Contents

1 Topics

1.1 Hello world

1.2 Small differences

1.2.1 Headers

Standard headers do not have a ".h" name suffix.

```
#include <iostream>
```

All the C standard headers are available with a "c" prefix, and without the ".h" name suffix.

```
#include <cstdio>
```

1.2.2 Null pointers

Since C++11, C++ has a null pointer *value*

```
int * p = nullptr;
```

There is also a definition of a null pointer *type* in the standard library, defined in the *cstddef* header:

```
typedef decltype(nullptr) nullptr_t; // std::nullptr_t
```

1.2.3 Boolean type and values

C++ has a specific Boolean type

```
bool hungry = false;
bool happy = true;
```

1.3 References

A reference is an *alias* to another object.

```
int i = 1;
int & ref = i; // reference initialization
```

The code above declares `ref` as *an alias* of object `i`. Notice that the reference is not itself an object. In fact, in this case the compiler might implement the reference without allocating any memory, conceptually replacing every use of `ref` with `i`. In some cases, the compiler might need to store a reference, in which case the underlying implementation will consist of a pointer object.

Notice that there is a fundamental difference between the *initialization* and an *assignment* of a reference.

```
int i = 1;
int j = 2;
int & ref = i; // reference initialization
ref = j; // assignment.  Of what object?
j = j + 1;
std::cout << ref << std::endl;
```

Since a reference is an alias and not an object, every reference must be immediately initialized.

```
int & ref; // error: makes no sense
```

Also, again since a reference is not an object, it makes no sense to declare arrays of references, pointers to references, and references to references.

```
int i; // i is the only object here
int & ref = i; // ref is now an alias of i
int & * p; // error: makes no sense
int & A[10]; // error: makes no sense
int & & double_ref; // error: makes no sense
```

1.4 Namespaces

Namespaces support basic modularization

```
namespace usi {

namespace math_lib {

int sum(int a, int b) {
    return a + b;
}

} // namespace math_lib

} // namespace usi

int vector_sum(int * begin, int * end) {
    int s;
    for(s = 0; begin != end; ++begin)
s = usi::math_lib::sum(s, *begin); // qualified name for function sum()
    return s;
}
```

Namespaces are what they say they are: confined spaces for names to modularize and avoid conflicts.

```
namespace usi {

namespace math_lib {
```

```
int sum(int a, int b) {
    return a + b;
}

} // namespace math_lib

} // namespace usi

namespace mod_7 {

double sum(int a, int b) {
    return (a + b) % 7;
}

}

using mod_7::sum;

int main() {
    int i;
    int j;
    i = sum(1,2);
    j = usi::math_lib::sum(1,2);
}
```

The C programming language does not have namespaces, and therefore a similar modularization is achieved through name prefixes:

```
/* namespace "usi" { print(); set_name(); get_name() */
void usi_print();
void usi_set_name(const char * name);
const char * usi_get_name();
```

1.5 "using" declarations

```
using std;

using InputStream = std::istream;
// equivalent to the following C declaration:
```

```
typedef std::istream InputStream;

int main() {
    cout << "ciao!" << endl;
};
```

