C++ Introductory Tutorial

Part II

Institute of Computer Graphics and Algorithms

Vienna University of Technology

Today

- Operator Overloading
- Templates
- STL
- Short Recap
- Classes cont'd
  - Methods C++ silently writes and calls
  - Interfaces in C++
  - The 4 casts of C++
- Exceptions
- Shared Pointer

Last Session's Topics

- Stages of the C++ build process
- Basic syntax
- Declaration vs. Definition (Headers)
- Data types
- Pointer & References
- Important C++ operators
- Global Scope
- Const correctness
- Passing variables
- Stack & Heap Memory
- Classes & Polymorphism

C Plus Plus

- Developed by Bjarne Stroustrup
  - 1979 Bell Labs
  - Originally named *C with Classes*
- Powerful type-safe language
- Used in
  - Games
  - Embedded Systems
  - High-performance application
  - Drivers, Kernels,...

C Plus Plus

- C++ is a federation of 4 languages
  - C
    - You can still do any low level C stuff (comes in handy when using C APIs like OpenGL)
  - Object oriented C++
    - Classes, Polymorphism, OOP
  - Template C++
    - Generic programming, template metaprogramming
  - Standard Template Library (STL)
    - A set of standard algorithms and data structures for C++

Operator Overloading

```
class my_vector {

public:
  float x,y;

  // constructors
  my_vector(void) : x(0.0f), y(0.0f) {}
  my_vector(float nx, float ny) : x(nx), y(ny) {}
};

my_vector v1(3.0f, -4.0f);
my_vector v2(-6.0f, 5.0f);

// why not add v1 and v2?
v1 + v2; // looks good, but doesn't compile...
```

Operator Overloading - Version 1

```
class my_vector {
  // ...
  my_vector operator+(const my_vector &second) const {
    cout << "operator+ in class" << endl;
    return my_vector(this->x + second.x, this->y + second.y);
  }

};


my_vector v1(3.0f, -4.0f);
my_vector v2(-6.0f, 5.0f);

v1 + v2; // now it works!
v1.operator+(v2); // equivalent, but "ugly" ;-)
// NOTE: first operand MUST be of type "my_vector"
// i.e. we cannot overload "+" for e.g. "int + my_vector"
// only for "my_vector + int"
```

Operator Overloading - Version 2

```
class my_vector {
  // no "operator+() defined in the class
  // => friend fix, only needed for private-member access...
 friend my_vector operator+(const my_vector &first,
                            const my_vector &second);
};
// overload "+" outside class
// no access to private members (=> friend fix)
my_vector operator+(const my_vector &first,
                    const my_vector &second) {
    cout << "operator+ outside class" << endl;
    return my_vector(first.x + second.x, first.y + second.y);
}

v1 + v2;
v1.operator+(v2); // does NOT work anymore!
operator+(v1, v2); // works!
// NOTE: we could overload "+" for e.g. "int + my_vector", too!
```

Operator Overloading - Version 1 or 2 ?

- Overloading inside class
  - access to private class-data (no need for "friend")
  - 1st operator must be of class-type
- Overloading outside class
  - if access to private class members is needed, add "friend" to the method-declaration and put declaration inside the class
  - full control over parameters (at least one must be of class-type, but need not be the first parameter)

Operator Overloading - Printing

```
class my_vector {
  // overload "<<" for printing coordinates of my_vector
  // through "friend" we also have access to private elements
  friend std::ostream &operator<<(std::ostream &out,
                                  const my_vector &v) {
    out << "( " << (v.x) << " | " << (v.y) << " )";
    return out;
  }
};

my_vector v1(3.0f, -4.0f);

cout << v1 << endl; // prints "( 3 | -4 )"
```

Operator Overloading – Pre-/Postfix

```
class my_vector {

  // prefix ++
  my_vector &operator++(void) {
    ++x; ++y; // we could also have written "x++; y++;"
    return *this;
  }

  // postfix ++
  // return vector before incrementing its components!
  my_vector operator++(int dummy) {
    my_vector tmp(x,y);
    x++; y++; // we could also have written "++x; ++y;"
    return tmp;
  }

};
```

Strings

- C-strings are just null-terminated char-arrays
  - programmer has to take care of mem-allocation
  - at least several lib-functions like strcpy(), strcat(), strcmp(), strlen(), strtod(), etc.
- C++-strings are class-objects
  - many operators pre-defined
  - no explicit mem-allocation necessary
    - less error-prone

