function]>`
- Example:

```
map <string, int> grades;  
grades["Peter"] = 99;  
grades["Mary"] = 100;  
grades["John"] = 98;  
cout<<"Mary's grade is "<<grades["Mary"] << endl;  
grades["Peter"] = 100; // Change grade
```

Maps

- Checking if an element is not in the map

```
if(grades.find("Luke") == grades.end()) {  
    cout<<"Luke is not in the grades list." << endl;  
}  
else {  
    cout<<"Luke's grade is "<< grades["Luke"] << endl;  
}
```

- Iterating over a map

```
for (map<string,int>::iterator it = grades.begin();  
    it != grades.end(); ++it) {  
    cout << "Name: " << it.first;  
    cout << "Grade: " << it.second << endl;  
}
```

- Erasing a key

```
grades.erase("Peter");
```

STL Strings

- The STL strings provide Java like string manipulation

```
#include<string>
```

```
...
```

```
string str1 = "Hello";
```

```
string str2 = "world."
```

```
string str3 = str1 + " " + str2;
```

```
cout << str3 << endl;
```

- You do not need to allocate or deallocate memory. Everything is done by the methods themselves.
- You can create a STL string from a C string:

```
string s("c string");
```

Some STL Strings Functions

- `str.length()`
 - Length of string
- `str[i]` or `str.at(i)`
 - Get *i*th character in the string.
- `size_t str1.find (const string& str2, size_t pos = 0)`
 - Find the position of a `str2` in `str1`. -1 if not found.
- `str.substr (size_t pos = 0, size_t n = npos)`
 - Returns a substring starting at `pos` with up to `n` chars.
- You can use `str.c_str()` to get the corresponding `const char *` string.

Memory Allocation Errors

- Explicit Memory Allocation (calling free) uses less memory and is faster than Implicit Memory Allocation (GC)
- However, Explicit Memory Allocation is Error Prone
 1. Memory Leaks
 2. Premature Free
 3. Double Free
 4. Wild Frees
 5. Memory Smashing

Memory Leaks

- Memory leaks are objects in memory that are no longer in use by the program but that ***are not freed.***
- This causes the application to use excessive amount of heap until it runs out of physical memory and the application starts to swap slowing down the system.
- If the problem continues, the system may run out of swap space.
- Often server programs (24/7) need to be “rebounced” (shutdown and restarted) because they become so slow due to memory leaks.

Memory Leaks

- Memory leaks is a problem for long lived applications (24/7).
- Short lived applications may suffer memory leaks but that is not a problem since memory is freed when the program goes away.
- Memory leaks is a “slow but persistent disease”. There are other more serious problems in memory allocation like premature frees.

Memory Leaks

Example:

```
int * i;  
while (1) {  
    ptr = new int;  
}
```

Premature Frees

- A premature free is caused when an object that is still in use by the program is freed.
- The freed object is added to the free list modifying the next/previous pointer.
- If the object is modified, the next and previous pointers may be overwritten, causing further calls to malloc/free to crash.
- Premature frees are difficult to debug because the crash may happen far away from the source of the error.

Premature Frees

Example:

```
int * p = new int;
* p = 8;
delete p; // delete adds object to free list
          // updating header info
...
*p = 9; // next ptr will be modified.
int *q = new int;
// this call or other future malloc/free
// calls will crash because the free
// list is corrupted.
// It is a good practice to set p = NULL
```

Double Free

- Double free is caused by freeing an object that is already free.
- This can cause the object to be added to the free list twice corrupting the free list.
- After a double free, future calls to malloc/free may crash.

Double Free

Example:

```
int * p = new int;

delete p; // delete adds object to free
list
delete p; // deleting the object again
          // overwrites the next/prev ptr
          // corrupting the free list
          // future calls to free/malloc
          // will crash
```

Wild Frees

- Wild frees happen when a program attempts to free a pointer in memory that was not returned by malloc.
- Since the memory was not returned by malloc, it does not have a header.
- When attempting to free this non-heap object, the free may crash.
- Also if it succeeds, the free list will be corrupted so future malloc/free calls may crash.

Wild Frees

- Also memory allocated with *malloc()* should only be deallocated with *free()* and memory allocated with *new* should only be deallocated with *delete*.
- Wild frees are also called “free of non-heap objects”.

Wild Frees

Example:

```
int q;
```

```
int * p = &q;
```

```
delete p;
```

```
// p points to an object without
```

```
// header. Free will crash or
```

```
// it will corrupt the free list.
```

Wild Frees

Example:

```
char * p = new char[100];  
p=p+10;
```

```
delete [] p;
```

```
// p points to an object without
```

```
// header. Free will crash or
```

```
// it will corrupt the free list.
```

Memory Smashing

- Memory Smashing happens when less memory is allocated than the memory that will be used.
- This causes overwriting the header of the object that immediately follows, corrupting the free list.
- Subsequent calls to malloc/free may crash
- Sometimes the smashing happens in the unused portion of the object causing no damage.

Memory Smashing

Example:

```
char * s = new char[8];  
strcpy(s, "hello world");  
  
// We are allocating less memory for  
// the string than the memory being  
// used. Strcpy will overwrite the  
// header and maybe next/prev of the  
// object that comes after s causing  
// future calls to malloc/free to crash.  
// Special care should be taken to also  
// allocate space for the null character  
// at the end of strings.
```

Errors

- Memory allocation errors are difficult to debug since the effect may happen farther away from the cause.
- Memory leaks is the least important of the problems since the effect take longer to show up.
- As a first step to debug premature free, double frees, wild frees, you may comment out free calls and see if the problem goes away.
- If the problem goes away, you may uncomment the free calls one by one until the bug shows up again and you find the offending free.

Debugging Memory Allocation Errors

- There are tools that help you detect memory allocation errors.
 - Valgrind
 - DrMemory
 - IBM Rational Purify
 - Bounds Checker
 - Insure++

These tools add overhead to the execution so you have to run them in a testing environment and not in production.

Garbage Collection

- Garbage collection is a subsystem that deletes objects once they are not reachable by the program.
- There is no garbage collection native in C++.
- However, there exist libraries such as Boehm's Conservative GC that can be used in C++:
http://www.hpl.hp.com/personal/Hans_Boehm/gc/
- Even with this library you have to follow certain rules, and there is no guarantee that a 3rd party library will follow these rules.
- A more realistic approach for GC in C++ is to use reference counting using “Smart Pointers”.

