

# C++ Mini-Course

- Part 1: Mechanics
- Part 2: Basics
- Part 3: References
- Part 4: Const
- Part 5: Inheritance
- Part 6: Libraries
- Part 7: Conclusion



C Rulez!



C++ Rulez!

# **C++ Mini-Course**

## **Part 1: Mechanics**

# C++ is a superset of C

- New Features include
  - Classes (Object Oriented)
  - Templates (Standard Template Library)
  - Operator Overloading
  - Slightly cleaner memory operations

# Some C++ code

## Segment.h

```
#ifndef __SEGMENT_HEADER__
#define __SEGMENT_HEADER__

class Point;
class Segment
{
public:
    Segment();
    virtual ~Segment();
private:
    Point *m_p0, *m_p1;
};

#endif // SEGMENT HEADER
```

## Segment.cpp

```
#include "Segment.h"
#include "Point.h"

Segment::Segment()
{
    m_p0 = new Point(0, 0);
    m_p1 = new Point(1, 1);
}

Segment::~~Segment()
{
    delete m_p0;
    delete m_p1;
}
```

# #include

`#include "Segment.h"`

Insert header file at this point.

`#include <iostream>`

Use library header.

# Header Guards

```
#ifndef __SEGMENT_HEADER__  
#define __SEGMENT_HEADER__  
  
// contents of Segment.h  
//...  
  
#endif
```

- To ensure it is safe to include a file more than once.

# Header Guards

**#ifndef**

**#define**

```
// contents of segment.H  
//...
```

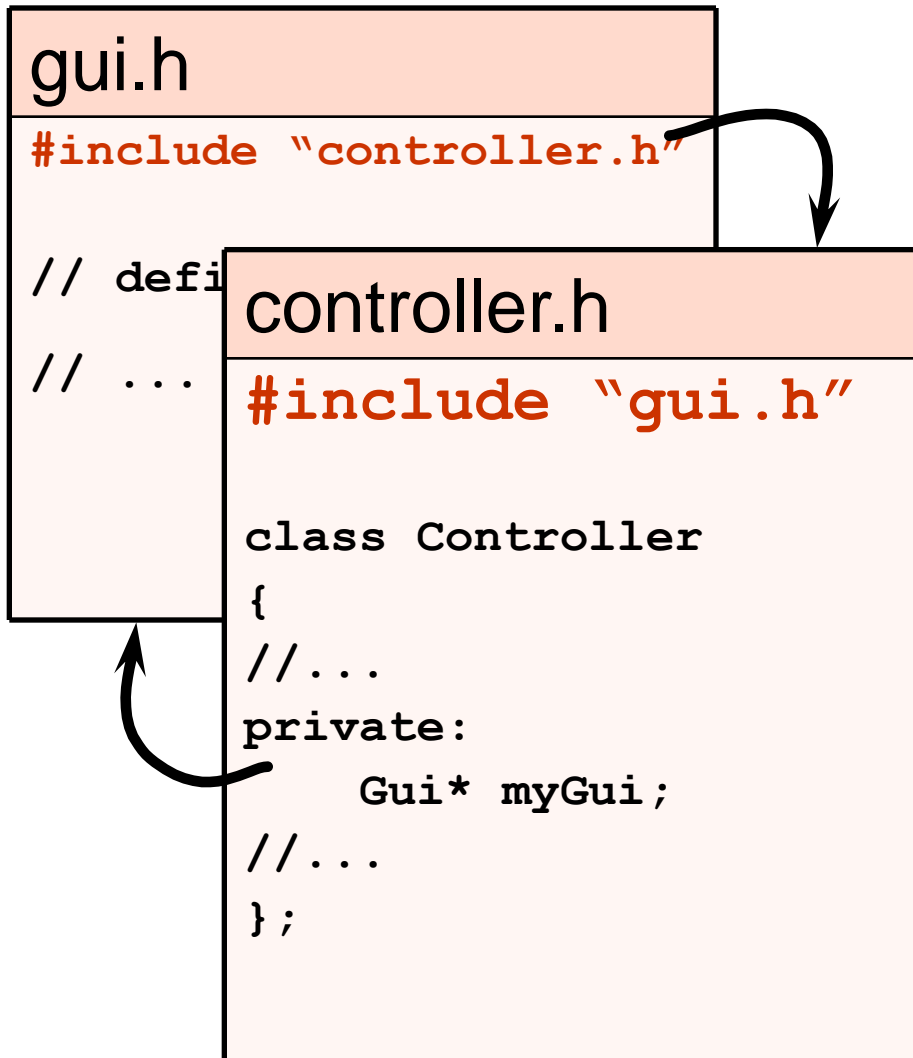
**#endif**

If this variable is  
not defined...

