C++ For Java Programmers

15-494 Cognitive Robotics
Ethan Tira-Thompson

Keyword Mapping

- Java ➙ C++
- Java API ➙ STL (& friends, e.g. Boost)
- Generics ➙ Templates
  - Same syntax:

    `Vector<foo>` ➙ `vector<foo>`
- interfaces ➙ multiple inheritance
- casting: `instanceof` ➙ `dynamic_cast<T>`
- final ➙ ~~virtual~~

C++: Rope to Hang Yourself

- File Layout
- Globals
- Macros
- Memory Management
- Overriding operators
- Multiple Inheritance

C++ File Layout

- In Java, everything goes in the `.java` file, cross-references "just work"
- In C++, the compiler isn't so smart
  - If one class depends on another, the dependent class needs to `#include` the other's file
  - If multiple classes depend on the same file, it might be included more than once
  - Have to wrap headers with a little bit of macro boilerplate:

```
#ifndef INCLUDED_Foo_h_
#define INCLUDED_Foo_h_
/* rest of the file ... */
#endif
```

I recommend this form, assuming a file named "Foo.h". This is an arbitrary choice, simply must be unique.

C++ File Layout

- Two types of C++ files:
  - `.h`: Definitions, documentation, and small implementations

```
//! the venerable "Hello World!"
void sayHello();
```

  - `.cc`: Implementation, aka "Translation unit"

```
void sayHello() {
   std::cout << "Hello World!" << std::endl;
}
```

  - Each translation unit is compiled independently
  - After compilation, units are *linked* into executable

C++ File Layout

- In C++, classes can be completely defined by the `.h` file:

Foo.h:

```
#include <iostream>

class Foo {
public:
  void sayHello() {
    std::cout << "Hello World!" << std::endl;
  }
};
```

you are going to hate this

C++ File Layout

- Do you want everything in the `.h`?
  - Might want to improve readability, or avoid `inline`
  - Split *definition* from *implementation*

Foo.h:

```
#include <iostream>

class Foo {
public:
  void sayHello();
};
```

← System Header

Foo.cc:

```
#include "Foo.h"

void Foo::sayHello() {
  std::cout << "Hello World!" << std::endl;
}
```

← User Header

Scope Specification

Template Usage

- Allows a class to be re-used with a variety of types
- Canonical example: `vector`
  - Want to store a resizable array of data, but it doesn't really matter what the data is
  - `vector<T>`, where `T` is any type: `vector<int>`, `vector<string>`, `vector<Foo>`, etc.
  - A class may make assumptions about type/capabilities of its template argument — results in cryptic compiler errors when the wrong type is passed.

Template Usage

Code:

Now we don't need std:: everywhere

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

int main (int argc, char * const argv[]) {

    vector<int> v;
    for(int i=0; i<10; ++i)
        v.push_back(i); //each call stores a copy of i

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

    //indexed usage
    for(int i=0; i<v.size(); ++i)
        cout << "The " << i << "th element is: " << v[i] << endl;

    return 0;
}
```

Hooray for operator overloading!

Output:

```
0
1
2
3
4
5
6
7
8
9
The 0th element is: 0
The 1th element is: 1
The 2th element is: 2
The 3th element is: 3
The 4th element is: 4
The 5th element is: 5
The 6th element is: 6
The 7th element is: 7
The 8th element is: 8
The 9th element is: 9
```

C++ Memory Management

- For every `new`, there should be a `delete`
  - Arrays have to use '`delete []`'

```
int * a=new int[10];
/* ... */
delete [] a;
```

- Don't use `malloc` / `free` (the old C-style)
  - These functions don't respect constructors or destructors, can cause all kinds of nasty problems.

Pointers vs. References

- When you have a pointer, prepend ‘*’ to access the value pointed to, and use ‘–>’ instead of ‘ . ’ to access members.
- References always return the referenced value
    - Can’t reassign a reference, must define at creation
- An “array” is just a pointer to the first element (no real “array” type)

Pointers vs. References

Pointers

```
int * pi;          // pointer to int
char** ppc;        // pointer to pointer to char
int* ap[15];       // array of 15 pointers to ints
int* f(char*);     // function taking a char* argument; returns a pointer to int

char c = 'a';
char * p = &c;     // p holds the memory address of c
char c2 = *p;      // c2 == 'a'
*p = 'b';          // c == 'b'  ,  c2 unaffected
p = &c2;           // p now holds the address of c2:  *p == c2 == 'a',  c==’b’
```

References

```
int & pi;          // illegal (uninitialized reference)
char&& ppc;        // illegal (no reference to reference)
int& ap[15];       // illegal (can't create array of references)
int& f(char&);     // legal! Function taking a char&, returns a reference to int

char c = 'a';
char & p = c;      // p now references c
char c2 = p;       // c2 == 'a'
p = 'b';           // c == 'b'  ,  c2 unaffected
p = c2;            // c == 'a'  , p still references c:  p == c == c2 == 'a'
```

Thanks: Stroustrup, *The C++ Programming Language*

Pointers vs. References