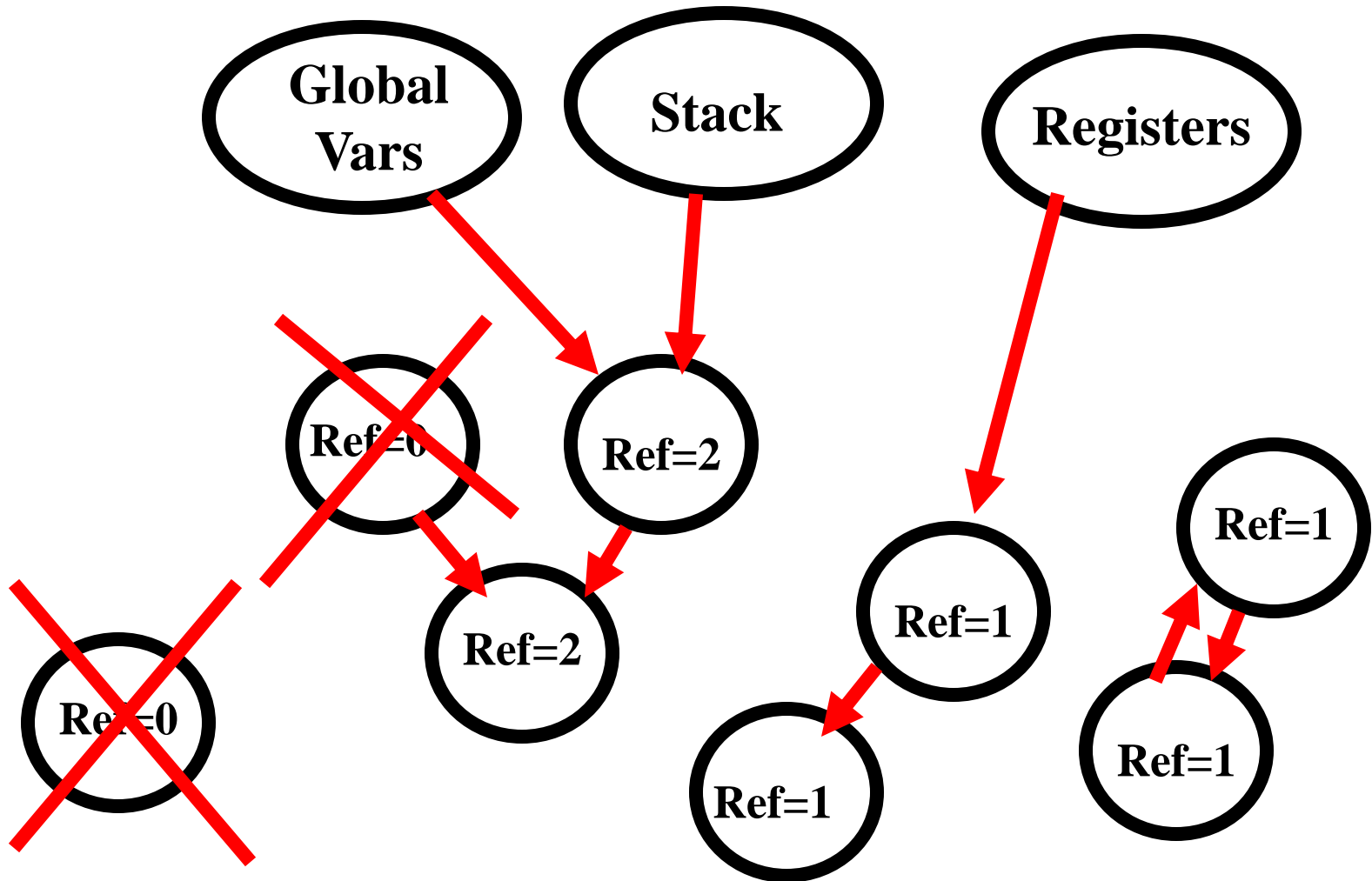
Reference Counting

- Reference counting is a garbage collection technique where an object has a reference counter that keeps track of the number of references to the object.
- Every time a new pointer points to the object, the reference counter is increased.
- When a pointer no longer points to the object, the reference is decreased.
- When the reference counter reaches 0, the object is removed.

Reference Counting

- In reference counting every object has a counter with the number of reference pointing to the object,
- When the reference count reaches 0 the object is removed.
- This is the approach used by languages such as Perl and Python or C++ with “Smart Pointers”.
- Advantage:
 - Easy to implement. Incremental (objects are freed while program is running)
- Disadvantage:
 - Slow. Every pointer assignment needs to increase/decreased a reference count.
 - Unable to free cycles
- In Multithreaded environments a mutex lock is necessary when incrementing/decrementing reference counter.

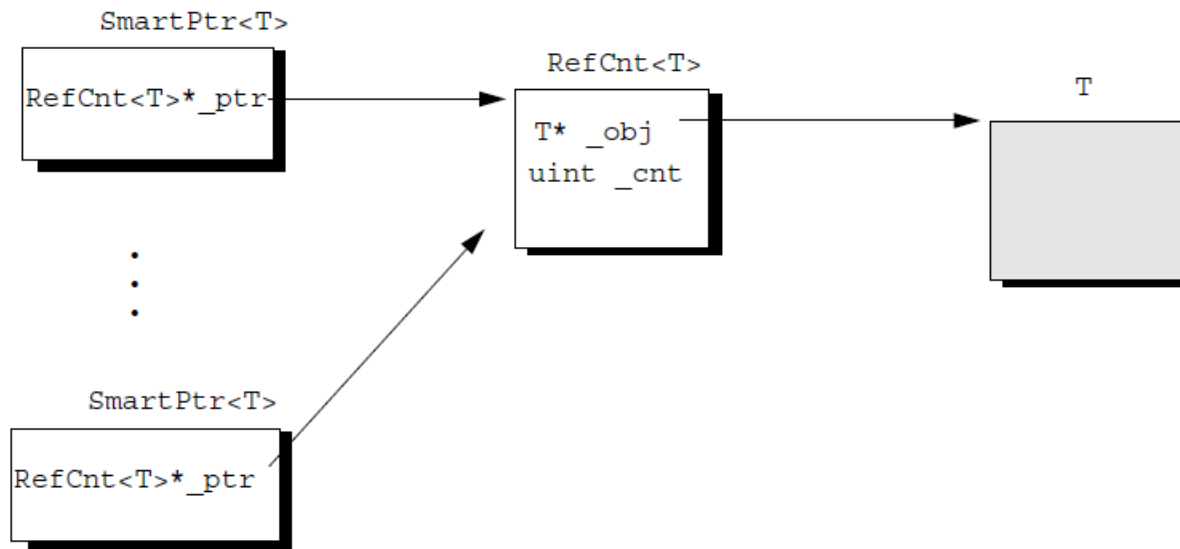
Reference Counting



Smart Pointers

- Based on Dr. Lavender Class notes at <http://www.cs.utexas.edu/~lavender/courses/cs371/lectures/lecture-15.pdf>
- Using operator overloading in C++ we can wrap the pointer operations such as “*” and “&” as well as “=” to increment/decrement a reference counter.
- Also we can wrap an object around a Reference Count wrapper.
- Multiple SmartPtr<T> template instances point to a single RefCnt<T> instance, which holds a pointer and a reference count to an instance of a class T (e.g., string).
- The SmartPtr<T> template is a *handle* that is used to access a reference counted object and which updates the reference count.
 - On construction of a SmartPtr<T> instance, the count is incremented
 - On destruction the count is decremented.
 - When the last SmartPtr<T> object is destructed, the RefCnt<T> is deleted, which in turn deletes the object of type T.