Define it.

End of guarded area.

# Circular Includes



- What's wrong with this picture?
- How do we fix it?



# Forward Declarations

gui.h

```
//Forward Declaration  
class Controller;
```

```
// defi  
// ...
```

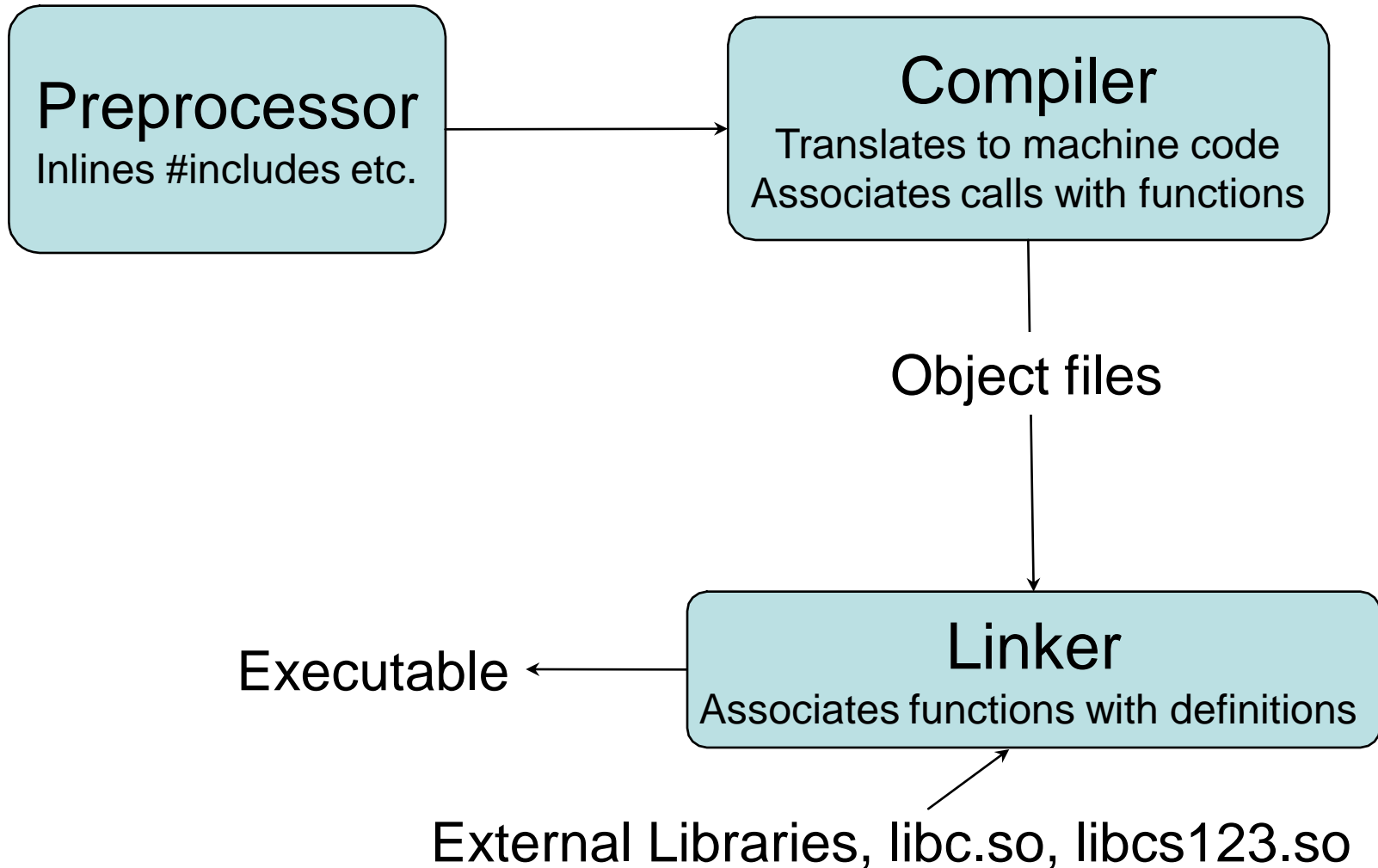
controller.h

```
//Forward declaration  
class Gui;
```

```
class Controller  
{  
//...  
private:  
    Gui* myGui;  
//...  
};
```

- In header files, only include what you must.
- If only pointers to a class are used, use forward declarations.

