An introduction to C++

C++ concepts

- C++ = C concepts + bigger library + classes + namespaces + some additional gear
- C concepts: syntax, data types, control structures, operators, pointer semantic etc.
- bigger library: strings, dynamic arrays, easier I/O

C++ additional gear

- explicit pass-by-reference
- more granular scoping
- operator overloading

C++ additional gear

```
int myfun(int a, int &b) {
    a++;    // a was passed by copy
    b++;    // b was passed by reference
}

int main() {
    int a = 0, b = 0;

    // now holds: a == 0, b == 0
    for (unsigned int i = 0 /* K.O. in C */; i < 10; i++) {
        myfun(a, b);  // mind: the call does not change anyhow
    }
    // now holds: a == 0, b == 10
}
```

C++ Input/Output

- streams (like `cin` & `cout`)
- << operator: write to
- >> operator: read from
- associativity makes << and >> "chains" possible

C++ Input/Output

```
// i am a C++ line-wide comment,
/* C-style block-wide comments are also ok */
#include <iostream>  // no more #include <stdio.h>

int main(int argc, char *argv[]) {
    unsigned int times;

    // no more printf()
    cout << "How many times? ";
    // no more scanf()
    cin >> times;

    for (unsigned int i = 0; i < times; i++) {
        // no more \n
        cout << "Hello world. (" << i << ")" << endl;
    }

    return 0;
}
```

C++ OOP

- C: pure functional
- Java: pure OOP (“everything is an object”)
- C++: mixed functional / OOP paradigm (“free” functions allowed)

C++' weak OOP

```
void freefunction() {
    // i am a function which does
    // not belong to any class
}

class aclass {              // i am a C++ class
    void classfunction() {
        // i am a C++ class method
    }
};

int main() {
    // i am the main function (j.a. "free" function)
}
```

C++ classes

- no class access specifiers (C++ has no "packages")
- methods & attributes can be `public` or `private` (or `protected`)
- constructors & destructors
- method overloading supported (see after)

```cpp
class Person {
    private:
        bool major;     // a private attribute

        void updatemajor() {    // a private method
            if (age >= 18) major = true;
            else major = false;
        }

    public:
        unsigned short age; // a public attribute

        // some public methods
        Person() {
            // constructor (overrides default empty constr)
            age = 0;
            major = false;
        }

        ~Person() { /* useless destructor (can be avoided) */ }

        void grow() { age++; updatemajor(); }

        void grow(unsigned short years) { age += years; }
};

int main() {
    Person a;
    a.grow();
    a.age++;
}
```

Mind

- access specifier "areas" (w.r.t. Java's per-item)
- definition of constructors & destructors
- how overloading is accomplished
- semicolon "`;`" after class definitions, no semicolon after function definitions
- object instantiation (more later)
- access to object methods and attributes

Inheritance

```
class Poly {
    public:
        float perimeter() { /* code here */ }
};

class Triangle: public Poly {
    public:
        float base() { /* code here */ }
        float height() { /* code here */ }
        // more triangle-specific methods here
};

int main() {
    Triangle tri;
    tri.perimeter();
    tri.height();
}
```

Interface methods

- `virtual` keyword enables implementing "interfaces" in Java style
- "virtual pure": virtual method without implementation
- classes sporting "virtual pure" methods = Java's abstract classes
- extend from such classes = implement an interface
- (avoid diamond problem w/ multiple inheritance)

Interface methods

```
class foo_interface {   // this could not be instantiated
    public:
        // the following are pure-virtual ("= 0")
        // pure virtual methods defer implementation to child classes
        virtual void mandatory_method_a(int itsarg) = 0;
        virtual bool mandatory_method_b() = 0;
};

class MyClass: public foo_interface {
    // must implement all virtual pure methods to become concrete
    void mandatory_method_a(int x) { /* "a" implementation */ }
    bool mandatory_method_b() { /* "b" implementation */ }
    // may have more custom methods then
};

int main() {
    // MyClass can instantiate objects: every method has an implementation
    MyClass inst;
    // I'm guaranteed inst offers its own _a and _b methods
}
```

More on C++ classes

- objects' `this` is a pointer
- the `const` key for methods which do not modify `this`
- multiple inheritance allowed (diamond problem present)

More on C++ classes

```
class Base {
    public:
        int a() { return 0; }
};

class FL: public Base {
};

class FR: public Base {
    public:
        int a() { return 1; }
        int myown() const {
            // I can not modify my object
         }
};

class Multiple: public FL, public FR {
};

int main() {
    Multiple x;
    x.a();  // error: a() call is ambiguous
}
```

Operator overloading

- customize how to treat expressions like `myobjc = myobja + myobjb`
- precious when working with classes
- define “magic” functions: `operator=`, `operator+`, `operator--`, `operator[]` et cetera

Operator overloading

```
typedef int coord_t;

class Point2D {
    protected:
        coord_t x;
        coord_t y;

    public:
        bool operator==(Point2D compare) const {
            return (x == compare.x && y == compare.y);
        }

        // also worths by reference
        Point2D operator+=(const Point2D &add) {
            this->x += add.x;
            this->y += add.y;
            return *this;
        }

        Point2D operator+(const Point2D &add) {
            Point2D result = *this;
            result += add;
            return result;
        }
};
```

The “Rule of three”

- If you need one of
  - custom destructor
  - custom copy-constructor
  - custom assignment operator `operator=`
- you’ll probably need all of them
- this is frequent when using pointers and dynamic memory inside the class

