FROM C TO C++

CRASH COURSE FOR C PROGRAMMERS

-PART I-

A BIT OF HISTORY AND CHARACHTERISTICS OF C++

HISTORY

- In the early 1970s, Dennis Ritchie introduced “C” at Bell Labs.
  - ✓http://cm.bell-labs.co/who/dmr/chist.html
- As a Bell Labs employee, **Bjarne Stroustrup** was exposed to and appreciated the strengths of C, but also appreciated the power and convenience of higher-level languages like Simula, which had language support for *object-oriented programming* (OOP).
  - ✓Originally called *C With Classes*, in 1983 it becomes C++
- In 1985, the first edition of The C++ Programming Language was released
- Standard in 1998 (ISO/IEC 14882:1998)

HISTORY

- Adding support for OOP turned out to be the right feature at the right time for the ‘90s. At a time when GUI programming was all the rage, OOP was the right paradigm, and C++ was the right implementation.
- At over 700 pages, the C++ standard demonstrated something about C++ that some critics had said about it for a while: C++ is a complicated beast.
- The first decade of the 21st century saw desktop PCs that were powerful enough that it didn’t seem worthwhile to deal with all this complexity when there were alternatives that offered OOP with less complexity.
  - ✓Java

HISTORY: JAVA OVER C++

- As a bytecode interpreted, rather than compiled, language, Java couldn't squeeze out all the performance that C++ could, but it did offer OOP, and the interpreted implementation was a powerful feature in some contexts
- Because Java was compiled to bytecode that could be run on a Java virtual machine, it was possible for Java applets to be downloaded and run in a web page
- Java's success led to an explosion of what are commonly called managed languages
- Common Language Interface, a Microsoft virtual machine, with implementations for Windows, Linux, and OS X, also supports a plethora of languages, as C++/CLI and C#/CLI

HISTORY: JAVA OVER C++

- Colleges soon discovered that managed languages were both easier to teach and easier to learn
  - ✓Because they don’t expose the full power of pointers directly to programmers, it is more restrictive, but it also avoids a number of nasty programming errors

THE BEAST WAKES UP AGAIN

- Performance has always been a primary driver in software development. The powerful desktop machines of the 2000s didn't signal a permanent change in our desire for performance; they were just a temporary blip.
- Modern mobile devices are very powerful computers in their own right, but they have a new concern for performance: *performance per watt*. For a battery-powered mobile device, there is no such thing as spare cycles.
- Cloud-based computers, that is, computers in racks of servers in some remote data center, are also powerful computers, but even there we are concerned about performance per watt. In this case, the concern isn't dead batteries, but power cost. Power to run the machines, and power to cool them.

STROUSTRUP

CHARACTERISTICS

- The most important feature of C++ is that it is both low- and high- level.
- Programming in C++ requires a discipline and attention to detail that may not be required of kinder, gentler languages that are not as focused on performance
  - ✓No garbage collector!

OPERATORS ARE THE SAME

Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	
		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`sizeof`	parameter pack	
		(*type*)	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	
16	Sequencing	`,`	comma separator	Left-to-right

TYPES ARE THE SAME

```
boolean a= true;
a= 6 > 5;
boolean b= false;

if(!b)
  std::cout << "Hello!\n";
```

DIFFERENCES

INPUT/OUTPUT

- Prefer the use of <iostream> for input/output operations
  - ✓ Header files in c++ have no ".h"

```
#include <iostream>

int main (int argc, char **argv) {

 int i; std::cout << "Please enter an integer value: ";
 std::cin >> i;
 std::cout << "The value you entered is " << i << std::endl;
 return 0;
}
```

NEW/DELETE

- The new and delete keywords are used to allocate and free memory
- They are "object-aware" so you'd better use them instead of malloc and free
  - ✓ In any case, never cross the streams (new/free or malloc/delete)

```
int *a = new int;
delete a;
int *b = new int[5];
delete [] b;
```

```
int ** p;
delete[] p;
delete p[1];
```

REFERENCES

- A reference allows to declare an alias to another variable. As long as the aliased variable lives, you can use indifferently the variable or the alias.

```
int x;
int& foo = x;
foo = 42;

std::cout << x << std::endl;
```

Foo has type “reference to int”

- References are extremely useful when used with function arguments since it saves the cost of copying parameters into the stack when calling the function (as pointers).