# Compilation



# OK, OK. How do I run my Program?

```
> make
```

And if all goes well...

```
> ./myprog
```

# **C++ Mini-Course**

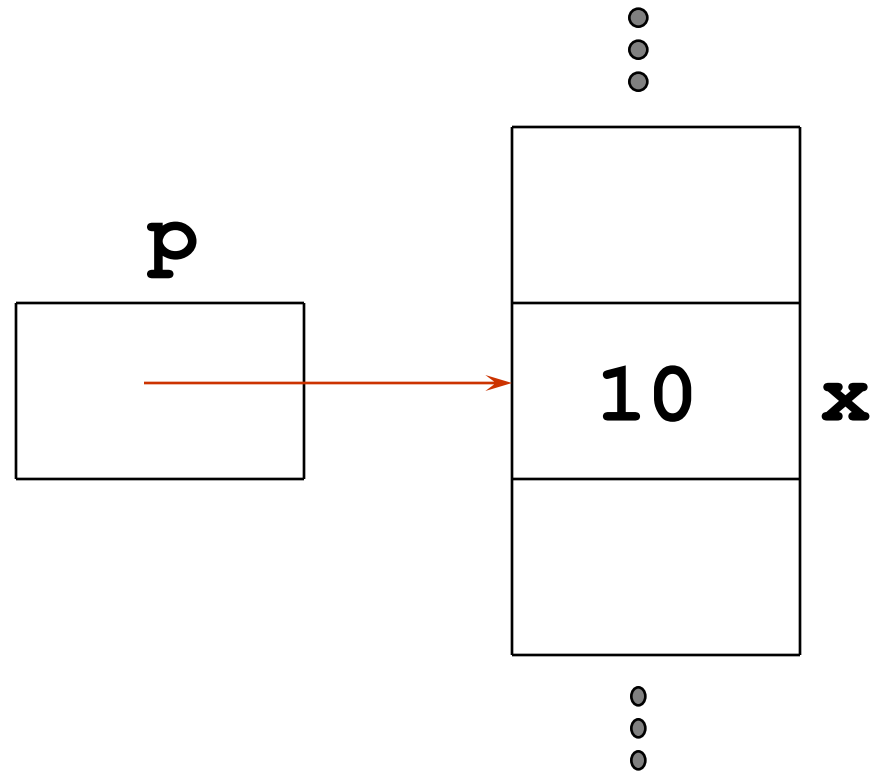
## **Part 2: Basics**

# What is a pointer?

```
int x = 10;
```

```
int *p;
```

```
p = &x;
```



**p** gets the address of **x** in memory.

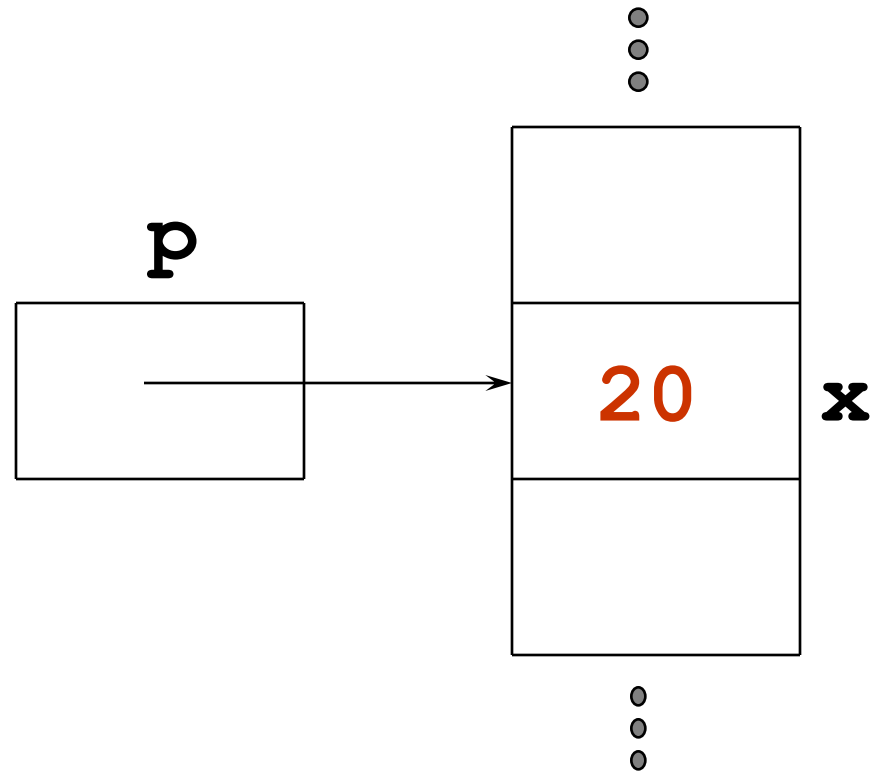
# What is a pointer?

```
int x = 10;
```

```
int *p;
```

```
p = &x;
```

```
*p = 20;
```



