UNIVERSITÄT BERN

12. A bit of C++

Oscar Nierstrasz

Roadmap

> C++ vs C
> C++ vs Java
> References vs pointers
> C++ classes: Orthodox Canonical Form
> A quick look at STL — The Standard Template Library

Roadmap

> **C++ vs C**
> C++ vs Java
> References vs pointers
> C++ classes: Orthodox Canonical Form
> A quick look at STL — The Standard Template Library

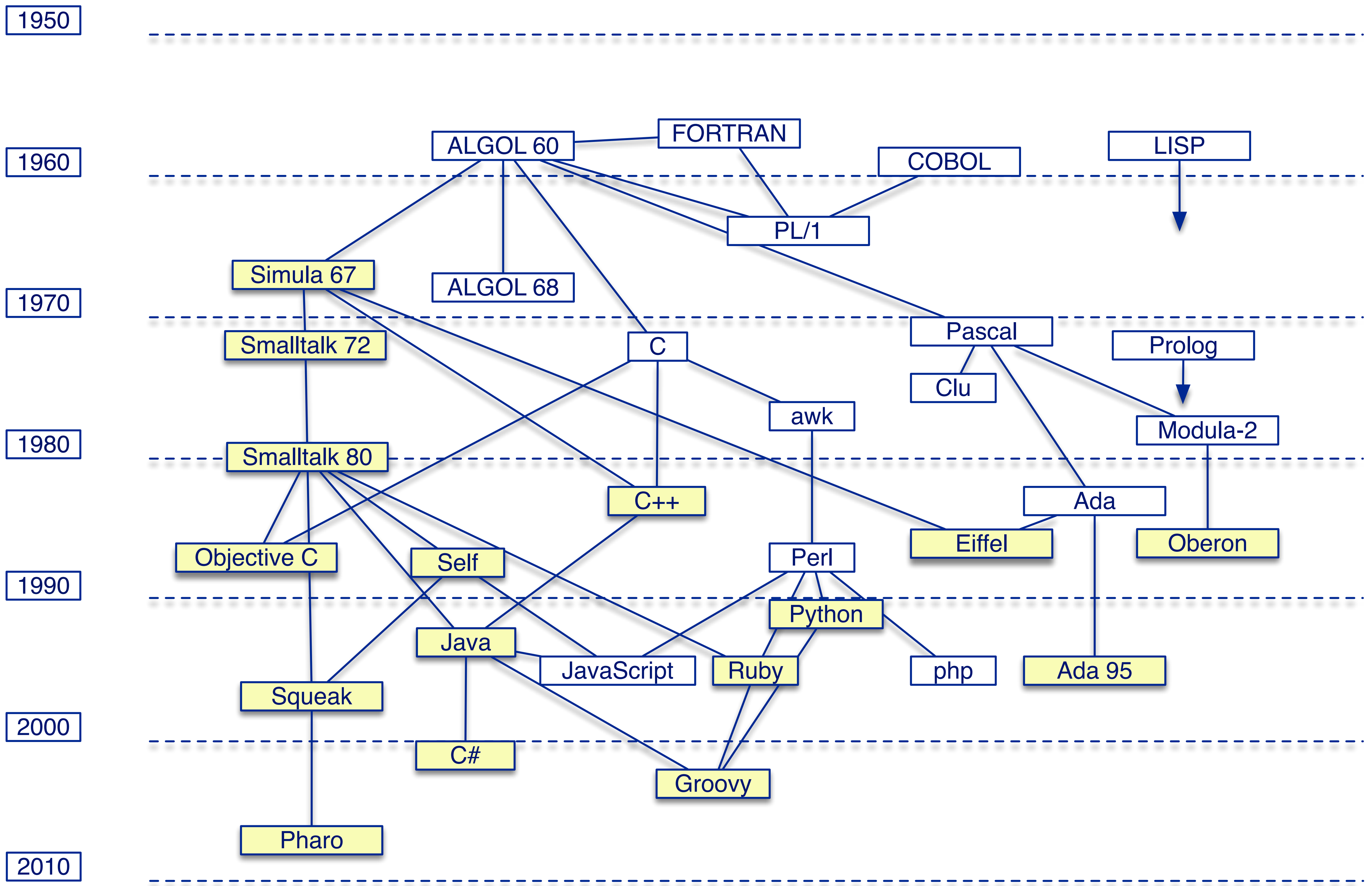
Selected C++ Texts

- Bjarne Stroustrup, *The C++ Programming Language*, 4th edition, Addison Wesley, 2013.
- Stanley B. Lippman and Josee LaJoie, *C++ Primer*, 5th edition, Addison-Wesley, 2012.
- Scott Meyers, *Effective Modern C++,* 2d ed., Addison-Wesley, 2014.
- Andrew Koenig and Barbara E. Moo, *Accelerated C++*, 2001
- David R. Musser, Gilmer J. Derge and Atul Saini, *STL Tutorial and Reference Guide*, 2d ed., Addison-Wesley, 2000.
- James O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

The book by Stroustrup is the reference manual but is not very suitable for learning the langauge. Lippman and LaJoie offers a good introduction and Meyers presents “best practices”.

In this lecture we will present the “orthodox canonical form” from Coplien's book.

Object-oriented language genealogy



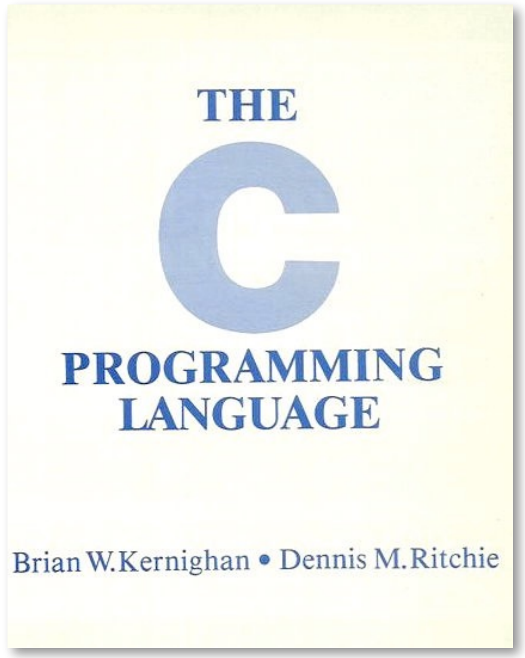
The original object-oriented language was Simula, designed by Ole-Johan Dahl and Kristen Nygaard in Oslo. They wanted a language to write simulation programs, so they added the notions of objects, classes and inheritance to Algol 60. As these things go, Simula programmers discovered that these mechanisms were useful for more than just simulation programs, and the notion of object-oriented programming as a general paradigm was born.