Strings

```
#include <string>
// some constructors
std::string cpp_str1("str1");
std::string cpp_str2("str2");
std::string cpp_str;

cpp_str = cpp_str1 + " " + cpp_str2;

cpp_str.append(" appended");

// =18
cout << "cpp_str.length() = " << cpp_str.length() << endl;

// prints "str1 str2 appended"
cout << "cpp_str = " << cpp_str << endl;
```

Strings – C++ to C

```
#include <stdio.h>
#include <string>

// C interface ... many OpenGL functions look similar!!!
void a_C_param_func(const char *c_str)
{
  printf("C: c_str = %s\n", c_str);
  std::cout << "C++: c_str = " << c_str << std::endl;
} //a_C_param_func()


// calling the function with a C++-string
std::string cpp_str("Let's (inter)face it, we all love C!");

a_C_param_func(cpp_str.c_str());
```

Templates

- C++ way of generic programming
  - generic ... type itself is a parameter
- Frees programmer from the need to copy-paste functions for different data types
  - C++ compiler generates needed code from template
    - ➔ compile-time polymorphism

Templates – Example (Motivation)

```
int my_max(int val1, int val2) {
  if(val1 >= val2)
    return val1;
  else
    return val2;
}

//we could add overladed versions of my_max() for other types:
/*
float my_max(float val1, float val2) {
  return (val1 >= val2) ? val1 : val2;
}

std::string my_max(std::string val1, std::string val2) {
return (val1 >= val2) ? val1 : val2;
}
*/
```

Templates - Example (Motivation)

```
int i_val1 = 4;
int i_val2 = 5;

// ==> int , int
// OK: prints 5, the maximum of 4 and 5
cout << my_max(i_val1, i_val2) << endl;
```

Templates - Example (Motivation)

```
float f_val1 = 4.5f;
float f_val2 = 5.5f;

// ==> float , float
// if the float-version of my_max is not defined,
// the maximum float value will be cast to an integer!
// at least the call will work due to implicit cast
// float-to-int...

// prints “5” instead of “5.5”
cout << my_max(f_val1, f_val2) << endl;
```

Templates - Example (Motivation)

```
std::string s_val1("four");
std::string s_val2("five");

// ==> string , string
// will fail to compile, if my_max() not defined
// for strings:
// no implicit cast string-to-int
cout << my_max(s_val1, s_val2) << endl;
```

Templates - Example

```
template<typename T> //alternatively: template<class T>
T my_generic_max(T val1, T val2)
{
  if(val1 >= val2)
    return val1;
  else
    return val2;
}

// works for (int,int), (float,float), (string,string)
// and many more out-of-the-box, as long as ">=" is
// defined for the data type
// but ambiguity e.g. in the (int,float)-case
```

Templates - Example

```
int i_val1 = 4;
float f_val2 = 5.5f;

// ==> int , float
// type ambiguity; will not compile ...
cout << my_generic_max(i_val1, f_val2) << endl;

// ... but can be solved ➔ explicit instantiation
cout << my_generic_max<int>(i_val1, f_val2) << endl; //=5
cout << my_generic_max<float>(i_val1, f_val2) << endl; //=5.5
```

Templates - Example

```
const char *s1 = "Vienna";
const char *s2 = "Baden";

cout << my_generic_max(s1, s2) << endl;

// prints "Baden" without a specialized function
// for handling C-strings; the problem is, that
// the strings' addresses are compared,
// not the strings themselves!
```

Templates - Example (Motivation)

```
template<> // no more template parameters left
const char *my_generic_max<const char *>(const char *val1,
                                          const char *val2) {
  cout << "template specialized for C-strings!" << endl;
  // compare strings, not pointers
  if(std::strcmp(val1, val2) >= 0)
    return val1;
  else
    return val2;
}
// with the additional template-function definition, we
// can handle the C-string compare case separately

cout << my_generic_max(s1, s2) << endl;
// prints "Vienna" as expected
```

STL

- STL ... Standard Template Library
- Software library that offers
  - containers
  - iterators
  - algorithms
  - functors
- Can be used with any built-in and user-defined type
  - sometimes certain operators must be defined for used types

