A Class Containing Objects

Implement a `Line` class that uses `Point` objects (slide 10) to represent the start and end points:

```
class Line {
public:
    Line(double x1, double y1, double x2, double y2);
    ...
private:
    ...
};
```

The following slides show alternative implementations. The variants with pointers are for illustration only, there is no reason to use dynamic memory in this class.

Class Line, Embedded Objects

Define the Point objects as member variables in the line class. Note *not* pointers to the objects:

```
class Line {
public:
    Line(double x1, double y1, double x2, double y2);
    ...
private:
    Point start;
    Point end;
};

Line::Line(double x1, double y1, double x2, double y2) :
    start(x1, y1), end(x2, y2) {}
```

- start(x1, y1) and end(x2, y2) are constructor calls which initialize the Point objects.
- No delete, no dynamic memory is involved.

Class Line, Dynamic Objects, C++98 version

Define "raw" pointers to the Point objects as member variables:

```
class Line {
public:
    Line(double x1, double y1, double x2, double y2);
    ...
private:
    Point* start;
    Point* end;
};

Line::Line(double x1, double y1, double x2, double y2) :
    start(new Point(x1, y1)), end(new Point(x2, y2)) {}
```

- Problem: the Point objects are never deleted, so there is a memory leak.

Destructors

The `Line` class needs a *destructor* which deletes the `Point` objects:

```
class Line {
public:
    ...
    ~Line();
private:
    Point* start;
    Point* end;
};

Line::~Line() {
    delete start;
    delete end;
}
```

- The destructor is called automatically when the object is destroyed and is responsible for cleaning up after the object.

Class Line, Dynamic Objects, Safe Pointers C++11

Define safe *pointers* to the Point objects as member variables:

```
class Line {
public:
    Line(double x1, double y1, double x2, double y2);
    ...
private:
    std::unique_ptr<Point> start;
    std::unique_ptr<Point> end;
};

Line::Line(double x1, double y1, double x2, double y2) :
    start(new Point(x1, y1)), end(new Point(x2, y2)) {}
```

- The dynamic objects are created in the constructor and handed over to the safe pointers.
- No visible delete, the safe pointers delete the objects when the pointers are destroyed (when the Line object is destroyed).

Inheritance

Inheritance in C++ is similar to Java inheritance, but member functions are by default not polymorphic. You must specify this with the keyword `virtual`.

```
class A {
public:
    void f();
    virtual void g();
};

class B : public A {
public:
    void f();
    virtual void g() override;
};
```

- `override` C++11 is not mandatory but recommended.

Calling Member Functions

```
void f() {
    A* pa = new A;
    pa->f(); // calls A::f
    pa->g(); // calls A::g

    A* pb = new B;
    pb->f(); // calls A::f, f is not virtual
    pb->g(); // calls B::g, g is virtual

    ...
    delete pa; // should have used unique_ptr ...
    delete pb;
}
```

- Non-virtual: the type of the *pointer* determines which function is called.
- Virtual: the type of the *object* determines which function is called.

Namespaces

Namespaces are used to control visibility of names (same purpose as Java packages). You can define your own namespaces. In a file *graphics.h:*

```
namespace Graphics {
    class Point { ... };
    class Line { ... };
}
```

In another file:

```
#include "graphics.h"

int main() {
    Graphics::Point p(1, 2);
    ...
}
```

Using Names from Namespaces

There are several ways to access names from namespaces (similar to importing from Java packages):

- Explicit naming:

```
Graphics::Point p(1, 2);
```

- `using` declarations:

```
using Graphics::Point;    // "import Graphics.Point"
Point p(1, 2);
```

- `using` directives:

```
using namespace Graphics; // "import Graphics.*"
Point p(1, 2);
```

Generic Programming

Java has generic classes:

```
ArrayList<Integer> v = new ArrayList<Integer>();
v.add(5);
int x = v.get(0);
```

C++ has generic (template) classes:

```
vector<int> v;
v.push_back(5);
int x = v[0];
```

- In Java, template parameters are always classes; may be of any type in C++. An `ArrayList` always holds objects, even if we use autoboxing and autounboxing as above.

Operator Overloading

In Java, operator + is overloaded for strings (means concatenation). In C++, you can define your own overloads of existing operators.

```
class Complex {
public:
    Complex(double re, double im);
    Complex operator+(const Complex& c) const;
private:
    double re;
    double im;
};

Complex Complex::operator+(const Complex& c) const {
    return Complex(re + c.re, im + c.im);
}
```

Now, you can add two complex numbers with `c1 + c2`.