Years later, Bjarne Stroustrup needed to write simulations at Bell Labs. Since he didn't have a Simula compiler, he decided to follow the path of Dahl and Nygaard, and he added objects, classes and inheritance to C. The rest is history.

What is C?

> C is a general-purpose, procedural, imperative language developed in 1972 by Dennis Ritchie at Bell Labs for the Unix Operating System.

- —Low-level access to memory
- —Language constructs close to machine instructions
- —Used as a *"machine-independent assembler"*



C is an imperative language with constructs that can easily be mapped to assembler code. The language is a simplified descendant of BCPL, another language from the 60s.

C was designed as a systems programming language, particularly for implementing the Unix operating system, itself a simplification of the Multics O/S.

My first C Program

Include standard io declarations

A preprocessor directive

```
#include <stdio.h>

int main(void)
{
     printf("hello, world\n");
     return 0;
}
```

Write to standard output

char array

Indicate correct termination

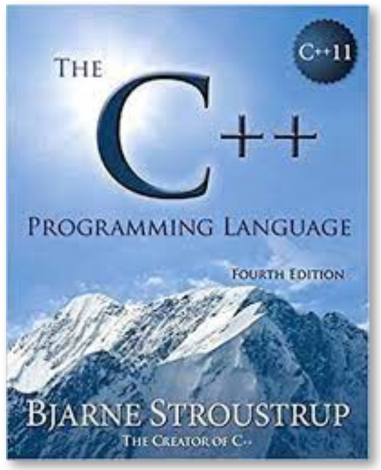
The C preprocessor (cpp) *expands macros and included header files*. Header files typically declarations of functions in object libraries that allow them to be type-checked and safely used. Here we need the declaration of the standard printf function from the stdio.h (standard I/O) header file.