STL – A Quick Overview

- STL containers store data
  - Some containers are
    - vector, list, queue, stack, map, pair
- STL iterators make it possible to step through the containers and randomly access elements
- STL algorithms perform common operations such as searching and sorting
- STL functors
  - functors are classes that overload the function operator *operator()*

STL - Vectors

- C++ blend of C-arrays
  - dynamic array
    - resizing (+memory management) done automatically
  - random access
  - inserting-/erasing possible
- Can still be passed to functions which expect "corresponding" C-arrays/C-pointers

STL - Vectors

```
#include <vector>

float curr_val = 0.0f;
std::vector<float> f_vec;

// both print "0"
cout << "f_vec.capacity() = " << f_vec.capacity() << endl;
cout << "f_vec.size() = " << f_vec.size() << endl;

for(int i=0; i<10; i++, curr_val+=0.1f)
    f_vec.push_back(curr_val);

// prints "13"
cout << "f_vec.capacity() = " << f_vec.capacity() << endl;
// prints "10"
cout << "f_vec.size() = " << f_vec.size() << endl;
```

STL - Vectors

```
#include <vector>

float curr_val = 0.0f;
std::vector<float> f_vec(10); // can initially store 10 floats

// both print "10"
cout << "f_vec.capacity() = " << f_vec.capacity() << endl;
cout << "f_vec.size() = " << f_vec.size() << endl;

for(int i=0; i<10; i++, curr_val+=0.1f)
    f_vec.at(i) = curr_val; // or f_vec[i] = curr_val;

// both print "10"
cout << "f_vec.capacity() = " << f_vec.capacity() << endl;
cout << "f_vec.size() = " << f_vec.size() << endl;
```

STL - Vectors

```
#include <vector>

std::vector<float> f_vec(10); // space for 10 floats

try {
  for(int i=0; i<100; i++, curr_val+=0.1f) {
    f_vec.at(i) = curr_val; // throws std::out_of_range
    f_vec[i] = curr_val; // same without idx checking!
  }
}
catch(std::out_of_range &e) {
  std::cerr << "oops, out of bounds!!!" << std::endl;
  std::cerr << e.what() << std::endl;
}
```

STL - Vectors

```
void print_vec(std::vector<std::string> &vec) {
  // print elements
  std::vector<std::string>::iterator it = vec.begin();
  for(; it != vec.end(); ++it) {
    std::cout << (*it) << std::endl;
  }
} //print_vec()

// NOTE: const_iterator!!!
void print_vec(const std::vector<std::string> &vec) {
  // print elements
  std::vector<std::string>::const_iterator it = vec.begin();
  for(; it != vec.end(); ++it) {
      std::cout << (*it) << std::endl;
  }
} //print_vec()
```

STL - Vectors (C - interface)

```
// C interface ... many OpenGL functions look similar!!!
void a_C_param_func(const unsigned int array_len,
                    const float *float_ptr) {

  for(int i=0; i<array_len; i++) {
   cout << "float_ptr[" << i << "]=" << float_ptr[i] << endl;
  }
} //a_C_param_func()

a_C_param_func(float_vec.size(),
               static_cast<float *>( &(float_vec[0]) ) );

// shorter, but less "clean"
a_C_param_func( float_vec.size(), &(float_vec[0]) );
```

STL - Pairs

- 2-tuple of data elements
  - 1st element called "first"
  - 2nd element called "second"
- Can be
  - assigned
  - copied
  - compared

STL - Maps

- Associative array
- Maps one data item (key) to another (value)
- Type of key must implement "<"
- Objects stored in array are of type **pair**

STL - Maps

```
#include <map>
// key(int), value(string); key ("int") must implement "<"
std::map<int, std::string> assoc_map;

assoc_map[3] = "three";
assoc_map[1] = "one";
assoc_map[4] = "four";

std::map<int, std::string>::iterator it = assoc_map.begin();

// prints items in map sorted by key (ascending)
for(; it != assoc_map.end(); ++it) {
  cout << "it->first = " << it->first << " , "; // key
  cout << "it->second = " << it->second << endl; // value
}
```

STL - Maps

```
#include <map>
// key(string), value(int)
std::map<std::string, int> assoc_map;

assoc_map["one"] = 1;
assoc_map["two"] = 2;
assoc_map["four"] = 4;

std::map<std::string, int>::iterator it = assoc_map.begin();

// prints items in map sorted by key (ascending)
for(; it != assoc_map.end(); ++it) {
  cout << "it->first = " << it->first << " , "; // key
  cout << "it->second = " << it->second << endl; // value
}
```