Smart Pointers



SmartPtr.h

Smart Pointers

```
#include <iostream>
using namespace std;
template<class T>
class RefCnt {
private:
    T* _obj;
    unsigned long _cnt;
public:
    RefCnt(T* p) : _obj(p), _cnt(0) {
        cout << "New object " << (void *) _obj << endl;
    }
    ~RefCnt() {
        cout << "Delete object " << (void *) _obj << endl;
        assert(_cnt == 0); delete _obj;
    }
    T* object() const { return _obj; }
    int inc() {
        _cnt++;
        cout << "increment:" << _obj << " _cnt=" << _cnt << "\n";
        return _cnt;
    }
    int dec() {
        _cnt--;
        cout << "decrement:" << _obj << " _cnt=" << _cnt << "\n";
        return _cnt;
    }
};
```

Smart Pointers

```
/* A ``smart pointer to a reference counted object */
template<class T> class SmartPtr {
    RefCnt<T>* _ptr; // pointer to a reference counted object of type T

    /* hide new/delete to disallow allocating a SmartPtr<T> from the heap */
    static void* operator new(size_t) {}
    static void operator delete(void*) {}

public:
    SmartPtr(T* p) {
        _ptr = new RefCnt<T>(p);
        _ptr->inc();
    }
    SmartPtr(const SmartPtr<T>& p) {
        _ptr = p._ptr;
        _ptr->inc();
    }

    ~SmartPtr() { if (_ptr->dec() == 0) { delete _ptr; } }

    void operator=(const SmartPtr<T>& p) {
        if (this != &p) {
            if (_ptr->dec() == 0) delete _ptr;
            _ptr = p._ptr;
            _ptr->inc();
        }
    }

    T* operator->() const { return _ptr->object(); }
    T& operator*() const { return *(_ptr->object()); }
};
```

Smart Pointers

testSmartPtr.cpp:

```
#include <string>
#include "SmartPtr.h"

typedef SmartPtr<string> StringPtr;

int main()
{
    cout << "StringPtr p = new string(\"Hello, world\")" << endl;
    StringPtr p = new string("Hello, world");

    cout << "StringPtr s = p" << endl;
    StringPtr s = p; // invokes copy constructor incrementing reference counter

    cout << "Invoke a String method on a StringPtr " << endl;
    cout << "Length of " << *s << " is " << s->length() << endl;

    cout << "s = new string(\"s\");" << endl;
    s = new string("s");

    cout << "p = new string(\"p\");" << endl;
    p = new string("p");

    cout << "Before Return"<< endl;

    return 0;
};
```

Smart Pointers

Makefile:

```
goal: TestSmartPtr
```

```
TestSmartPtr: TestSmartPtr.cpp SmartPtr.h  
    g++ -o TestSmartPtr TestSmartPtr.cpp
```

clean:

```
    rm -f TestSmartPtr core
```

Smart Pointers

Output:

```
lore 228 $ ./TestSmartPtr
StringPtr p = new string("Hello, world")
New object 0x22878
increment:0x22878 _cnt=1
StringPtr s = p
increment:0x22878 _cnt=2
Invoke a String method on a StringPtr
Length of Hello, world is 12
s = new string("s");
New object 0x22898
increment:0x22898 _cnt=1
decrement:0x22878 _cnt=1
increment:0x22898 _cnt=2
decrement:0x22898 _cnt=1
p = new string("p");
New object 0x228b8
increment:0x228b8 _cnt=1
decrement:0x22878 _cnt=0
Delete object 0x22878
increment:0x228b8 _cnt=2
decrement:0x228b8 _cnt=1
Before Return
decrement:0x22898 _cnt=0
Delete object 0x22898
decrement:0x228b8 _cnt=0
Delete object 0x228b8
```

Lock Guards

- Lock Guards is a wrapper that automatically locks a mutex lock when entering a function and unlock the mutex when exiting the function.

- Assume the following code:

```
pthread_mutex_t mutex;
void mt_function() {
    mutex_lock( &mutex);
    // Synchronized code

    ...
    if (error) {
        return; // Oops. Forgot to unlock mutex
    }
    mutex_unlock(&mutex);
}
```

Lock Guards

- Mutex locks may be prone to errors. The user may forget to unlock the mutex before returning, or an exception may be thrown while holding the mutex.
- Lock Guards are objects that wrap a mutex lock to lock it during construction at the beginning of the method and unlock it during destruction at the end of the method.

Lock Guards

```
class LockGuard {
    pthread_mutex_t & mutex;
public:
    LockGuard( pthread_mutex_t & mutex) {
        this->mutex = mutex;
        pthread_mutex_lock(&mutex);
    }
    ~LockGuard() {
        pthread_mutex_unlock(&mutex);
    }
};
```

Lock Guards

```
void mt_function() {  
    LockGuard myLock( mutex)  
    // Synchronized code  
  
    ...  
    if (error) {  
        return;  
        // Oops. Forgot to unlock mutex  
        // No problem!  
        // Destructor of LockGuard will unlock it at return.  
    }  
    // No need to call pthread_mutex_unlock() at return.  
    // Destructor of LockGuard will unlock it at return.  
}
```

Final Review

- Java and C++
- Example of a C++ program. Stack.h and Stack.cpp
- Reference Data Types
- Passing by Reference and by Value
- Constant Parameters
- Default Parameters.
- Function Overloading
- Operator Overloading

Final Review

- Classes
- private:, protected: public:
- friends
- Inline functions
- new and delete
- Objects as local variables
- Constructors and Destructors
- Copy constructor
- Assignment operator in classes
- Exception Handling

Final Review

- Namespaces “using namespace std”
- Standard namespaces. “using namespace std;”.
- Public inheritance
- Constructor in a subclass
- Dynamic cast
- Virtual Methods
- Abstract classes

Final Review

- Virtual Destructors
- Private and Protected Inheritance
- Multiple Inheritance
- Parameterized Types and Templates
- Writing a Template: A ListInt class and a ListGeneric template.
- Using a Template
- Iterator Templates
- Default Template Parameters
- Function Templates

Final Review

- C++ Input/Output Library
- File Streams
- Standard Template Library (STL)
- Containers and Iterators
- Vectors
- Lists
- Maps
- STL Strings
- Memory Allocation Errors
- Smart Pointers
- Lock Guards

To Study

- Class slides
- Study Guide Homework
- Projects
- Textbook
- Grade allocation
 - Final Exam: 30%
 - Final Project: 65%
 - Homework 2%
 - Attendance 3%