C programs always start from a `main` function that may take arguments (here not) and returns an integer representing the program status (0 for success). The program prints `"hello, world\n"`, an array of characters terminated by an ascii 0 (for a total of 14 characters).

What is C++?

A *"better C"* (http://www.research.att.com/~bs/C++.html) that supports:

> Systems programming
> Object-oriented programming (*classes* & *inheritance*)
> Programming-in-the-large (*namespaces*, *exceptions*)
> Generic programming (*templates*)
> Reuse (large class & template libraries)



C++ is a general-purpose, high-level programming language with low level facilities. In the late 80s C++ became a very popular commercial language. It supports many programming styles — procedural, data abstraction, OO and template programming.

A namespace is a context in which a group of one or more identifiers might exist.

```
namespace foo {
    int bar;
}
using namespace foo;
```

Namespace resolution is hierarchical.

In Java the idea of a namespace is embodied by Java packages.

e.g., java.lang.String. NB: Namespaces are not hierarchical in Java.

C++ vs C

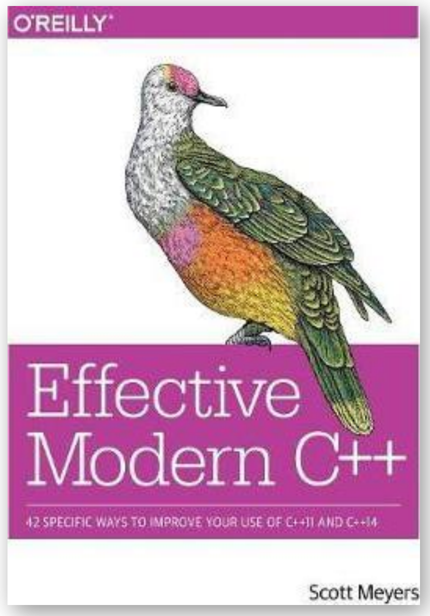
Most C programs are also C++ programs.

Nevertheless, good C++ programs usually do not resemble C:

> avoid macros (use `inline`)
> avoid pointers (use references)
> avoid `malloc` and `free` (use `new` and `delete`)
> avoid arrays and `char*` (use `vectors` and `strings`) ...
> avoid `structs` (use classes)

C++ encourages a different style of programming:

> avoid procedural programming
  — *model your domain* with classes and templates



Macros in C are handled by the C preprocessor. In C++ you should use constant declarations and inline functions instead.

The use of pointers in C is a source of many bugs. In C++ there is a safer mechanism called a "reference" (actually an alias), which we will see in this lecture.

Memory blocks are allocated and released in C using the malloc and free functions. Instead in C++ you allocate objects, using new and delete.

Low-level arrays and character pointers from C are replaced by dedicated classes in C.

Primitive data structures (structs) are replaced by classes in C++.

Roadmap

> C++ vs C
> **C++ vs Java**
> References vs pointers
> C++ classes: Orthodox Canonical Form
> A quick look at STL — The Standard Template Library

Hello World in Java

```
package p2;
// My first Java program!
public class HelloMain {
    public static void main(String[] args) {
        System.out.println("hello world!");
        return 0;
   }
}
```

The Java version of the “hello world” program looks a bit like the C version, except that the main program is a static method of a class, and the print function is a method of the object `System.out`.

“Hello World” in C++

Include standard iostream classes

Use the standard namespace

A C++ comment

```
#include <iostream>
using namespace std;
// My first C++ program!
int main(void)
{
  cout << "hello world!" << endl;
  return 0;
}
```

cout is an instance of ostream

operator overloading
(two *different* argument types!)

As in the C version, we have to include the appropriate header file declaring the libraries we plan to use, in this case, the iostream library. (Note that we do not need to specify the ".h" suffix.) We also specify that we will use (import) the *standard namespace* (alternative, we can refer to `cout` and `endl` as `std::cout` and `std::endl`).

We know C++ style comments from Java.

The object `cout` is declared in the standard namespace (similar to Java's System.out), as is `endl`.

In C++ we can overload operators (in Java you can only overload methods). Here we use << which is define to work with ostream objects and strings.