DIFFERENCES WITH POINTERS

- C++ references differ from pointers in several essential ways:
  - ✓It is not possible to refer directly to a reference object after it is defined; any occurrence of its name refers directly to the object it references.
  - ✓Once a reference is created, it cannot be later made to reference another object; it cannot be *reseated*. This is often done with pointers.
  - ✓References cannot be *null*, whereas pointers can; every reference refers to some object, although it may or may not be valid.
  - ✓References cannot be uninitialized. Because it is impossible to reinitialize a reference, they must be initialized as soon as they are created.

REFERENCES AS SYMBOLIC LINKS

- References can be thought of as "Symbolic links" in file system terminology
- Symbolic links can be modified as though they were the file they were linked
- Similarly, references can be deleted (whether by going out of scope or explicitly being removed if they were allocated in heap) and the original object that was referenced remains
- Similarly, once a symbolic link has been created it can never be changed

EXAMPLE 1

- Other than just a helpful replacement for pointers, one convenient application of references is in function parameter lists, where they allow passing of parameters used for output **with no explicit address-taking** by the caller.

```
void square(int x, int& result){
   result = x * x;
}
```

```
square(3, y);
```

This call places 9 in *y*

```
square(3, 6);
```

Compiler error

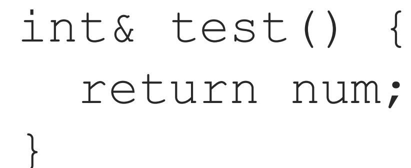
RETURN BY REFERENCE

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

int num= 1;
int& test();

int main() {
   test() = 5;
   cout << num;
   return 0;
}
```

5



```
int& test() {
   return num;
}
```

return num + 1;   NO!

NO!

```
int& test() {
   return 2;
}
```

NO!

```
int& test() {
   int n = 2;
   return n; }
```

WHEN TO USE REFERENCES

- *The C++ standard is very careful to avoid dictating how a compiler must implement references, but every C++ compiler implements references as pointers*
- int &ri = i;
  - ✓ *allocates the same amount of storage as a pointer, and places the address of i into that storage*
- As a general rule:
  - ✓ Use references in function parameters and return types to define useful and self-documenting interfaces
  - ✓ Use pointers to implement algorithms and data structures

DEFAULT PARAMETERS

- You can specify default values for function parameters. When the function is called with fewer parameters, default values are used

```
float foo( float a=0, float b=1, float c=2 ) {
  return a+b+c;
}

cout << foo(1) << endl << foo(1,2) << endl <<
foo(1,2,3) << endl;
```

- You should obtain values 4, 5 and 6

NAMESPACE

- Namespace allows to group classes, functions and variable under a common scope name that can be referenced elsewhere

```
namespace first { int var = 5; }
namespace second { int var = 3; }

cout << first::var << endl << second::var << endl;
```

- You should obtain values 3 and 5. There exists some standard namespace in the standard template library such as std.

WHY NAMESPACES

- Only one entity can exist with a particular name in a particular scope. This is seldom a problem for local names, since blocks tend to be relatively short, and names have particular purposes within them, such as naming a counter variable, an argument, etc...
- But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.
- Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

GLOBAL SCOPE

- Namespaces can be defined in global (or namaspace) environment

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

int main() {
    namespace C {
        var x= 6;
    }
    cout << "Print " << A::x;
    cout << " " << C::x << std::endl;
}
```

NAMESPACE TO AVOID COLLISIONS

- Where identifier is any valid identifier and named_entities is the set of variables, types and functions that are included within the namespace.

```
namespace identifier {
   named_entities
}
```

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

namespace foo {
  int value() { return 5; }
}

namespace bar {
  const double pi = 3.1416;
  double value() { return 2*pi;}
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```

USING

- The keyword **using** introduces a name into the current declarative region (such as a block).

```
#include <iostream>
using namespace std;
namespace first {
  int x = 5;
}
namespace second {
  double x = 3.1416;
}

int main () {
{ using namespace first;
  cout << x << '\n'; }
{ using namespace second;
  cout << x << '\n'; }
return 0;
}
```

5
3.1416

```
#include <iostream>
using namespace std;
namespace first {
int x = 5; int y = 10;
}
namespace second {
double x = 3.1416; double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

5
2.7183
10
3.1416

OVERLOADING

- Function overloading refers to the possibility of creating multiple functions with the same name as long as they have different parameters (type and/or number).