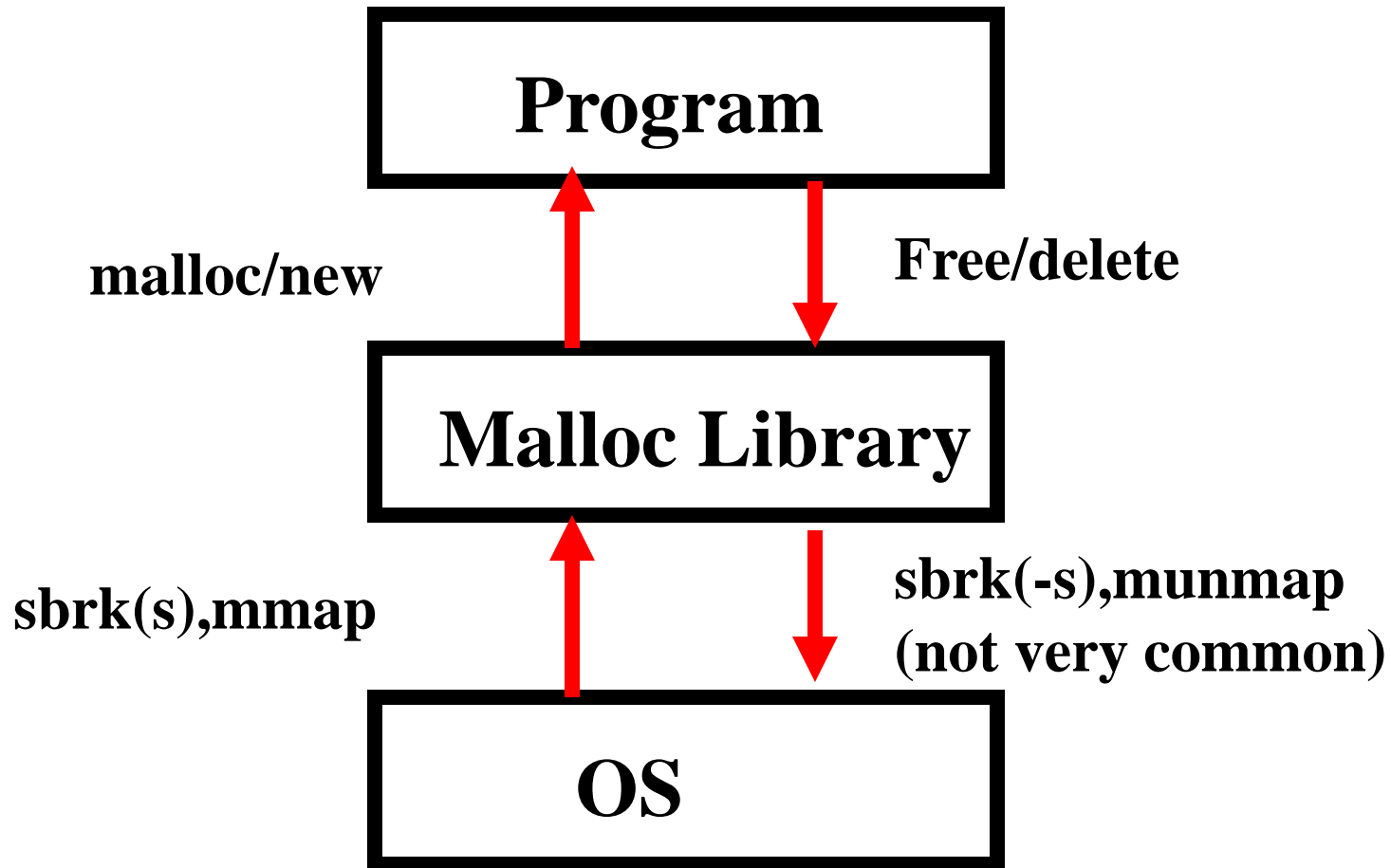
Explicit Memory Management

- In explicit memory management a program frees objects by calling free/delete explicitly
- The program uses a “malloc library” usually provided by the C standard library libc.
- Memory is requested from the OS and added to a free list.
- Subsequent requests are satisfied from memory from the free list.
- Free/delete returns memory to the free list.

Explicit Memory Management

- Memory is requested to the OS in big chunks (8KB).
- This decreases the number of times that memory is requested from the OS
- Also, it is difficult to return memory to the OS since the OS handles memory in pages and memory allocator handle memory in bytes.

Explicit Memory Management

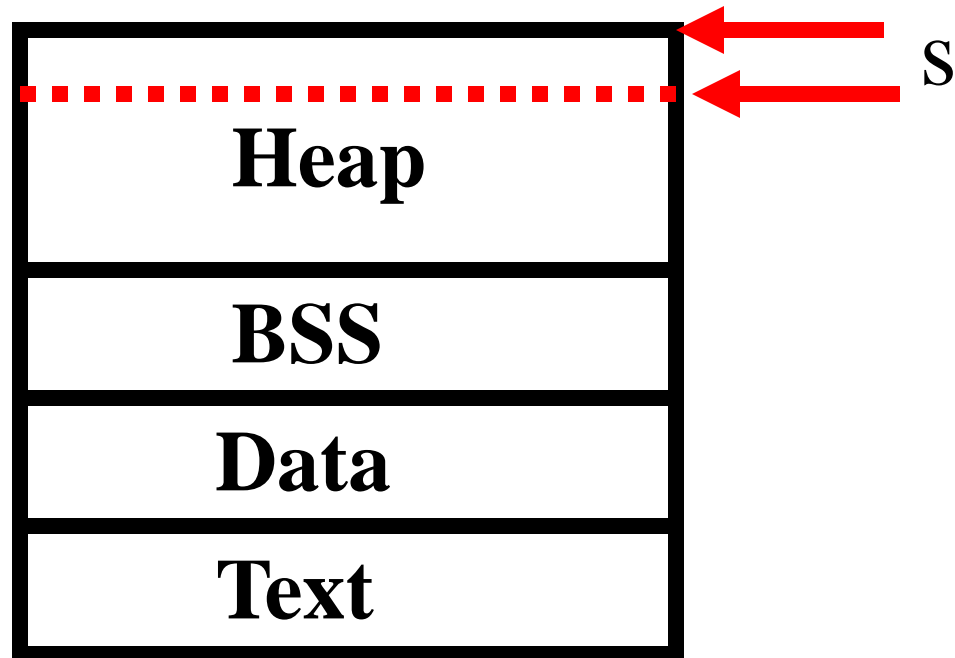


Explicit Memory Management

- When the memory allocator runs out of memory in the free list it calls `sbrk(s)`.
- `Sbrk(s)` increases the size of the heap by `s` bytes.
- `Sbrk` returns a pointer to the old heap limit.
- You can decrease the size of the heap by passing a negative size to `sbrk`. This will shrink the heap.

Explicit Memory Management

sbrk(s)



Explicit Memory Management

- An alternative to `sbrk()` is to use `mmap` with an anonymous mapping.
- This will return a group of pages initialized with 0's.
- You can return memory to the OS by using `munmap`.

Explicit Memory Management

- Returning memory back to the OS is difficult because:
 - Sbrk only can return memory to the OS if it happens to be at the end of the heap.
 - Munmap can return memory to the OS if whole pages are unused.
- Most memory allocators do not return memory to the OS. However this has not been a problem.

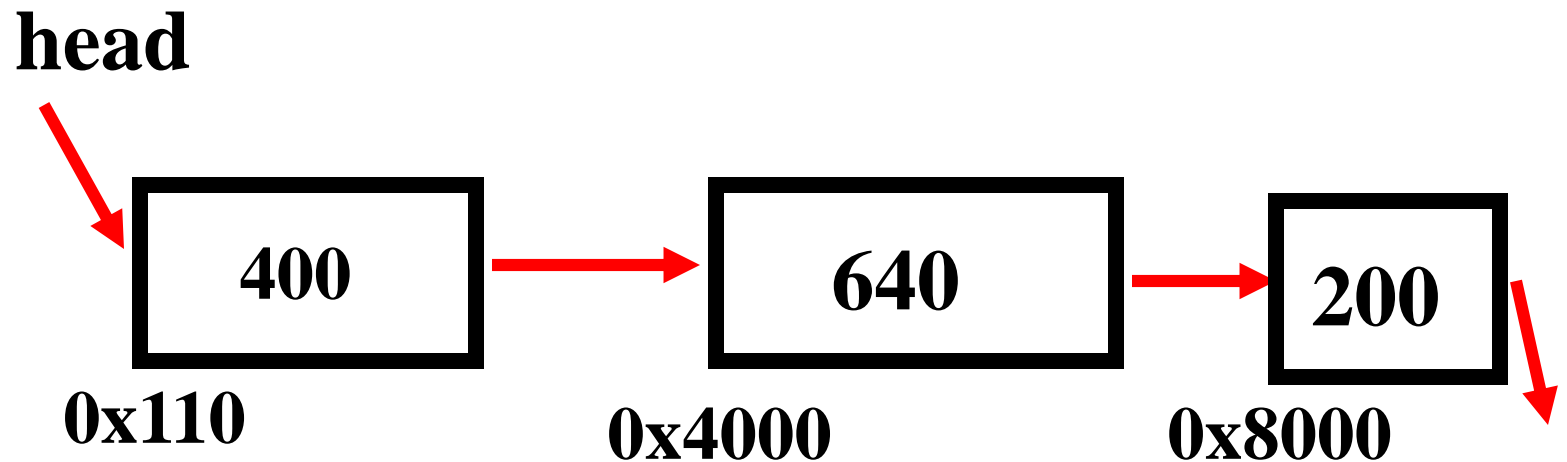
Malloc Implementation

- There are several data structures that can be used for memory allocation:
 - Single Free list
 - Segregated Free Lists
 - Cartesian Tree
 - Boundary tags

Malloc Single Free List

- In this implementation of malloc all the free memory is stored in a single free list.
- Each chunk in the free list has a header with the size of the chunk and a pointer to the next chunk.