Miscellaneous - Common Pitfalls

```
int* a, b;
// NOTE: b is NOT pointer-to-int!
// a is pointer-to-int, b is int
// equivalent to:
//  int *a;
//  int b;

int tmp=23;
int& c=tmp, d=tmp;
// NOTE: d is NOT reference-to-int!
// c is reference-to-int, d is int
// equivalent to:
// int &c=tmp;
// int d=tmp;
```

Miscellaneous - Common Pitfalls

```
// this might (or might not) compile, depending
// on your compiler:
std::map<int, std::vector<int>> some_map;

// PROBLEM:
// the compiler might not parse the above ">>" correctly
// as two separate ">" symbols
// ==> don't forget to insert a whitespace between
// the two ">"
std::map<int, std::vector<int> > some_map;
```

Miscellaneous - Reading a File

```
#include <fstream> // among others...

string read_file(const string &filename) {

  std::ifstream ifile(filename.c_str());

  return string(std::istreambuf_iterator<char>(ifile),
                std::istreambuf_iterator<char>());
}
```

Miscellaneous – Writing to a File

```
#include <fstream> // among others...

void write_string_to_file(const string &file_name,
                          const string &str) {
  std::ofstream out_file;

  out_file.open(file_name.c_str());

  // write string "str" to file, just as if you'd
  // write to the command line!
  out_file << str << endl;

  out_file.close();
}
```

Miscellaneous – Converting to String

```
#include <sstream> // among others...

// conversion of type T to string
template<typename T>
std::string to_string(const T &val) {
  std::ostringstream oss;
  oss << val;
  return oss.str();
}

// usage:
int i=123;
std::string conversion_to_string = to_string(i);
```

Miscellaneous – Converting from String

```
#include <sstream> // among others...

// conversion from string to type T
template<typename T>
 T from_string(const std::string &str, T &t) {
  std::istringstream iss(str);
  iss >> t;
  return t;
}

// usage:
int num;
from_string("54321" , num);
// call-by-reference => no need to assign
// return value explicitly
```

Miscellaneous – Including Files

- Different syntax for the preprocessor directive #include
  - basic difference is the search strategy to find header files

Miscellaneous – Including Files

- <> used for "standard include files"
  - search for file starts in include directories (directory which is defined first is also searched first)

    ```
    #include <GL/glew.h>
    ```

  - for system header files, no suffix necessary

    ```
    #include <iostream>
    ```

    - might not even exist as actual files, are just "known"
    - for each C-standard-header `<X.h>` there is a C++-header `<cX>`

Miscellaneous – Including Files

- “ “ used for header files from the current project
  - search starts in the local directory, then searches the include directories (as with <>)

```
#include "in_current_dir.hpp"
```

Methods C++ silently writes

- An empty class

```
class Empty {

};
```

Methods C++ silently writes

- Is actually this:

```
class Empty {
  public:
    Empty() { ... } // default constructor
    Empty(const Empty& rhs) { ... } // copy constructor
    ~Empty() { ... } // default destructor
    //(non-virtual except a case class has virtual dtor
    // copy assignment operator
    Empty& operator=(const Empty& rhs) { ... }
};
```

Methods C++ silently writes

- Default Constructor
  - Takes no arguments
  - Is only provided if no other Constructors are declared by you
- Default Destructor
  - Is not virtual
- Default Copy Constructor and Copy Assignment Operator
  - Simply copy all data members over to the target object

Copy Constructor vs. Copy Assignment Op.

- They are called in two different situations:

```
DeepPerson somePerson;
//This will call DeepPerson::DeepPerson(const DeepPerson& rhs)
//Equivalent to DeepPerson copyConstructed(somePerson);
DeepPerson copyConstructed = somePerson;

DeepPerson copyAssigned;
//This will call
//   DeepPerson::DeepPerson& operator=(const DeepPerson& rhs);
copyAssigned = somePerson;
```

Example 1: ShallowPerson.h

```
class ShallowPerson {

	public:
		ShallowPerson(const string& name);
		~ShallowPerson();

		const string& getName() const;
		void setName(const string& name);
		//ShallowPerson is OWNER of child's data
		ShallowPerson * createChildWithName(const string&
name);
		const ShallowPerson * getChild() const;

		virtual string saySomething() const;

	private:

		string _name;
		ShallowPerson* _child;

};
```