**\*p** is the value at the address **p**.

# What is a pointer?

```
int x = 10;  
int *p = NULL;
```

Declares a pointer  
to an integer

```
p = &x;
```

& is **address** operator  
gets address of x

```
*p = 20;
```

\* **dereference** operator  
gets value at the location  
stored in **p**

# Allocating memory using **new**

```
Point *p = new Point(5, 5);
```

- **new** can be thought of as a function with slightly strange syntax
- **new** allocates space to hold the object.
- **new** calls the object's constructor.
- **new** returns a pointer to that object.



# Deallocating memory using **delete**

```
// allocate memory
Point *p = new Point(5, 5);
...
// free the memory
delete p;
p = NULL;
```

For every call to **new**, there must be exactly one call to **delete**. It's a good practice to set to **NULL** afterwards to protect against double deletes.

# Using **new** with arrays

```
int x = 10;  
int* nums1 = new int[10]; // ok  
int* nums2 = new int[x]; // ok
```

- Initializes an array of 10 integers on the heap.
- Equivalent to the following C code

```
int* nums = (int*)malloc(x * sizeof(int));
```

- Equivalent to the following Java code
- ```
int[] nums = new int[x];
```

# Using **new** with multidimensional arrays

```
int x = 3, y = 4;  
int** nums3 = new int[x][4]; // ok  
int** nums4 = new int[x][y]; // BAD!
```

- Initializes a multidimensional array
- Only the first dimension can be a variable. The rest must be constants.
- Use single dimension arrays to fake multidimensional ones

# Using **delete** on arrays

```
// allocate memory  
int* nums1 = new int[10];  
int* nums3 = new int[x][4][5];
```

...

```
// free the memory  
delete[] nums1;  
delete[] nums3;
```

- Have to use **delete []**, or else only the first element is deleted.

# Destructors

- `delete` calls the object's **destructor**.
- `delete` frees space occupied by the object.
- A **destructor** cleans up after the object.
- Releases resources such as memory.

# Destructors – an Example

```
class Segment
{
public:
    Segment();
    virtual ~Segment();
private:
    Point *m_p0, *m_p1;
};
```

# Destructors – an Example

```
Segment::Segment()
```

```
{
```

```
    m_p0 = new Point(0, 0);
```

```
    m_p1 = new Point(1, 1);
```

```
}
```

```
Segment::~~Segment()
```

```
{
```

```
    delete m_p0;
```

```
    delete m_p1;
```

```
}
```

# New vs Malloc

- **Never** mix new/delete with malloc/free

| Malloc                                                                                                     | New                                                                                              |
|------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| Standard C Function                                                                                        | Operator (like ==, +=, etc.)                                                                     |
| Used sparingly in C++; used frequently in C                                                                | Only in C++                                                                                      |
| Used for allocating chunks of memory of a given size without respect to what will be stored in that memory | Used to allocate instances of classes / structs / arrays and will invoke an object's constructor |
| Returns void* and requires explicit casting                                                                | Returns the proper type                                                                          |
| Returns NULL when there is not enough memory                                                               | Throws an exception when there is not enough memory                                              |
| Every malloc() should be matched with a free()                                                             | Every new/new[] should be matched with a delete/delete[]                                         |



# Syntactic Sugar “->”

```
Point *p = new Point(5, 5);
```

```
// Access a member function:
```

```
(*p).move(10, 10);
```

```
// Or more simply:
```

```
p->move(10, 10);
```

# Stack vs. Heap

On the Heap /  
Dynamic allocation

```
drawStuff() {  
    Point *p = new Point();  
    p->move(10,10);  
    //...  
}
```

On the Stack /  
Automatic allocation

```
drawStuff() {  
    Point p();  
    p.move(5,5);  
    //...  
}
```

What happens when **p** goes out of scope?