Malloc Single Free List



Malloc Single Free List

- During allocation a chunk of the right size is searched, split if necessary and the remainder is returned to the free list.
- When an object is freed, the free list is traversed to find the right place of insertion and the object is coalesced if possible.
- If the request cannot be satisfied with the existing free list, the allocator gets memory from the OS.
- The request for memory to the OS is larger (E.g. 8KB) than the size requested by malloc. This reduces the number of times memory is requested from the OS since this is slow.
- Chunks in the list are ordered by address
 - Allocation: $O(n)$ where $n = \#$ of chunks
 - Free: $O(n)$

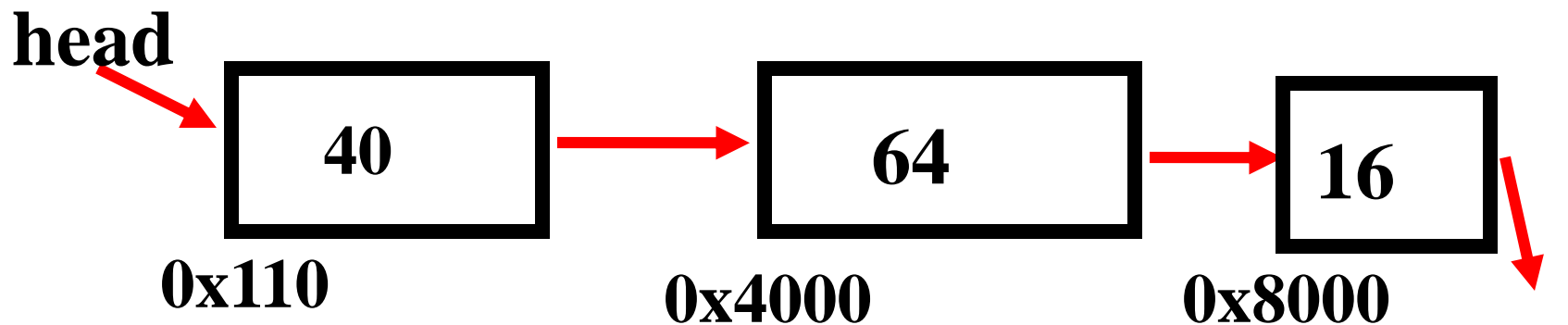
Malloc Single Free List

- Two Allocation Policies:
 - First Fit:
 - Use the first chunk in the free list that satisfies the request.
 - BestFit
 - Choose smallest chunk that satisfies the request.

Notes on malloc

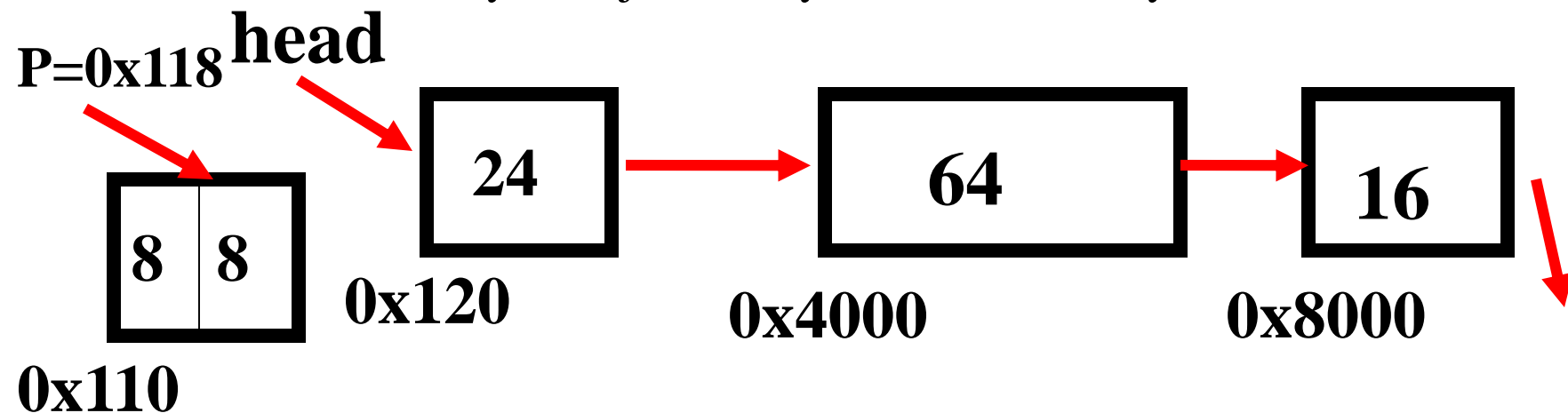
- Malloc also allocates memory for a header. The header stores the size of the object.
- The header will be needed when the object is freed.
- Malloc will return a pointer after the header.
- The memory returned by malloc is aligned to 8 bytes, that is, the address of the object is a multiple of 8 bytes.
- This is because RISC architectures need certain types to be stored in aligned memory. Doubles have to be stored in 8 byte boundaries in memory.
- If you try to store an int value into a non-aligned address in SPARC you will get a SIGBUS error. In the x86 architecture you will not get an error but the execution will slow down.