Example 1 : ShallowPerson.cpp

```
ShallowPerson::ShallowPerson(const string& name)
        : _name(name), _child(NULL)
{
}

ShallowPerson::~ShallowPerson() {
        if (_child != NULL)
                delete _child;
}

ShallowPerson* ShallowPerson::createChildWithName(const string& name)
{
        if (_child == NULL) {
                _child = new ShallowPerson(name);
                return _child;
        } else {
                //Do appropriate error logging in here
                return NULL;
        }
}
```

Example1: ShallowPerson.cpp cont'd

```
string ShallowPerson::saySomething() const {

	std::ostringstream oss;

	oss << _name << " (@ " << this << ") who has";

	if (_child != NULL) {
		oss << " a child ";
		oss << _child->saySomething();
	} else {
		oss << " no child.";
	}

	return oss.str();
}
```

Example 1 : SilentlyWrittenStuff.cpp

- Consider the following:

```
ShallowPerson john("John");

//John is going to have a kid "Johnny"
//Note that he is supposed to be "the owner" of the child
//(in terms of memory)

//Johnny in turn, is going to have another kid "Johnny-Lee"
ShallowPerson* johnny = john.createChildWithName("Johnny");
johnny->createChildWithName("Johnny-Lee");

//Prints:
//John (@ 0x7fff5fbff1a0) who has a child
//           Johnny (@ 0x100100220) who has a child
//                  Johnny-Lee (@ 0x100100270) who has no child.
cout << john.saySomething() << endl;
```

Example 1 : SilentlyWrittenStuff.cpp cont'd

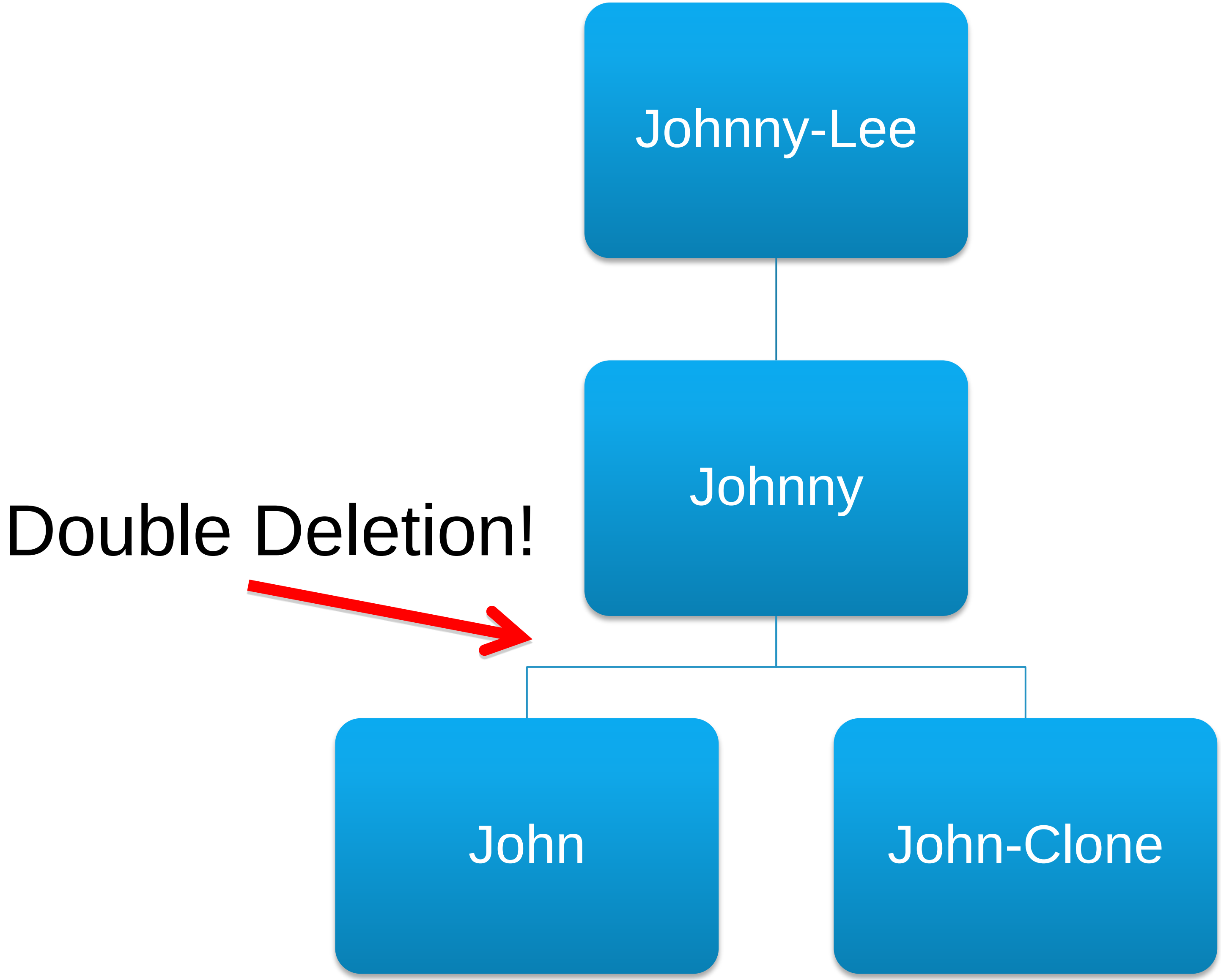
- This is going to cause troubles:

```
//Now what happens if we copy John?
ShallowPerson johnCopy = john;

//Prints:
//John-Clone (@ 0x7fff5fbff180) who has a child
//          Johnny (@ 0x100100220) who has a child
//                 Johnny-Lee (@ 0x100100270) who has no child.
cout << johnCopy.saySomething() << endl;
```