1.6 Exceptions

C++ provides *exceptions* as a powerful error-handling mechanism. You may throw any value.

```
#include <string>
#include <iostream>

const char * small_error = "okay";
const char * big_error = "not okay";
const char * nasty_error = "AAAHHhhhh!";

int f(int x) {
    if (x > 7)
return x - 2;

    switch(x) {
    case 0: throw 10;
    case 1: throw 20;
    case 2: throw small_error;
    case 3: throw big_error;
    default: throw nasty_error;
    }
}

int main() {
    int i;
    std::cout << "Input: ";
    std::cin >> i;
    try {
std::cout << "Output: " << f(i) << std::endl;
    } catch (int ex) {
std::cout << "error number " << ex << std::endl;
```

```
    } catch (const char * error_msg) {
std::cout << error_msg << std::endl;
    }
}

#include <string>
#include <iostream>

const char * small_error = "okay";
const char * big_error = "not okay";
const char * nasty_error = "AAAHHhhhh!";

int f(int x) throw (const char *) {
    if (x > 7)
return x - 2;

    switch(x) {
    case 0: throw 10;
    case 1: throw 20;
    case 2: throw small_error;
    case 3: throw big_error;
    default: throw nasty_error;
    }
}

int main() {
    int i;
    std::cout << "Input: ";
    std::cin >> i;
    try {
std::cout << "Output: " << f(i) << std::endl;
    } catch (int ex) {
std::cout << "error number " << ex << std::endl;
    } catch (const char * error_msg) {
std::cout << error_msg << std::endl;
    }
}
```

1.6.1 Example with "throw" specification

```
#include <string>
#include <iostream>
#include <exception>

const char * small_error = "okay";
const char * big_error = "not okay";
const char * nasty_error = "AAAHHhhhh!";

void abandon_ship() {
    std::cout << "Mayday mayday mayday!" << std::endl;
}

int f(int x) throw(int) {
    if (x > 7)
return x - 2;

    switch(x) {
    case 0: throw 10;
    case 1: throw 20;
    case 2: throw small_error;
    case 3: throw big_error;
    default: throw nasty_error;
    }
}

int main() {
    std::set_unexpected(abandon_ship);

    int i;
    std::cout << "Input: ";
    std::cin >> i;
    try {
std::cout << "Output: " << f(i) << std::endl;
    } catch (int ex) {
std::cout << "error number " << ex << std::endl;
    } catch (const char * error_msg) {
std::cout << error_msg << std::endl;
    }
```

```
}
```

1.6.2 Example with "noexcept" specification

```
#include <string>
#include <iostream>
#include <exception>

const char * small_error = "okay";
const char * big_error = "not okay";
const char * nasty_error = "AAAHHhhhh!";

void abandon_ship() {
    std::cout << "Mayday mayday mayday!" << std::endl;
}

void ciao() noexcept {
    std::cout << "ciao!" << std::endl;
}

void goodbye() {
    std::cout << "goodbye!" << std::endl;
}

int f(int x) throw(int) {
    if (x > 7)
return x - 2;

    switch(x) {
    case 0: throw 10;
    case 1: throw 20;
    case 2: throw small_error;
    case 3: throw big_error;
    default: throw nasty_error;
    }
}

int main() {
    std::set_unexpected(abandon_ship);
```

```
    int i;
    std::cout << "Input: ";
    std::cin >> i;
    try {
std::cout << "Output: " << f(i) << std::endl;
    } catch (int ex) {
std::cout << "error number " << ex << std::endl;
    } catch (const char * error_msg) {
std::cout << error_msg << std::endl;
    }
    try {
ciao(); // compiler can ignore the whole try-catch
    } catch (int ex) {
// because this will never happen
std::cout << "error number " << ex << std::endl;
    }
    try {
goodbye(); // compiler must deal with the try-catch blocks
    } catch (int ex) {
std::cout << "error number " << ex << std::endl;
    }
}
```

1.7 Classes

1.7.1 Example: a String class

```
#include <cstddef>

class String {
 public:
    String();
    String(const char *);
    String(const char * begin, const char * end);
    String(const char * begin, std::size_t len);
    String(const String &);

    ~String();

    std::size_t size() const;
```

```
    std::size_t append(const char *);

 private:
    char * data;
    std::size_t string_size;
    std::size_t allocated_size;
};
```

1.7.2 Constructors and initialization

A constructor has code, just like any other methods. A constructor may also specify a list of initialization expressions for the data members. For example:

```
#include <string>

class named_object {
    std::string name;

 public:
    named_object();
    named_object(const char * n);

    std::ostream & print(std::ostream & output);
};

named_object::named_object()
    : name("Nemo") {};

named_object::named_object(const char * n)
    : name(n) {};
```

Notice that the list defines initializations of the data members in terms of their specific constructors. In this case, `name` is of type std::string, which has a constructor that takes a `const char *` and that is invoked in both cases.