First Fit

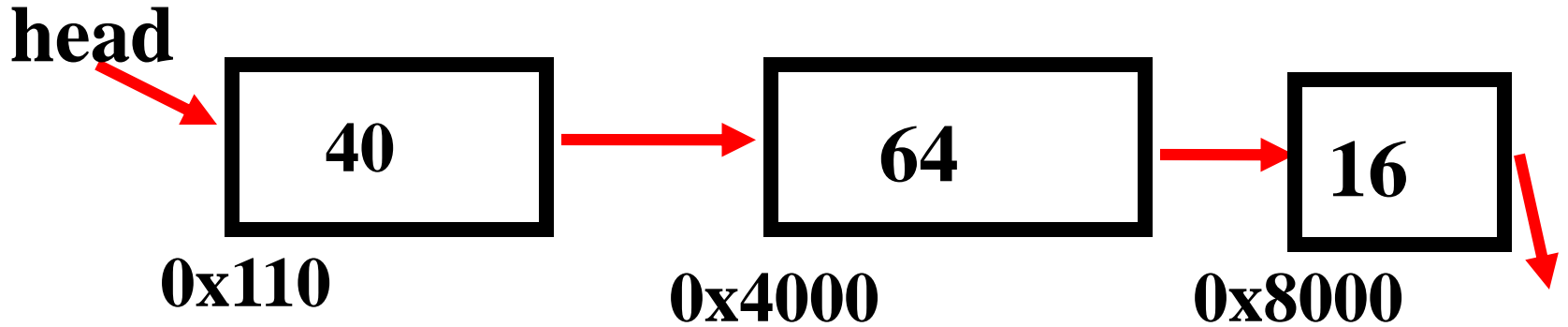


`p = malloc(8);`

8 bytes object + 8 bytes header = 16 bytes total

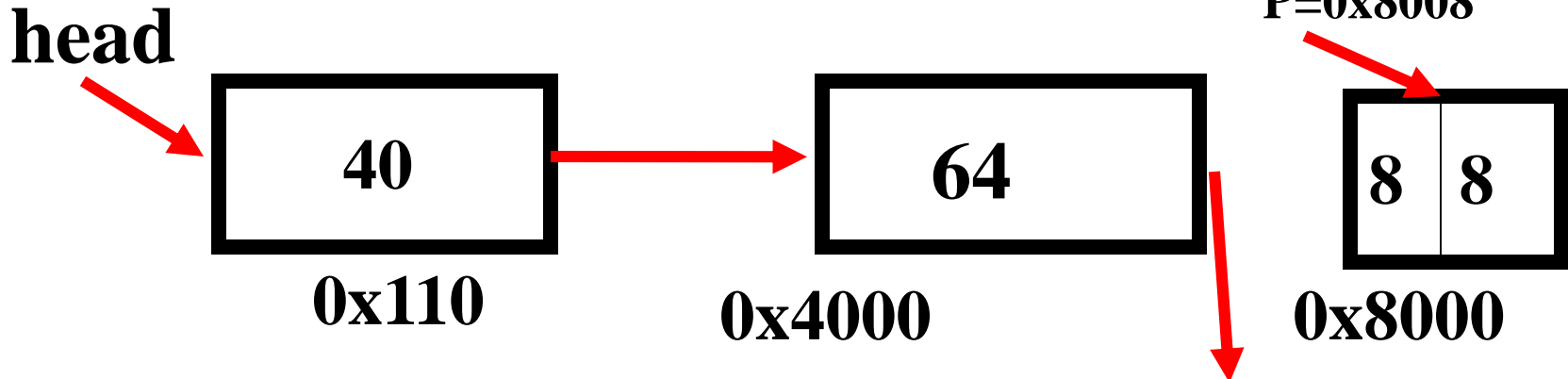


Best Fit



`p = malloc(8);`

8 bytes object + 8 bytes header = 16 bytes total
P=0x8008

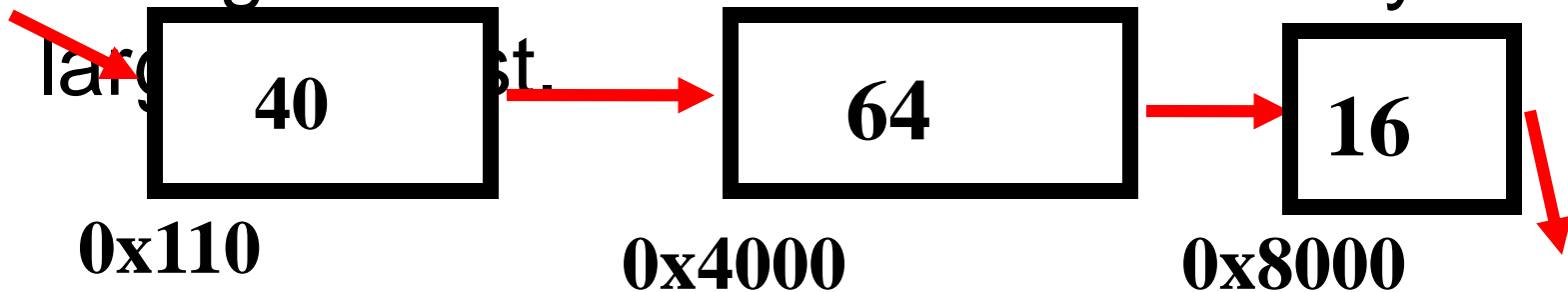


First Fit vs. Best Fit

- First Fit is faster
- Best Fit uses less space.
- In first fit if the search always starts at the beginning, it will create many small objects at the beginning of the list.
- Next fit:
 - It is like first fit but the next search starts where the last ended. This speeds first fit and it will prevent the accumulations of small chunks at the beginning of the list.

External Fragmentation

- External Fragmentation is the waste of memory in the free list due to small non-contiguous blocks that cannot satisfy a



```
p = malloc(100);
```

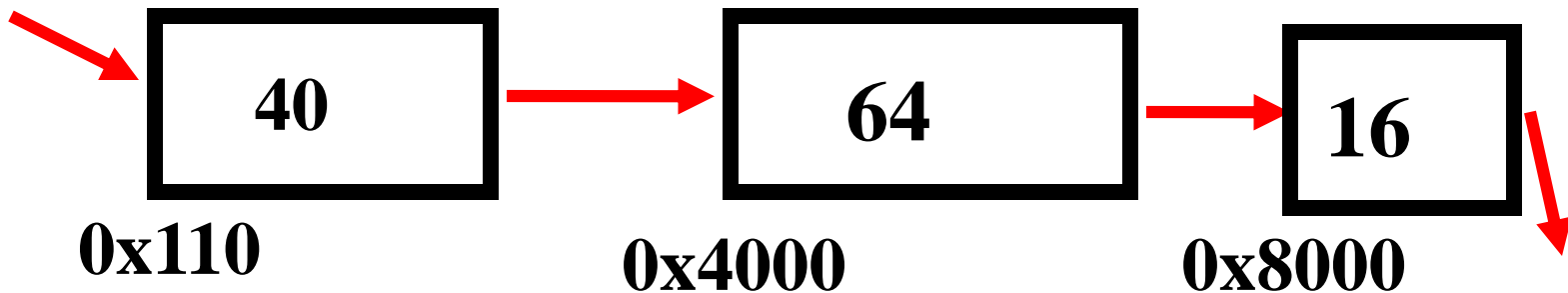
100 bytes object + 8 bytes header = 108 bytes total

The allocation cannot be satisfied even though the free list has more than 108 bytes. The allocator will need to get more memory from the OS.

External Fragmentation

- The external fragmentation can be measured as:

$$\text{Ext. Fragmentation\%} = 100 * (1 - \text{size_largest_block} / \text{total_mem_in_free_list})$$