```
float add( float a, float b ){
  return a+b;
}

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

Use different names!
add_f_f e add_i_i

- It is not legal to overload a function based on the return type (but you can do it anyway)

```
int func();
string func();
int main() { func(); }
```

```
int func();
string func();
int main() { (string)func(); }
```

NESTING NAMESPACES

C++17 extension

- It is possible to nest namespaces

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

namespace A {
    int x= 8;
     namespace B {
         int x= 9;
    }
}

int main() {
   cout << "Print " << A::x;
   cout << " "  << A::B::x
        << std::endl
}
```

Print 8 9

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

namespace A {
    int x= 8;
     namespace B {
         int x= 9;
    }
}

namespace A::C {
    int x= 11;
}

int main() {
    cout << "Print " < A::x;
    cout << " "<< A::C::x
         << std::endl;
}
```

Print 8 11

ANONYMOUS NAMESPACE

- Same effect as internal linkage

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

namespace A {
    static int x= 8;
}


int main() {
     using namespace A;
    cout << "Print " << x << endl;
}
```

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

namespace  {
    int x= 8;
}


int main() {
    cout << "Print " << x << endl;
}
```

NAME COLLISION FOR NAMESPACES

Yes

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

namespace A {
    static int x= 8;
}

namespace A {
    static int y= 8;
}


int main() {
    using namespace A;
    cout << "Print " << x
        << endl;
}
```

No

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

namespace A {
    static int x= 8;
}

namespace A {
    static int x= 8;
}


int main() {
    using namespace A;
    cout << "Print " << x
        << endl;
}
```

DIFFERENCES FOR CONST

- const objects can be used as compile-time values in C++, not in C
- A name with file scope that is explicitly declared const, and not explicitly declared extern, has internal linkage, while in C it would have external linkage
- This feature allows the user to put const objects in header files that are included in many compilation units.

```
const int a = 50;
const int c = 100;
const int d = 100;
int endX = c + a;
int endY = d;
```

C No / C++ Yes

```
const int var_a = 1;
int var_b = 1;
```

Only var_b can be used in other files

EXCEPTIONS

- **Exception handling** is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional conditions requiring special processing – often changing the normal flow of program execution
  - ✓ In general, an exception breaks the normal flow of execution and executes a pre-registered *exception handler*
- In C **error checking** maintains normal program flow with later explicit checks for contingencies reported using special return values or some auxiliary global variable such as C's *errno*

```
int a= -1;

try {
  float *array = new float[a];
}
catch( std::bad_alloc e ) {
  std::cerr << "I caught: " << e.what() << std::endl;
}
```

I caught: std::bad_alloc

CREATING EXCEPTIONS

```
#include <stdexcept>

class MyException : public std::runtime_error {

  public: MyException() : std::runtime_error("Exception") { };

};
```

EXCEPTIONS

- The term *exception* is typically used in a specific sense to denote a data structure storing information about an exceptional condition
- One mechanism to transfer control, or *raise* an exception, is known as a *throw*
- From the point of view of the author of a routine, raising an exception is a useful way to signal that a routine could not execute normally - for example, when an input argument is invalid
- Lisp ’60s-’70s
- Table-driven approach: it creates static tables at compile time and link time that relate ranges of the program counter to the program state with respect to exception handling. Compiler + runtime system

OBJECT-ORIENTED PROGRAMMING

CLASSES

- A class can be considered as an extended concept of a data structure: instead of holding only data, it can hold both data and functions.
- An object is an instantiation of a class.
- *Access specifiers*: an *access specifier* is one of the following three keywords: *private*, *public* or *protected*.

```
class Foo {
  int attribute;
  int function( void ) { };
};

struct Bar {
  int attribute;
  int function( void ) { };
};
```

```
Foo foo; foo.attribute = 1; // WRONG
Bar bar; bar.attribute = 1; // OK
```

ACCESS SPECIFIERS

- **Private** members of a class are accessible only from within other members of the same class (or from their *"friends"*).
- **Protected** members are accessible from other members of the same class (or from their *"friends"*), but also from members of their derived classes.
- Finally, **public** members are accessible from anywhere where the object is visible.