Note that there are *two different* << operators used here: one takes an ostream and string as arguments, and the other takes an ostream and an `endl` object as its arguments.

Makefiles / Managed Make in CDT or CLion

You could compile it all together by hand:

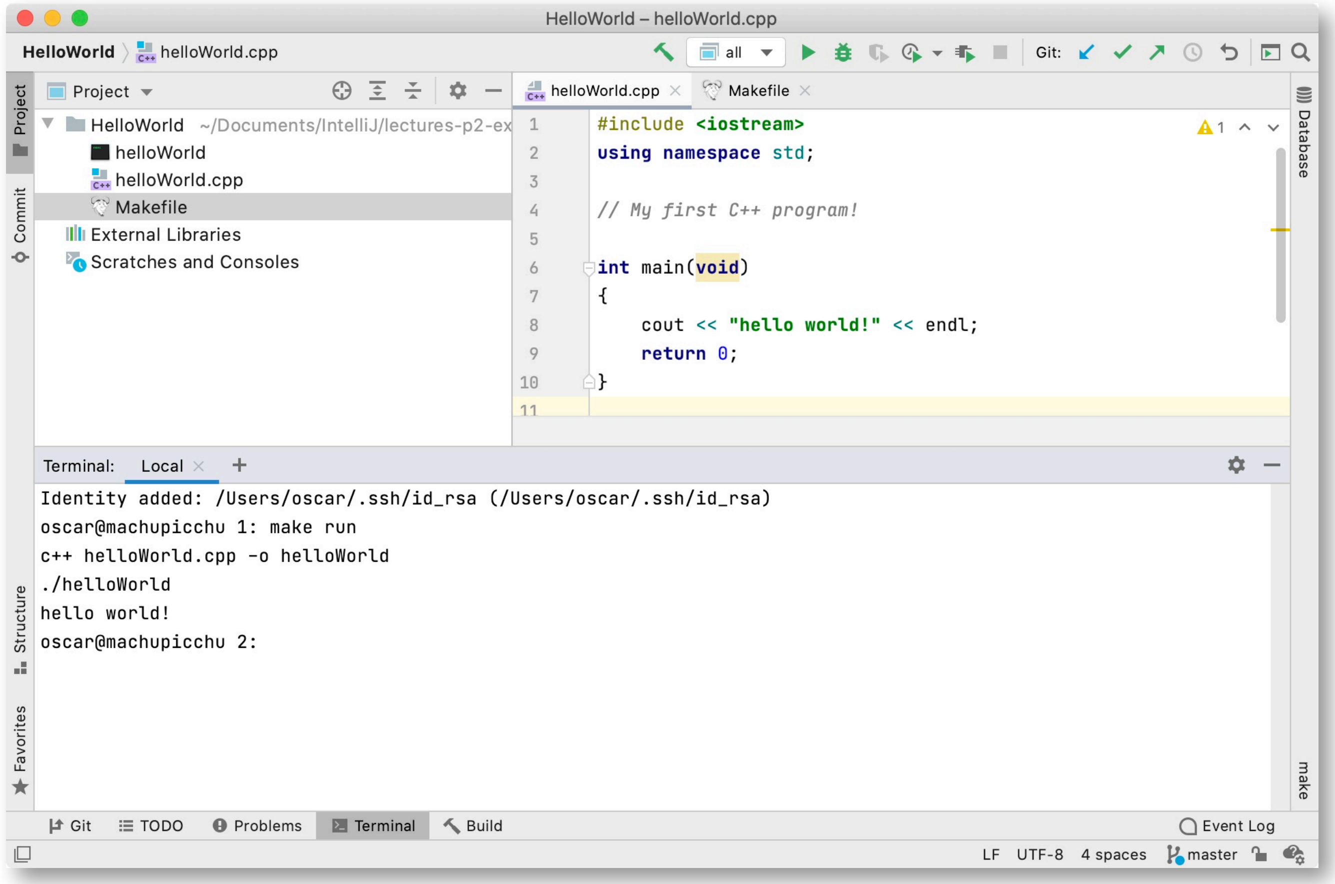
```
c++ helloWorld.cpp -o helloWorld
```