- When `johnCopy` and `john` gets destroyed this code will crash
  - Why?
  - → Double Deletion in `ShallowPerson::~ShallowPerson()`

Example 1 : ShallowPerson cloned hierarchy



Example 1 : DeepPerson.h

- Lets do it correctly and control the copying process more tightly by adding the appropriate declarations

```
class DeepPerson {
     public:

          ... //Constructors/Destructors omitted for brevity

          //Copy Constructor
          DeepPerson(const DeepPerson& rhs);
          //Copy Assignment Operator
          DeepPerson& operator=(const DeepPerson& rhs);

          ... //other methods and members same as in ShallowPerson.h
};
```

Example 1 : DeepPerson.cpp

```
DeepPerson::DeepPerson(const DeepPerson& rhs)
        : _name(rhs.getName()+"-clone"), _child(NULL)
{
        if (rhs.getChild() != NULL) {
                _child = new DeepPerson(*rhs.getChild());
        }
}

DeepPerson& DeepPerson::operator=(const DeepPerson& rhs) {

        if (this == &rhs) {
                return (*this);
        } else {
                _name = rhs.getName();
                //Same as in Copy Constructor
                if (rhs.getChild() != NULL) {
                        _child = new DeepPerson(*rhs.getChild());
                }
                return (*this);
        }
}
```