EXAMPLE

```
class Rectangle {
  int width, height;

  public:
    void set_values (int x, int y) {
      width = x; height = y;
    }

    int area() {
      return width*height;
    }
};


int main () {
  Rectangle rect;
  rect.set_values (3,4);
  std::cout << "area: " << rect.area();
  return 0;
}
```

area: 12

CONSTRUCTOR

- What would happen in the previous example if we called the member function area before having called set_values?
  - ✓An undetermined result, since the members width and height had never been assigned a value.
- A class can include a special function called its *constructor*, which is automatically ***called whenever a new object of this class is created***, allowing the class to initialize member variables or allocate storage.
  - ✓Stack or heap (with new())
- This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void.

EXAMPLE

```
int main () {
  Rectangle* rect= new Recangle (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " <<  rect->area()
<< endl;
  cout << "rectb area: " << rectb.area()
<< endl;
  return 0;
}
```

```
class Rectangle {
  int width, height;

  public:
    Rectangle (int x, int y) {
      width = x;
      height = y;
    }

    void set_values (int x, int y) {
      width = x; height = y;
    }

    int area() {
      return width*height;
    }
};
```

rect area: 12
rectb area: 30

Conceptual error?

rect not deallocated

DESTRUCTOR

- Destructors fulfill the opposite functionality of *constructors*:
  - ✓they are responsible for the necessary cleanup needed by a class when its lifetime ends.
- The classes we have defined in previous chapters did not allocate any resource and thus did not really require any clean up.
- But if we allocate something in dynamic memory, we need to release it.
- A **destructor** is a member function very similar to a *default constructor*: it takes no arguments and returns nothing, not even void.
  - ✓It also uses the class name as its own name, but preceded with a tilde sign (~)

EXAMPLE

```
#include <iostream>
#include <string>
using namespace std;

class MyString {
  private:
    string* ptr;

  public:
    // constructors:
    MyString() : ptr(new string) {}
    MyString(const string& str) : ptr(new string(str)) {}
    // destructor:
    ~MyString () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
Example4 foo;
Example4 bar ("Example");
cout << "bar's content: " << bar.content() << '\n';
return 0;
}
```

SOME MORE DIFFERENCES

- In C, character literals such as 'a' have type int, and thus sizeof('a') is equal to sizeof(int)
- In C++, character literals have type char, and thus sizeof('a') is equal to sizeof(char)

- The comma operator in C always results in an r-value even if its right operand is an l-value, while in C++ the comma operator will result in an l-value if its right operand is an l-value

```
int i; int j;
(i, j) = 1; // Valid C++, invalid C
```

SOME MORE DIFFERENCES

- “Tentatives of definitions” do not exist anymore

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

int i;
int i= 3;

int main() {
    cout << "Print " << i << endl;
}
```

- C++ does not allow this. Only one definition of any given variable is allowed within a program

TO BE CONTINUED…E.G.,

- Overloading operators
- Keyword **this**
- **Static** and **const function** members
- Copy constructor
- Friend functions and classes
- Inheritance (multiple)
- Polymorphism

- Templates

HOW TO START PROGRAMMING IN C++

G++

c++ — -bash — 80×24