The direct base classes of a class can be considered data members, and therefore can also be initialized in the same way. For example:

```
class pixmap {
    // this section is, by default, private
```

```
public:
    pixmap(const char * n): name(n) {};

private:
    const char * name;
};

class ascii_pixmap : public pixmap { // class ascii_pixmap extends pixmap
public:
    ascii_pixmap(unsigned int width, unsigned int height);

private:
    unsigned int width;
    unsigned int height;
    char * data;
    char fg;
    char bg;
};

ascii_pixmap::ascii_pixmap(unsigned int w, unsigned int h)
    : pixmap("ascii"), width(w), height(h), fg('*'), bg(' ') {
    data = new char[width*height]; // WARNING: check allocation
    std::cout << "ascii_pixmap::ascii_pixmap() called." << std::endl;
}
```

1.7.3 Copy constructor

A copy constructor for a class `T` is a constructor that is declared to take a reference to an object or type `T` as parameter. More specifically `T &`, `const T &` (and other cases).

```
class vec2d {
public:
    double x;
    double y;

    vec2d() : x(0), y(0) {}; // "default" constructor
    vec2d(double a, double b) : x(a), y(b) {};  // other constructor

    vec2d(const vec2d & v) : x(v.x), y(v.y) {}; // copy constructor
```

```
};
```

A copy constructor is invoked every time you assign or copy an object, implicitly or explicitly:

```
int main() {
    vec2d v1;  // invokes default constructor
    vec2d v2(10, 3.2);  // invokes the second constructor
    vec2d v3 = v2;  // invokes the copy constructor
    vec2d v4(v3);  // also invokes the copy constructor
};

bool orthogonal(vec2d u, vec2d v) { // u and v are also copy-constructed
    return (u.x*v.x + u.y*v.y == 0);
}
```

1.7.4 Virtual methods

The following code exemplifies the syntax and semantics of virtual methods.

```
class A {
public:
    virtual int f();
    virtual int g(int x);

    int a;
};

int A::f() {
    std::cout << "A::f() called" << std::endl;
    return 0;
}

int A::g(int x) {
    std::cout << "A::g(" << x << ") called" << std::endl;
    return x + 1;
}

class B : public A {
public:
    virtual int f();
```

```
    int b;
};

int B::f() {
    std::cout << "B::f() called" << std::endl;
    return 10;
}

int main() {
    A * a;
    B * b = new B();

    a = b;

    std::cout << a->f() << std::endl;
    std::cout << b->g(10) << std::endl;
}
```

1.8 Operator overloading

1.9 Container library

One of the great advantages of C++ over C is the utility of the container library.

It is fair to say that the C standard library lacks even basic data structures and algorithms. On the contraty, the standard library of C++ is very rich and also versatile.

Let's look at some basic library features. The library covers strings.

```
#include <string>
#include <iostream>

int main() {
    std::string s;
    while (std::cin >> s) {
std::string::size_type pos = s.find("ciao");
if (pos == std::string::npos) {
    std::cout << s << " does not contains the string ‘ciao’." << std::endl;
} else {
    std::cout << s << " contains the string ‘ciao’ at position " << pos << std::endl;
```

```
}
    }
}
```

As it turns out, strings are a lot like a *container* of the container library of the C++ standard library. So, let's look at perhaps the most basic container, namely the `vector`:

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> A;
    int x;

    while (std::cin >> x)
A.push_back(x);

    std::cout << "I just read " << A.size() << " numbers:" << std::endl;
    for (std::vector<int>::iterator itr = A.begin(); itr != A.end(); ++i)
std::cout << *itr << std::endl;
}
```