# Summary with Header File

header file

begin header  
guard

forward declaration

class declaration

constructor

destructor

member variables

need semi-colon

end header guard

## Segment.h

```
#ifndef __SEGMENT_HEADER__
#define __SEGMENT_HEADER__

class Point;
class Segment {
    public:
        Segment();
        virtual ~Segment();
    protected:
        Point *m_p0, *m_p1;
};

#endif // __SEGMENT_HEADER__
```

# Syntax Note

- The following two statements mean the same thing
- They both allocate space for a Point object on the stack:
  - `Point p ();`
  - `Point p = Point ();`

# **C++ Mini-Course**

## **Part 3: References**

# Passing by value

```
void Math::square(int i) {  
    i = i*i;  
}
```

```
int main() {  
    int i = 5;  
    Math::square(i);  
    cout << i << endl;  
}
```

# Passing by reference

```
void Math::square(int &i) {  
    i = i*i;  
}
```

```
int main() {  
    int i = 5;  
    Math::square(i);  
    cout << i << endl;  
}
```

# What is a reference?

- An alias – another name for an object.

```
int x = 5;
```

```
int &y = x; // y is a  
           // reference to x
```

```
y = 10;
```

- What happened to x?
- What happened to y?



# What is a reference?

- An alias – another name for an object.

```
int x = 5;
```

```
int &y = x; // y is a  
           // reference to x
```

```
y = 10;
```

- What happened to x?
- What happened to y? – **y is x.**

# Why are they useful?

- Unless you know what you are doing, do not pass objects by value; either use a pointer or a reference.
- Some people find it easier to deal with references rather than pointers, but in the end there is really only a syntactic difference (neither of them pass by value).
- Can be used to return more than one value (pass multiple parameters by reference)

# Passing by reference: the bottom line

- The syntax is as though the parameter was passed by value.
- But behind the scenes, C++ is just passing a pointer.
- The following two are basically the same thing:

```
void increment(int &i)    void increment(int *i)
{
    i = i + 1;
}
...
int i;
increment(i);

{
    *i = i + 1;
}
...
int i;
increment(&i);
```

# How are references different from Pointers?

| Reference                                                                                                   | Pointer                                                                                                           |
|-------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| <code>int &amp;a;</code>                                                                                    | <code>int *a;</code>                                                                                              |
| <code>int a = 10;</code><br><code>int b = 20;</code><br><code>int &amp;c = a;</code><br><code>c = b;</code> | <code>int a = 10;</code><br><code>int b = 20;</code><br><code>int *c = &amp;a;</code><br><code>c = &amp;b;</code> |

# Asterisks and Ampersands

- In a type declaration, ‘\*’ indicates that you are declaring a pointer type.
  - Otherwise ‘\*’ is a dereference operator—gets the actual object from a pointer to the object.
- In a type declaration, ‘&’ indicates that you are declaring a reference.
  - Otherwise ‘&’ is the “address of” operator—gets a pointer to an object from the object itself.

# **C++ Mini-Course**

## **Part 4: const**

# Introducing: `const`

```
void Math::printSquare(const int &i) {  
    i = i*i; ← Won't compile.  
    cout << i << endl;  
}
```

```
int main() {  
    int i = 5;  
    Math::printSquare(i);  
    Math::printCube(i);  
}
```

# Can also pass pointers to `const`

```
void Math::printSquare(const int *pi) {  
    *pi = (*pi) * (*pi);  
    cout << pi << endl;  
}
```

← Still won't compile.

```
int main() {  
    int i = 5;  
    Math::printSquare(&i);  
    Math::printCube(&i);  
}
```



# Declaring things `const`

```
const River nile;
```

```
const River* nilePc;
```

```
River* const nileCp;
```

```
const River* const nileCpc
```

# Read pointer declarations right to left

```
// A const River  
const River nile;
```

```
// A pointer to a const River  
const River* nilePc;
```

```
// A const pointer to a River  
River* const nileCp;
```

```
// A const pointer to a const River  
const River* const nileCpc
```

# Let's Try References

```
River nile;
```

```
const River &nileC = nile;
```

```
// Will this work?
```

```
River &nile1 = nileC;
```

# How does `const` work here?

```
void Math::printSquares(const int &j,  
    int &k) {  
    k = k*k;    // Does this compile?  
    cout << j*j << ", " << k << endl;  
}
```

```
int main() {  
    int i = 5;  
    Math::printSquares(i, i);  
}
```

# Returning `const` references is OK

```
class Point {  
public:
```

```
    const double &getX() const;
```

```
    const double &getY() const;
```

```
    void move(double dx, double dy);
```

```
protected:
```

```
    double m_x, m_y;
```

```
};
```

```
const double &  
Point::getX() const {  
    return m_x;  
}
```

Function won't  
change `*this`.