```
class Foo {
public:
  int member;
  string getName() const { return "Foo"; }
};

Foo a;
Foo & r = a; //reference
Foo * p = &a; //pointer

cout << "Access via value: "
  << a.getName() << " is " << a.member << endl;

cout << "Access via reference: "
  << r.getName() << " is " << r.member << endl;

cout << "Access via pointer: "
  << p->getName() << " is " << p->member << endl;
```

Value vs. Pointer vs. Reference

- When creating functions, you have 3 choices for each argument

1. Pass by value (default)
    - A copy is made of each argument, original untouchable
    - Best for primitive values, but nothing else

```
// good
int f(int x);

// bad, unnecessary copying, slicing (will be explained)
void drawShape(Shape s, Transform t);

// bad, vector does a deep copy -- could be large
void setValues(vector<int> v);
```

Value vs. Pointer vs. Reference

- When creating functions, you have 3 choices for each argument

2. Pass by reference (pointer)
    - Best when you want to allow NULL as a valid argument
    - Sometimes implies passing control of the memory's allocation

```
// only good if you intend to take an array
int f(int * x); //better to say 'f(int x[])' to be clear

// bad use for Shape, requires a value
// good use for Transform, NULL would be acceptable
void drawShape(Shape * s, Transform * t);

// bad, a "set" function requires a non-NULL value
void setValues(vector<int>* v);
```

Value vs. Pointer vs. Reference

- When creating functions, you have 3 choices for each argument

3. Pass by reference (reference)

    - Best for everything else

```
// overkill, unless you intend to modify the value passed
// (e.g. if there are multiple values to return)
int f(int& x);

// good use for Shape, but now Transform is required
// (consider overloading the function)
void drawShape(Shape& s, Transform& t);

// good
void setValues(vector<int>& v);
```

Value vs. Pointer vs. Reference

- Don't forget '`const`'!
  - Get in the habit of using `const` by default, removing it when necessary
  - Say you are creating a function '`muck`' which is not supposed to modify the value it is passed

```
void muck(Foo f); //bad, how big is 'Foo'?
void muck(Foo& f); //bad, might accidentally modify original
void muck(const Foo& f); //good - no copy, and still read-only
```

  - On a related note, if `muck` is a member of a class, and should not modify the class:

```
void muck(const Foo& f) const;
```

Value vs. Pointer vs. Reference

- Thus, the ideal function definitions:

```
// we'll assume f doesn't intend to return a value in x
int f(int x);


// pass Shape by const reference and optional Transform by pointer
void drawShape(const Shape& s, const Transform * t=NULL);
// or use overloading:
void drawShape(const Shape& s);
void drawShape(const Shape& s, const Transform& t);


// last one...
void setValues(const vector<int>& v);
```

Value vs. Pointer vs. Reference

- Don’t return references to local variables

```
Shape& unify(const Shape& s1, const Shape& s2) {
   Shape ans;
   ans = /* however union is done... */
   return ans; // BZZZT, error:
               // ans is disappearing, how can it be referenced?
}
```

- Have to either return a member variable, or allocate on the heap (caller is responsible for deletion)

```
Shape* unify(const Shape& s1, const Shape& s2) {
    Shape * ans = new Shape();
    *ans = /* however union is done */;
    return ans; // caller has to delete
}
```

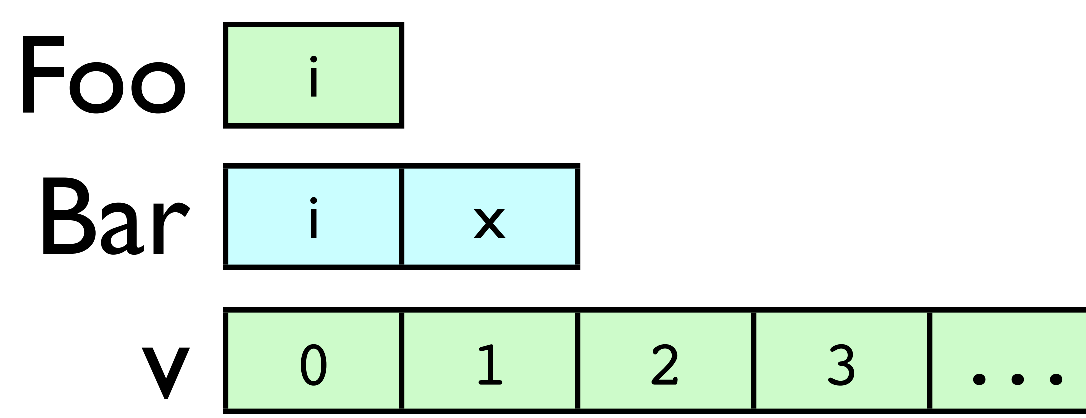
Gotchas: Slicing

- Slicing
  - Store **pointers** when inheritance may be possible
    - can't store references, no way to reassign them
  - Say you have `vector<Foo>`. What happens when you insert a subclass with additional fields?