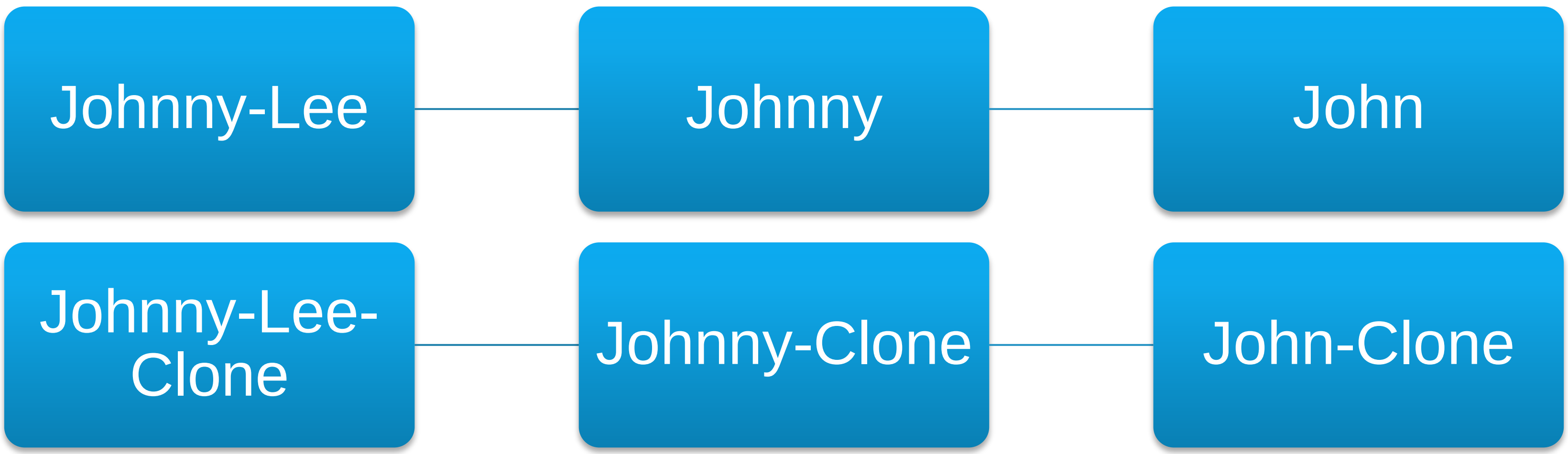
Example 1 : SilentlyWrittenStuff.cpp

- Now `DeepPerson` creates a deep (vs. shallow) copy of its child and we won't run into troubles:

```
DeepPerson johnCopy(john);

//Prints:
//John-clone (@ 0x7fff5fbff180) who has a child
// Johnny-clone (@ 0x1001002e0) who has a child
//    Johnny-Lee-clone (@ 0x100100350) who has no child.
cout << johnCopy.saySomething() << endl;
```

Example 1 : DeepPerson cloned hierarchy



Copy Constructor vs. Copy Assignment Op.

- They are called in two different situations:

```
DeepPerson somePerson;
//This will call DeepPerson::DeepPerson(const DeepPerson& rhs)
DeepPerson copyConstructed = somePerson;

DeepPerson copyAssigned;
//This will call
//   DeepPerson::DeepPerson& operator=(const DeepPerson& rhs);
copyAssigned = somePerson;
```

- Therefore you can't implement one in terms of the other
  - But you can provide e.g. a private init method to share code

Example 1 : AmnesicStudent.cpp

- What about inheritance?

```
AmnesicStudent::AmnesicStudent(const AmnesicStudent& rhs)
        : _matNumber(rhs.getMatNumber())
{
}

AmnesicStudent& AmnesicStudent::operator=(const AmnesicStudent& rhs)
{
        if (this == &rhs)
                return (*this);

        _matNumber = rhs.getMatNumber();

        //Who is going to copy the members of our base class
DeepPerson??
        return (*this);
}
```

Example 1 : SilentlyWrittenStuff.cpp

- Lets see how that works out:

```
AmnesicStudent aMike("Mike", 123456);

//Prints: "I am Mike, Student no. 123456".
cout << aMike.saySomething() << endl;

//Copy Mike to Mike-Clone
AmnesicStudent aMikeClone(aMike);

//Prints: "I am default-name, Student no. 123456".
cout << aMikeClone.saySomething() << endl;
```

Example 1 : How to fix AmnesicStudent.cpp

```
HealthyStudent::HealthyStudent(const HealthyStudent& rhs) :
        DeepPerson(rhs), _matNumber(rhs.getMatNumber())
{
}

HealthyStudent& HealthyStudent::operator=(const HealthyStudent& rhs)
{
        if (this == &rhs)
                return (*this);

        //We also have to call the copy operator of our base class
        //or else the name won't get copied properly
        DeepPerson::operator=(rhs);

        _matNumber = rhs.getMatNumber();

        return (*this);
}
```

Copy Constructor and Copy Assignment Op.

- Provide own implementation
  - If simply copying/assigning data members is not sufficient
  - Declare private to prevent unintended copying
  - Default implementations are actually not too bad, but be aware of them!
- If you DO provide own implementation
  - Don’t forget to modify after adding members
  - Don’t forget to call base classes’ Copy Constructor and CAOp (see Example code)

Pure Virtual Methods in C++

- Similar to abstract classes in Java
- Use them to declare interfaces you require subclasses to implement

```
class TalkativeInterface {

	public:
		virtual ~TalkativeInterface() {}
		virtual string saySomething() const = 0;
};

class Person : public TalkativeInterface {

	public:
		... //everything else omitted
		//Person has to implement this or it won’t 
compile
		virtual string saySomething() const;
	...
};
```

Exceptions

```
void someFunctionA() {
        //We can throw anything
        throw string("Flying message.");
}

...

try {

        someFunctionA();

//Always catch by reference
} catch (const string& s) {

        cout << "Caught a string " << s << endl;

}
```