```
MacBook-Francesco:c++ francescosantini$ g++ --version
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-
gxx-include-dir=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.plat
form/Developer/SDKs/MacOSX10.12.sdk/usr/include/c++/4.2.1
Apple LLVM version 8.1.0 (clang-802.0.42)
Target: x86_64-apple-darwin16.5.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault
.xctoolchain/usr/bin
MacBook-Francesco:c++ francescosantini$
```

INSTALL G++

- Linux: sudo apt-get update
  - ✓ to update your package list with the most recent version of g++
- sudo apt-get install g++

- Mac: Starting with the OS X Mavericks (10.9) you will need to install clang and Xcode version 5 or later. Clang is a modern replacement for g++. It is backwards compatible and almost all the commands are the same.

- Packages for IDE, e.g., Eclipse
  - ✓ https://eclipse.org/downloads/packages/eclipse-ide-cc-developers/keplersr2

HELLO WORLD

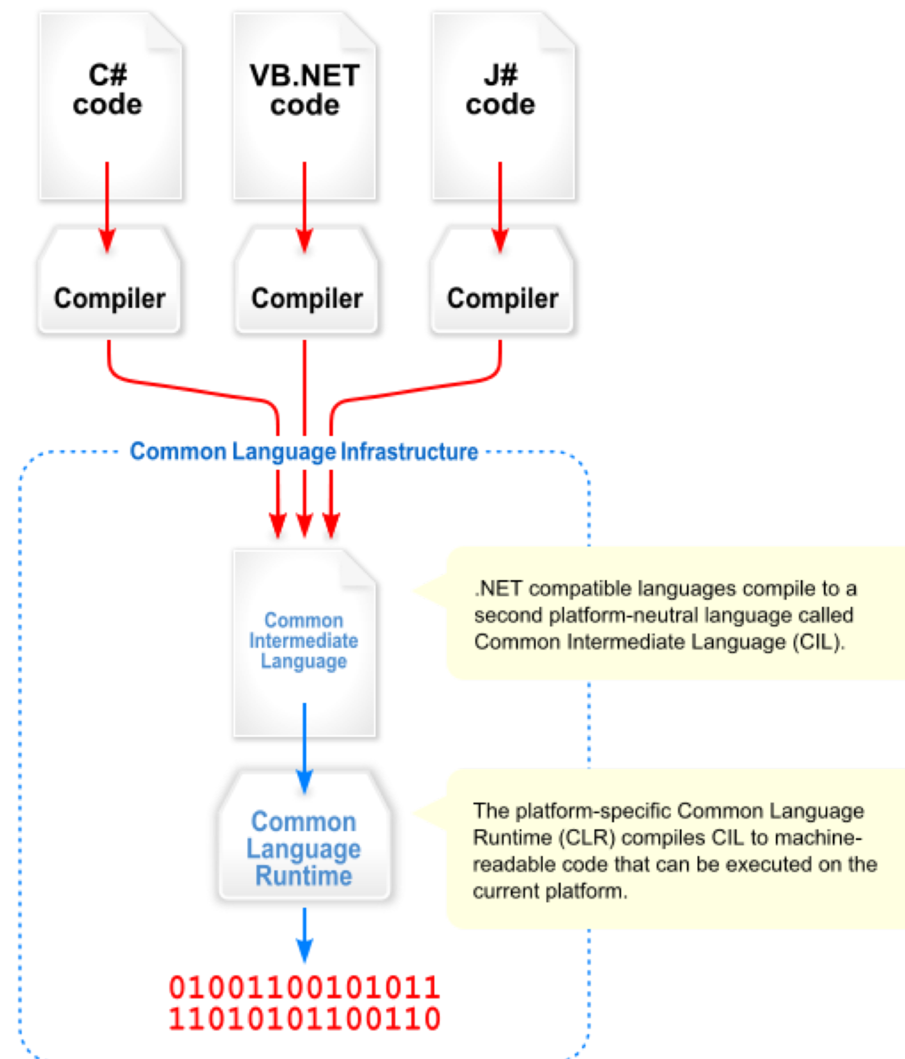
```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello, World!\n";
  return 0;
}
```

```
g++ -Wall -Wextra -Wpedantic -o hello hello.cpp
```

COMPILING C++ FOR CLI

- A lot of native C++ code will actually just compile and run on C++/CLI (Common Language Infrastructure)



COMPILING C CODE WITH G++

- If you take an existing C code base and compile it with a C++ compiler, what sort of issues can you expect to crop up?
  - ✓The main source of problems was that C++ is more strict about types

```
Foo *foo; foo = malloc(sizeof(*foo));
```
No

```
Foo *foo; foo = (Foo*)malloc(sizeof(*foo));
```
Yes

- Incompatibilities Between ISO C and ISO C++: http://david.tribble.com/text/cdiffs.htm#C++-vs-C

SO TO FINISH...

- Most of C code works if compiled with g++, but you could need to rewrite parts of it because of differences

```
#include <stdio.h>

int i;

int main(void) {
    printf("Hello world %d\n", i);

}
```

It compiles
(look at .h)