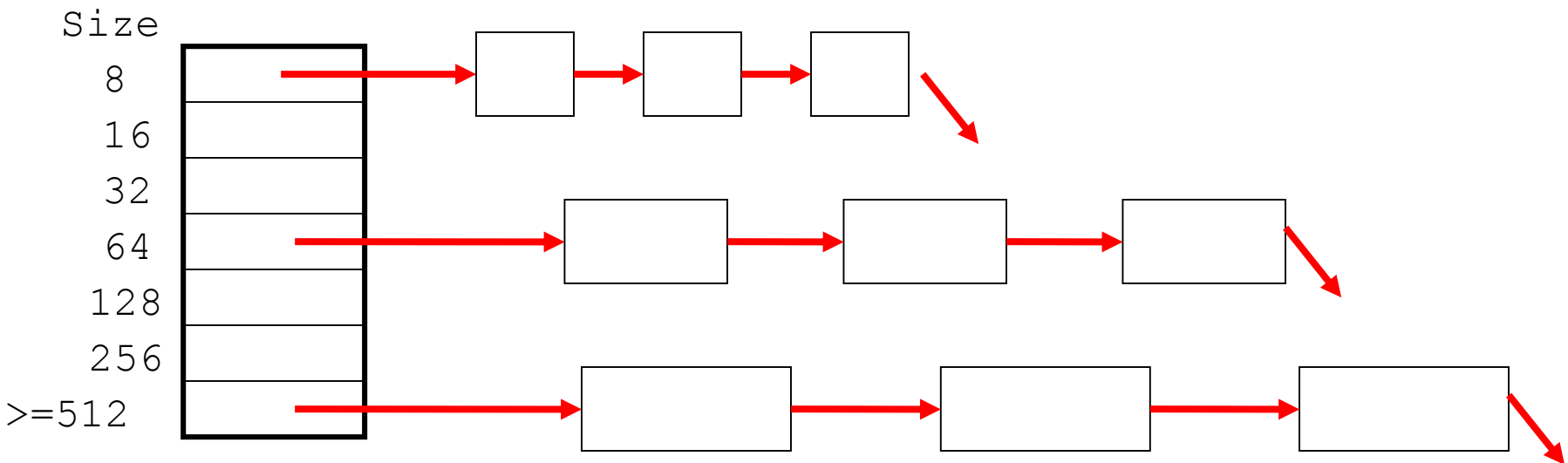


$$\text{Ext. Fragmentation\%} = 100(1 - 64/120) = 47\%$$

If there is only one block in the list ext fragmentation=0%

Segregated Free Lists

- It is another data structure for memory allocation.
- There are multiple free lists for different sizes.



Segregated Free Lists

- Very often the fixed sizes are powers of two.
- Objects are allocated from the free list of the nearest size.
- If a free list is empty, the allocator gets a page of memory from the OS and populates the corresponding free list.
- Some implementations of free lists do not have coalescing. Once an object is created for a size it will be of that size for the entire execution of the program.

Segregated Free Lists

- Coalescing an object would require traversing all the blocks to find out if the neighboring blocks are free
- With segregated free lists:
 - Allocation (small objects) = $O(1)$
 - Free (small objects) = $O(1)$ no coalescing.
- Segregated free list allocators are fast but they use more memory.
- The BSD UNIX allocator uses segregated free lists

External and Internal Fragmentation

- External Fragmentation:
 - Waste of memory due to having non-contiguous blocks in the free list that cannot satisfy an allocation for a large object.
- Internal Fragmentation:
 - Waste of memory due to the allocator returning a larger block than the memory requested.
 - Internal Frag = $100 * (1 - \text{Size_requested} / \text{Mem Allocated})$

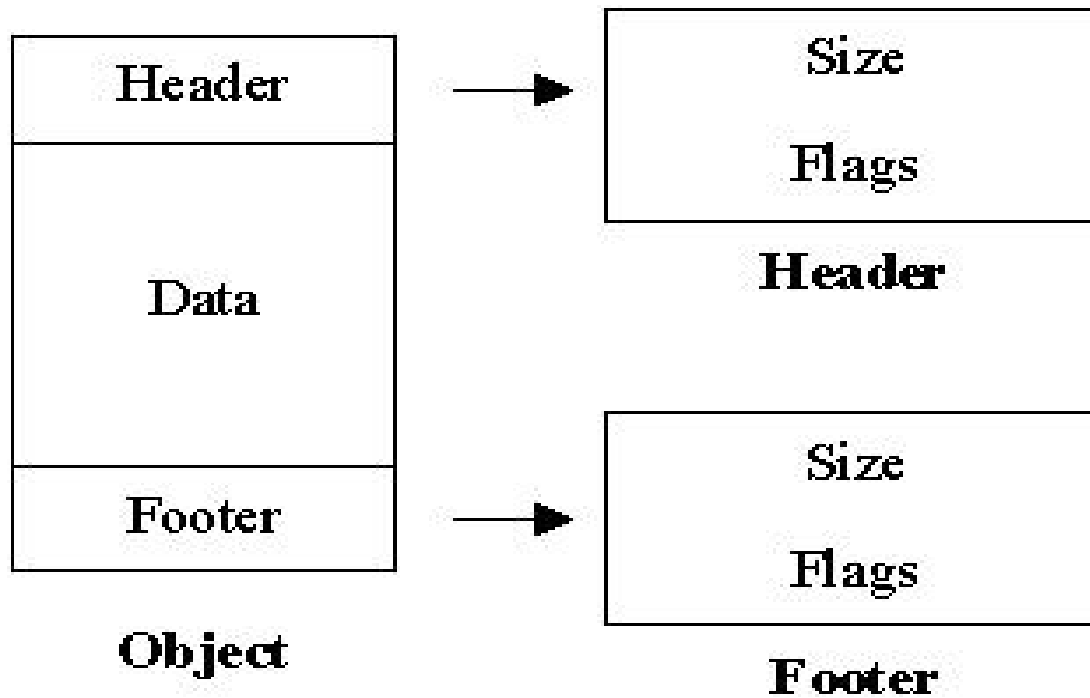
Cartesian Trees

- The free objects are organized as a tree
- The tree is organized by size in the Y axis and by address in the X axis.
- This allows allocation and freeing in $O(\log n)$
- The SunOS allocator uses Cartesian Trees.

Boundary Tags

- Boundary Tags allow identifying the neighboring objects in $O(1)$. This allows coalescing of objects in $O(1)$
- Each object ***allocated or free*** has a header and a footer.
- The header and the footer contain the size of the object as well as a flag that tells if the object is allocated or not.
- When freeing an object it is possible to check if the neighbors are free in constant time and coalesce them if possible.

Boundary Tags



Flags:

0 - Free

1 - Allocated

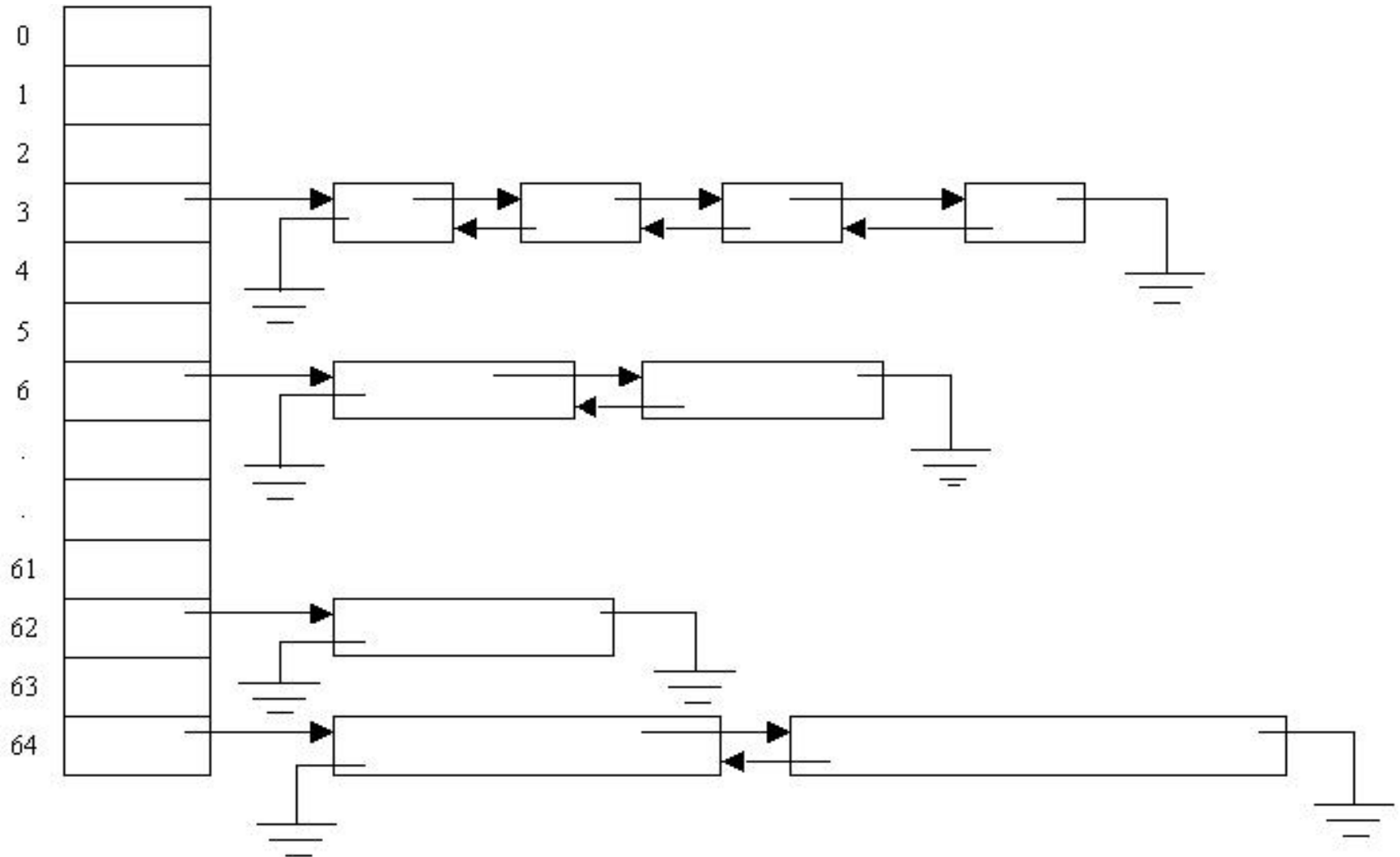
Boundary Tags

Header 1
Data 1
Footer 1
Header 2
Data 2
Footer 2
Header 3
Data 3
Footer 3
...
...

Boundary Tags

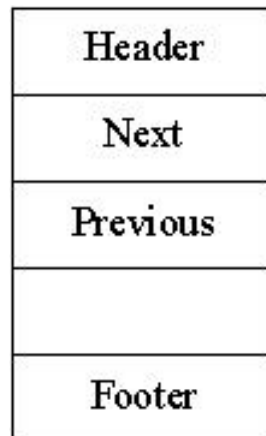
- The memory allocator you will implement also stores the objects in segregated free lists.
- This allows memory allocation in $O(1)$ for small objects.
- There are 65 lists. Lists 0 to 64 store objects of sizes $8 * i$ including header and footer where i is the number of the list.
- List 65 stores objects larger or equal than 512

Boundary Tags



Boundary Tags

- # When the object is free, the header also contains a pointer to the next object in the list and a pointer to the previous object in the list.
- # This allows removal of an object in the middle of a list from a list in constant time. That is needed after coalescing.



Free Object

Allocation Algorithm

1. Round up requested size to the next 8 byte boundary (if size ==0 assume 1 byte data)
2. Add the size of the header and the footer:
 - # $\text{real_size} = \text{roundup8}(\text{requested size}) + \text{sizeof}(\text{header}) + \text{sizeof}(\text{footer})$
3. Lookup the corresponding list
 - # $\text{list} = \text{real_size} \geq 512 ? 64 : ((\text{real_size}) \gg 3)$
4. If the list is non empty and list <64, remove a block from that list and return it.

Allocation Algorithm

5. If the list is empty, search for a list with objects of larger size. If there is one non-empty list with larger objects, remove one of the objects and split it. Use one portion of the object to satisfy the allocation request and return the remainder to the corresponding list.
6. If the remainder is smaller than the size of the header plus the footer plus next plus previous making the object unusable, then do not split the object and use the entire object to satisfy the allocation.

Allocation Algorithm

7. If there is not enough memory in the free lists to satisfy the allocation, then request memory from the OS using `sbrk()`. Round up the request of memory to the OS to a 16KB boundary.

Free Algorithm

1. Check the footer of the left neighbor object (the object just before the object being freed) to see if it is also free. If that is the case, remove the left neighbor from its free list using the previous and next pointers and coalesce this object with the object being freed.

Free Algorithm

2. Check the header of the right neighbor object (the object just after the object being freed) to see if it is also free. If that is the case, remove the right neighbor from its free list using the previous and next pointers and coalesce it with the object being freed.
3. Place the freed object in the corresponding free list and update the header and footer.

Fence Posts

- If the object freed is at the beginning of the heap, it is likely that your allocator will try erroneously to coalesce memory beyond the beginning of the heap.
- Also other libraries may call `sbrk()` causing a hole in the heap.

Fence Posts

- To prevent coalescing with memory beyond the beginning or the end of the chunks that belong to the heap your allocator will:
 - every time a chunk of memory is requested from the OS, your allocator has to add a dummy footer at the beginning of the chunk with the flag set to “allocated”.
 - Also at the end of the chunk you will add a header with the flag set to “allocated”.
- If two chunks are consecutive, you should remove the fence posts between them.

Fence Posts

End of Heap

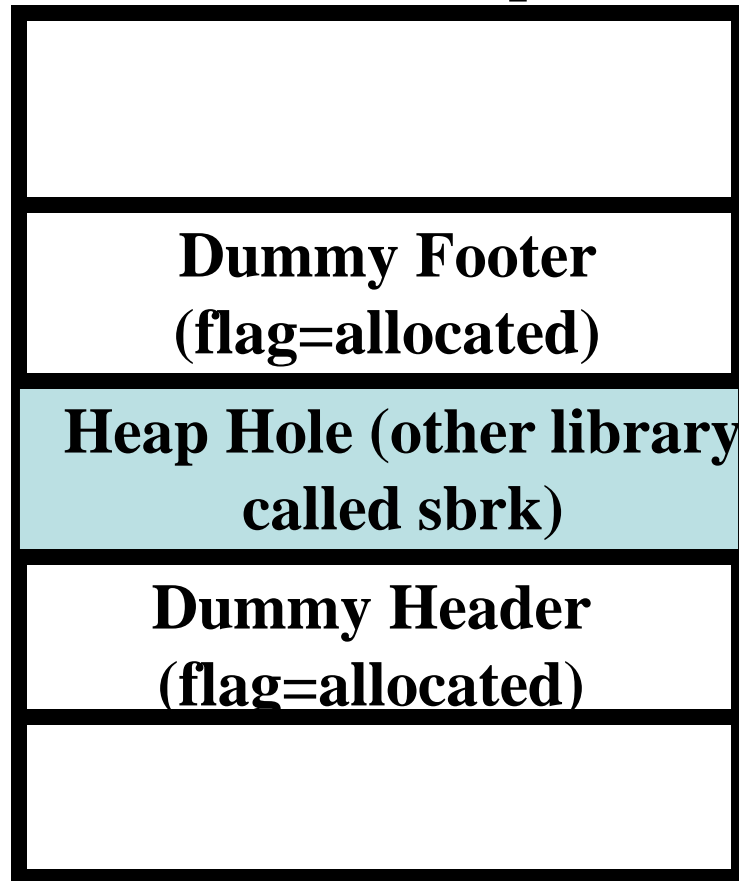
Dummy Header
(flag=allocated)

Dummy footer
(flag=allocated)

Start of Heap

Fence Posts

End of Heap



Start of Heap

DL Malloc

- The memory allocator you are building is based on Doug Lea's malloc implementation that is public domain:
<http://g.oswego.edu/dl/html/malloc.html>
- This allocator is the standard allocator in Linux