General Program Structure

A main program uses a class `Stack`. The program consists of three compilation units in separate files:

```
class Stack {...};      // class definition, in file
                        // stack.h (header file)
Stack::Stack() { ... } // member function definitions,
                        // in file stack.cc
int main() { ... }      // main function, in file example.cc
```

It should be possible to compile the *.cc* files separately. They need access to the class definition in *stack.h*, and use an `include` directive for this:

```
#include "stack.h"      // own unit, note " "
#include <iostream>     // standard unit, note < >
```

stack.h

```
#ifndef STACK_H // include guard, necessary to avoid
#define STACK_H // inclusion more than once
#include <cstddef> // for size_t
class Stack {
public:
    Stack(std::size_t sz);
    ~Stack();
    void push(int x);
    ...
private:
    int* v; // pointer to dynamic array
    std::size_t size; // size of the array
    std::size_t top;  // stack pointer
};
#endif
```

- This file is only included, not compiled separately.
- Don't write `using namespace` in a header file. A file that includes the header file may not be prepared for this.

stack.cc

```
#include "stack.h" // for class Stack

using namespace std; // ok here, this file will not be included

Stack::Stack(size_t sz) : v(new int[sz]), size(sz), top(0) {}

Stack::~Stack() { delete[] v; }

void Stack::push(int x) { v[top++] = x; }

...
```

This file can be compiled separately:

```
g++ -c stack.cc      // generates object code in stack.o
```

example.cc

```
#include "stack.h"   // for class Stack
#include <iostream>  // for cin, cout, ...

using namespace std; // ok here, this file will not be included

int main() {
    Stack s(100);
    s.push(43);
    ...
}
```

Compilation, linking and execution:

```
g++ -c example.cc                  // generates example.o
g++ -o example example.o stack.o // generates example
./example
```

Data Types, Expressions, and Such

- Data types
- Strings, vectors, arrays
- Expressions
- Statements
- Functions

Primitive Data Types

Character `char`, usually one byte; `wchar_t`, `char16_t`, `char32_t` `C++11`, wide characters.

Integral `int`, often 32 bits; `short`, 16 or 32 bits; `long`, 32 or 64 bits; `long long` `C++11` at least 64 bits.

Real `double`, often 64 bits; `float`, 32 bits; `long double` `C++11` often 80 bits.

Boolean `bool`, `true` or `false`; also see next slide.

- Sizes are implementation defined.
- The operator `sizeof` gives the size of an object or a type in character units: `sizeof(char) == 1` always, `sizeof(int) == 4` typically.
- Integral types and characters may be `signed` or `unsigned`. Implementation defined if a plain `char` is signed or unsigned.

Booleans

In early C++ standards, `int`'s were used to express boolean values: the value zero was interpreted as `false`, a non-zero value as `true`. There are still implicit conversions between booleans and integers/pointers. The following is correct C++, but shouldn't be used in new programs:

```
int a;
cin >> a;
if (a) { ... }      // equivalent to if (a != 0)

Point* p = list.getFirstPoint();
while (p) { ... }  // equivalent to while (p != 0)
                   // 0 is C++98 for 'nullptr'
```

Variables and Constants

Rules for declaration of variables are as in Java:

```
int x;           // integer variable, undefined. The compiler
                 // isn't required to check the use of
                 // uninitialized variables
double y = 1.5; // double variable, initial value 1.5
```

Constant values:

```
const int NBR_CARDS = 52;
```

Variables may be defined outside of functions and classes; then they become globally accessible. Use with great care.

Enumerations

An enumeration is a type which defines a set of constant names. The new standard introduced "scoped enums" C++11, where the constant names are local to an enum (similar to constants in a class). Example:

```
enum class open_mode {INPUT, OUTPUT, APPEND};
```

An object of an enumeration type may only be assigned one of the enumerator values:

```
int open_file(const string& filename, open_mode mode) {
    ...
}

int file_handle = open_file("input.txt", open_mode::INPUT);
```

References

A reference is an alternative name for an object, an "alias". It is *not* the same as a Java reference (which is a pointer). References are mainly used as function parameters and as function return values.

A reference must be initialized when it's created, and once initialized it cannot be changed. There are no "null references".

```
int ival = 1024;     // a normal int
int& refval = ival; // refval refers to ival
refval += 2;         // adds 2 to ival
int ii = refval;     // ii becomes 1026
```

A `const` reference may refer to a `const` object, a temporary object or an object needing conversion. Not much use for this now, becomes important later.

```
const int& r1 = 42;
const double& r2 = 2 * 1.3;
const double& r3 = ival;
```

Pointers

A pointer points to a memory location and holds the address of that location. Unlike Java, a pointer may point to (almost) anything: a variable of a basic type like `int`, a stack-allocated variable, ... A pointer is declared with a `*` between the type and the variable name:

```
int* ip1;      // can also be written int *ip1
Point* p = nullptr; // pointer to object of class Point.
               // nullptr is "no object", same as Java's null.
               // nullptr is new in C++11, C++98 used 0
```

One way (not the most common; the most common is to use `new` to create a dynamic variable on the heap) of obtaining a pointer value is to precede a variable with the symbol &, the *address-of* operator:

```
int ival = 1024;
int* ip2 = &ival;
```

Dereferencing Pointers

The contents of the memory that a pointer points to is accessed with the * operator. This is called *dereferencing*.

```
int ival = 1024;
int* ip2 = &ival;
cout << *ip2 << endl; // prints 1024
*ip2 = 1025;
```

Anything can happen if you dereference a pointer which is undefined or null. Comparing pointers with references:

- Pointers must be dereferenced, not references
- Pointers can be undefined or null, not references
- Pointers can be changed to point to other objects, not references