Exceptions

- When using exceptions you have to code carefully
  - There is no **finally** in C++!!

```
void someFunctionC() {

        string* s = new string("SOME");

        //Oops, this will throw out_of_range
        //AND create a memory leak...
        char c = s->at(4);

        //Could use shared_ptr in here!

        delete s;

}
```

Exceptions

- Throw lists allows the runtime to restrict the thrown objects to certain types:

```
void someFunctionD() throw (std::out_of_range) {
      throw string("Flying message");
}
```

- BUT:
  - The compiler won't complain
  - During runtime above code will call unexpected()
    - Which is really bad!

Exceptions

- Use and reuse classes of `#include <stdexcept>`
  - You can catch exceptions polymorphically:

```
void someFunctionD() throw (std::out_of_range) {
        string s("SOME");
        s.at(4);
}

...

try {

        someFunctionD();

//out_of_range is subclass of exception
} catch (const std::logic_error& e) {
        cout << "Got an exception: " << e.what() << endl;
}
```

Exceptions - Summary

- Very different from using exceptions in Java
- You can throw anything
- You don’t have to catch
- Use throw lists to restrict thrown types
- Be aware of stack-unwinding after throw
- Reuse standard exceptions
- Keep the following in mind:
  - Constructor may throw
  - Destructor must not throw
  - Memory management is getting trickier
  - There are performance implications

Casting

- A perfect OO C++ world rarely needs casting
- Very often casting tries to fix bad design
- 4 C++ style casts:
  - **static_cast<T >()**
  - **dynamic_cast<T >()**
  - **reinterpret_cast<T >()**
  - **const_cast<T >()**
- And the C-style cast
  - **(T)**
  - ...which should be completely avoided

static_cast

- This is why static_cast is better than C-style cast:

```
int a = 7;
unsigned int b = static_cast<unsigned int >(a); // ok
double* p1 = (double*) &a;// ok (but a is not a double)
double* p2 = static_cast<double*>(&a); // error
```

- See http://www2.research.att.com/~bs/bs_faq2.html#static-cast

dynamic_cast

- Uses Runtime Type Information (RTTI) for polymorphic objects
- Useful for downcasting scenarios

```
void checkType(DeepPerson* p) {

    //Is this a healthy student?
    HealthyStudent* healthyStudentPtr =
               dynamic_cast<HealthyStudent* >(p);

    if (healthyStudentPtr != NULL) {
        cout << p->getName() << " is HealthyStudent." << endl;
    }
}
```

- Performance cost due to accessing RTTI

Shared Pointer

- Implements a very useful paradigm:
  - Resource Acquisition Is Initialization (RAII)
  - Object takes ownership

```
#include <tr1/memory>

typedef shared_ptr<string > StringRef;

...
//sRef automatically deletes the string when leaving scope
StringRef sRef(new string("String on heap."));

//Use StringRef just like a pointer to a string (string*)
cout << "The string '" << *sRef
     << "' is " << sRef->size() << " characters long" << endl;
```

Shared Pointer

- They play well together with exceptions
- This won’t cause memory leaks:

```
void someFunctionA() {

        //sRef is destroyed during unwinding of the
stack
        //Therefore the string on heap memory
        //is properly cleaned up
        StringRef sRef(new string("My String"));

        throw std::exception();

}
```

Shared Pointer

- They also work polymorphically
- And they can be stored in a STL containers

```
typedef shared_ptr<Person > PersonRef;
typedef shared_ptr<Student > StudentRef;

...

PersonRef pRef(new Student("Mike", 123456));

vector<PersonRef > people;
people.push_back(pRef);
//reuse objects
//This won't create memory leaks, as the original
shared_ptr is still in the vector!
pRef = PersonRef(new Person("John"));
people.push_back(pRef);
```

Shared Pointer

- Aka “smart pointer”, “auto pointer”
- Take ownership of data
- You can use them like normal pointers
- Enable much cleaner memory management
- Work well together with STL and exceptions

- Will be part of next C++ standard
- Also check out boost lib’s implementation:
  http://www.boost.org/doc/libs/1_42_0/libs/smart_ptr/smart_ptr.htm

The End

- Thanks for your attention!

Software Failure.   Press left mouse button to continue.

Guru Meditation #00000025.65045338