# **C++ Mini-Course**

## **Part 5: Inheritance**

# Classes vs Structs

- Default access specifier for classes is private; for structs it is public
- Except for this difference, structs are functionally the same as classes, but the two are typically used differently: structs should be thought of as lightweight classes that contain mostly data and possibly convenience methods to manipulate that data and are hardly ever used polymorphically

```
struct Point {
    int x;
    int y;

    // convenience constructor
    Point(int a, int b)
        : x(a), y(b)
    { }

    // @returns distance to another point
    double distance(const Point &pnt) {
        int dx = m_x - pnt.x;
        int dy = m_y - pnt.y;
        return math.sqrt(dx*dx + dy*dy);
    }
};
```

```
class Segment {
public:
    Segment();
    virtual ~Segment();

    void setPoints(int x0, int y0, int x1, int y1);

protected:
    Point *m_p0, *m_p1;
};

void Segment::setPoints(int x0, int y0, int x1, int y1) {
    m_p0 = new Point(x0, y0);
    m_p1 = new Point(x1, y1);
}
```

# How does inheritance work?

must include parent header file

DottedSegment publicly inherits from Segment

```
#include "Segment.h"  
class DottedSegment : public Segment  
{  
    // DottedSegment declaration  
};
```



# virtual

- In Java every method invocation is dynamically bound, meaning for every method invocation the program checks if a sub-class has overridden the method. You can disable this (somewhat) by using the keyword “final” in Java
- In C++ you have to declare the method virtual if you want this functionality. (So, “virtual” is the same thing as “not final”)
- You should declare methods virtual when they are designed to be overridden or will otherwise participate in an inheritance hierarchy.

# pure virtual functions

- In Java, the “abstract” keyword means the function is undefined in the superclass.
- In C++, we use pure virtual functions:
  - `virtual int mustRedfineMe(char *str) = 0;`
  - This function must be implemented in a subclass.

# Resolving functions

In Java:

```
// Overriding methods
public void overloaded() {
    println("woohoo");
    super.overloaded();
}
```

```
//constructor
public Subclass() {
    super();
}
```

In C++:

```
// Overriding methods
void Subclass::overloaded() {
    cout<<"woohoo"<<endl;
    Superclass::overloaded();
}
```

```
//constructor
public Subclass() :
    Superclass()
{ }
```

# Make destructors virtual

- Make sure you declare your destructors virtual; if you do not declare a destructor a non-virtual one will be defined for you

```
Segment ();  
virtual ~Segment ();
```

this is important

# **C++ Mini-Course**

## **Part 6: Libraries**

# Namespaces

- Namespaces are kind of like packages in Java
- Reduces naming conflicts
- Most standard C++ routines and classes and under the `std` namespace
  - Any standard C routines (malloc, printf, etc.) are defined in the global namespace because C doesn't have namespaces

# using namespace

```
#include <iostream>
...
std::string question =
    "How do I prevent RSI?";
std::cout << question << std::endl;
```

```
using namespace std;
```

```
string answer = "Type less.";
cout << answer << endl;
```

Bad practice to do in header files!

# STL

- Standard Template Library
- Contains well-written, templated implementations of most data structures and algorithms
  - Templates are similar to generics in Java
  - Allows you to easily store anything without writing a container yourself
- Will give you the most hideous compile errors ever if you use them even slightly incorrectly!



# STL example

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

typedef vector<Point> PointVector;
typedef PointVector::iterator PointVectorIter;

PointVector v;
v.push_back(Point(3, 5));

PointVectorIter iter;
for(iter = v.begin(); iter != v.end(); ++iter) {
    Point &curPoint = *iter;
}
```

# **C++ Mini-Course**

## **Part 7: Conclusion**

# Other Resources

- The Java To C++ tutorial on the website is probably your best source of information
- The big thick book by Stroustrup in the back of the Sun Lab is the ultimate C++ reference
- A CS 123 TA, or specifically your mentor TA if you have been assigned one

# Question and Answer Session