Hiding implementation

- split header (“.hpp”) and implementation (“.cpp”) files
- define classes in hpps, but do not implement methods
- define methods in cpps
- `#include` the hpp in the relative cpp

```
// the following goes in foo.hpp
class Foo {
    private:    // unfortunate: private stuff cannot be hidden
        int privar;
    public:
        Foo(); /* constructor */
        // these are not pure virtual: their implementation
        // is expected somewhere by the compiler
        int a();
        bool b(float x);
};


// the following goes in foo.cpp

 /* other include files here */
#include "foo.hpp"

Foo::Foo() { /* constructor implementation */ }

int Foo::a() { /* "a" implementation */ }

bool Foo::b(float x) { /* "b" implementation */ }
```

C++ templates

- generic classes or functions:
  - defined to operate on abstract types
  - instantiated/called with concrete types
- accomplished through static syntactical processing
- recently adopted by Java as “Java generics”

```
template<class T>
class MyContainer {
    private:
        T *holder;
        unsigned int top;
    public:
        MyContainer(unsigned int room);
        ~MyContainer();
        void put(T item);
        // more stuff
};

template<class T>
MyContainer<T>::MyContainer(unsigned int room) {
    holder = new T[room];
    top = 0;
}

template<class T>
MyContainer<T>::~MyContainer() {
    delete[] holder;
}

template<class T>
void MyContainer<T>::put(T item) {
    holder[top] = item;
    top++;
}

int main() {
    MyContainer<int> mc(5);
    mc.put(10); // OK
    mc.put("foo");       // rejected by the compiler
}
```

Data in C++

- primitive types have common C-style use, but they are objects nonetheless:
  `int *x = new int();`
  `float a(10.2);` etc
- pointers are handled the same way C does (dereference, addrof etc)
- objects usage feature some difference wrt Java

Objects in C++

- no garbage collector available
- objects can be instantiated statically much like primitive types
- static objects are deleted at their scope's end
- static class methods and object methods are invoked in different manners

Objects and dynamic memory

- objects differs from object pointers
- `this` is an object pointer
- `new` differs from `new[]` (objects vs arrays)
- `delete` differs from `delete[]`
- C's `malloc`, `calloc` etc can also be used when `new` cannot accomplish the task

```
class Foo { };

class Bar {
    private:
        float *dyfloat;

    public:
        Bar() { dyfloat = new float[100]; /* an array */ }
        ~Bar() { /* *dyfloat deletion necessary */ delete[] dyfloat; }
        void act() {
            Foo stfoo;  // stfoo is a statically allocated object
            this->dyfloat[0] = 123.4;   // this.dyfloat[0] is wrong
            return; // x is automatically destroyed here
        }
};

int main() {
    Bar *y; // object pointer: no actual Bar objects exist so far
    y = new Bar();  // an instance is constructed, y points to it
    y = new Bar();  // another instance appears, the older one still resides in
                    // memory but is no longer accessible (it is garbage)
    y->act();       // access via object pointers: objptr->member
    (*y).act();     // equivalent
    delete y;       // *y's destructor is called; the 2nd instance disappears

    Bar z;
    z.act();        // access via objects: obj.member
}
```

Exceptions

- usual `try` and `catch` keywords
- objects are thrown, not object pointers
- `throw` in function interface is optional
- (poor) standard exception library/hierarchy in `stdexcept`

```cpp
#include <stdexcept>
#include <iostream>

using namespace std; // look for names into the standard library

void fun(int a) throw (runtime_error, logic_error) {
    switch (a) {
        case 0:
            throw runtime_error("Description"); // mind: no *new*
            // this is never reached
        case 1:
            throw logic_error("Description");
    }
}

int main() {
    try {
        fun(0);
    } catch (runtime_error &ex) {
        cout << ex.what() << endl;
    } catch (logic_error &ex) {
        cout << ex.what() << endl;
    } catch (...) { // this catches every other kind
        return 1;
    }
    throw runtime_error();  // will cause the process to receive a SIGABRT
}
```

Namespaces

- containers for functions, classes & data types, structures etc
- avoid naming collisions
- useful for “wrapping up” stuff with a common feature

```cpp
#include <iostream> // defines its actors in the namespace "std"

namespace useless {    // define something under the namespace "useless"
    int fun() { return 0; }
    struct strct {
        int a;
        char b;
    };
    class Cl {
        public:
            void donothing() { }
    };
}

int main() {
    useless::fun();     // call function "fun()" from namespace "useless"

    struct useless::strct strinst;  // defines variable from strct structure
    strinst.a = 0;

    useless::Cl *clinst = new useless::Cl();
    clinst->donothing();

    std::cout << "Done nothing." << std::endl;

    using namespace std;        // from now on, "std" is the default namespace
    cout << "Now exiting..." << endl;
}
```

C++ standard library

- standard library (strings, exceptions etc) +
- STL (Standard Template Library) from SGI +
- the whole C standard library

STL

- Dynamic containers (lists, sets etc)
- Iterators
- Algorithms (searching, sorting etc)
- more

```
#include <iostream>     // defines cin, cout and the rest for I/O
#include <stdexcept>    // defines some exceptions
#include <vector>       // defines the Vector class
#include <string>       // defines the string class

using namespace std;

int main() {
    string name = "Some name";

    if (name.empty() || name.length() > 4)
        cin >> name;

    name = name + " more " + "pieces";

    vector<string> v;   // defines a vector of strings
    v.push_back(name);  // now v = ("Some name more pieces")
    name = "x";
    v.push_back(name);  // now v = ("Some name more pieces", "x")
    // vectors can be accesses like arrays (it overloads [])
    cout << "First: " << v[0] << endl << "Size: " << v.size() << endl;

    throw runtime_error("An error.");
}
```

Slide legenda

- words like `this` relate to code
- stars () relate to the relevance of the section w.r.t. systemc: low (1), medium (2) and hi (3 stars)