```
class Foo {
public:
  int i;
};
```

```
class Bar : public Foo {
public:
  int x;
};
```

```
int main (int argc, char * const argv[]) {
  vector<Foo> v; // vector of 'Foo's
  v.push_back(Bar()); // insert a 'Bar'
}
```



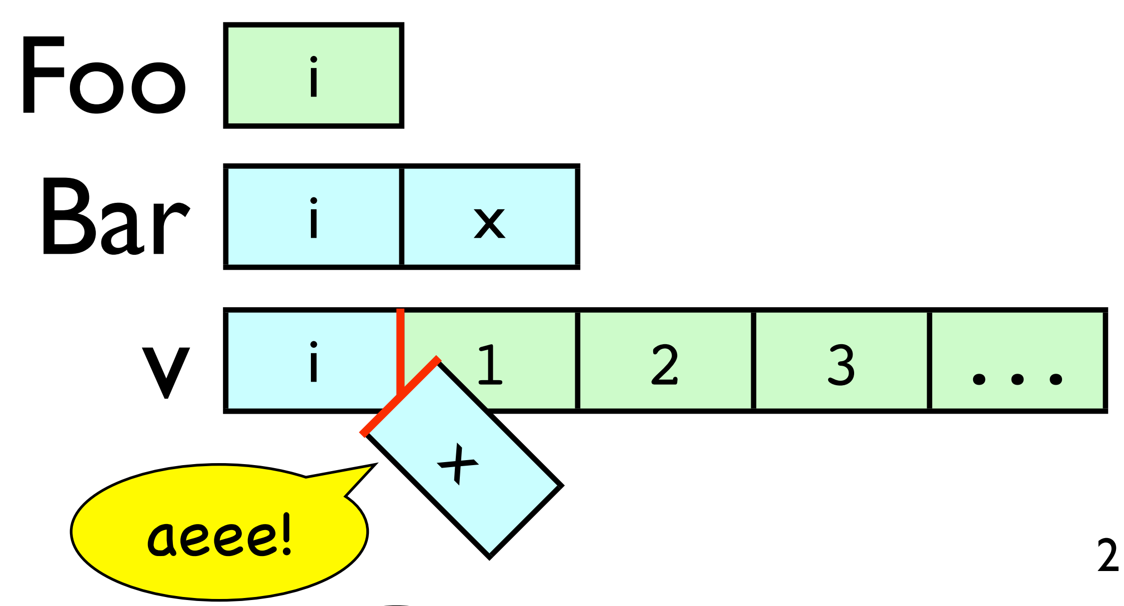
Gotchas: Slicing

- Slicing
  - Store **pointers** when inheritance may be possible
    - can’t store references, no way to reassign them
  - Say you have vector<Foo>. What happens when you insert a subclass with additional fields?

```
class Foo {
public:
  int i;
};
```

```
class Bar : public Foo {
public:
  int x;
};
```

```
int main (int argc, char * const argv[]) {
  vector<Foo> v; // vector of ‘Foo’s
  v.push_back(Bar()); // insert a ‘Bar’
}
```



Gotchas: Slicing

- Slicing
  - Store **pointers** when inheritance may be possible
    - can't store references, no way to reassign them
  - Say you have `vector<Foo>`. What happens when you insert a subclass with additional fields?
    - Answer: **Very bad things**; the additional fields are cut off -- only the '`Foo`' portion is stored (at best!)
    - This is called "slicing"
  - Use `vector<Foo*>` instead, now elements can be any subclass.

Gotchas: `char*` vs. string

- Remember an array is a pointer to the first element.
- C used an array of chars, terminated by '0' (aka '\0') as its string representation.
  - `strcpy()`, `strcat()`, `strcmp()`, …
  - Sometimes elegant, but sometimes inefficient, and error prone to boot
  - Better than its contemporaries: Pascal-style strings store the length in the first byte, limited to 256 characters.

Gotchas: `char*` vs. string

- C++ has a better idea: `string` class
  - souped-up `vector<char>`
- Can automatically create a `string` from a `char*`

```
void print(const string& s);
print("foo"); // works! (compiler implicitly calls string constructor)
```

- But have to explicitly request a `char*` from a `string`

```
void print(const char* s);
string s="foo";
print(s); //doesn't work
print(s.c_str()); //the solution: call c_str()
```

Scratching the Surface

- Further reading:
  - *The C++ Programming Language*, Bjarne Stroustrup

    The definitive, practical, and insightful reference from the language's creator.
  - *Effective C++ : 55 Specific Ways to Improve Your Programs and Designs*, Scott Meyers

    Don't learn the fine points the hard way, read this instead. (Others in his series also recommended)
  - An STL Reference (I don't have any particular favorite)

    Stroustrup's book is a good introduction to the STL highlights, but there's a lot to be expanded on.