Or you could use a *Makefile* to manage dependencies:

```
helloWorld : helloWorld.cpp
	c++ $@.cpp -o $@
```

```
make helloWorld
```

Or you could use *cdt with eclipse* or *CLion* to create a standard managed make project



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

// My first C++ program!

int main(void)
{
    cout << "hello world!" << endl;
    return 0;
}
```

```
Identity added: /Users/oscar/.ssh/id_rsa (/Users/oscar/.ssh/id_rsa)
oscar@machupicchu 1: make run
c++ helloWorld.cpp -o helloWorld
./helloWorld
hello world!
oscar@machupicchu 2:
```

The C++ compiler can be known under various names.

A *makefile* specifies a number of rules for creating generated files that depend on other files. In a complex system, there may be many dependencies. When you change one file, you want to avoid having to recompile everything. The make system uses the makefile dependencies to compute the minimum number of files that have to be regenerated.

Here we specify that the *helloWorld* executable depends on the helloWorld.cpp source files. The rule to generate it is

```
 c++ $@.cpp -o $@
```

where `$@` stands for the name of the file to be created, in other words:

```
 c++ helloWorld.cpp -o helloWorld
```

To generate the program we just execute: `make helloWorld`

C++ Design Goals

"C with Classes" designed by Bjarne Stroustrup in early 1980s:

> Originally a translator to C
  —Initially difficult to debug and inefficient

> Mostly *upward compatible* extension of C
  —"As close to C as possible, but no closer"
  —Stronger type-checking
  —Support for object-oriented programming

> Run-time efficiency
  —Language primitives close to machine instructions
  —*Minimal cost for new features*

In the 1980s, computers were much slower, and it was critical that both compilation time and the execution time of the compiled programs be as efficient as possible. Since object-oriented programming introduced addition levels of indirection over procedural programming (i.e., to look up methods in classes), the original design of C++ carefully limited the points in the code where additional costs might be incurred. Programmers therefore have full control over the costs they are willing to pay for language features.

C++ Evolution

C with Classes	**Classes** as structs; **inheritance**; virtual functions; inline functions
C++ 1.0 (1985)	**Improved type system**; **`new` and `delete` operators**; function prototypes
C++ 2.0 (1989)	**Multiple inheritance**; local classes; protected members
C++ 3.0 (1993)	**Templates**; **exception handling**
C++98 (1998)	**Namespaces**; **RTTI** (Runtime Type Information)
C++11 (2011)	Standard library and core language enhancements; lambdas
C++14 (2014)	Small extensions
C++17 (2017)	Numerous new and removed language tweaks

The original version of C++, known as “C with Classes”, was just a translator that generated C code. It supported classes, inheritance, inline functions and “virtual functions” (i.e. methods that would be looked up dynamically based on the receiver).

One of the most important contributions of C++ was the improvement to the type system. This later influenced the design of the C language.

Multiple inheritance is a controversial feature supported only by some OO languages (but noot by Java or Smalltalk).

Templates were an important addition to C++, supporting parameterized code. In Java this is supported (partially) by generics.

Namespaces were added relatively late in the process, to support large-scale programming in which name clashes between independently developed subsystems must be avoided.

RTTI allows programs to obtain information at run-time about the types of objects in a running system.

Java and C++ — Similarities and Extensions

Similarities:

> primitive data types (in Java, platform independent)
> syntax: control structures, exceptions ...
> classes, visibility declarations (public, private)
> multiple constructors, this, new
> types, type casting (safe in Java, not in C++)
> comments

Key Java Extensions:

> **garbage collection**
> **standard abstract machine**
> **generics** instead of templates
> standard classes (came later to C++)
> packages (now C++ has namespaces)
> final classes
> autoboxing

Java’s syntax was designed to resemble that of C++, to make it easy for C++ programmers to learn the language. In fact, the resemblance between the languages is mostly superficial. C++ remains a systems programming language (like C) for implementing platforms, and Java is an applications programming language with high portability across platforms.

Java Simplifications of C++

> **no multiple inheritance** — implement multiple interfaces
> **no pointers** — just references
> **no destructors** — garbage collection and `finalize`
> no functions — can declare `static` methods
> no global variables — use `public static` variables
> no linking — dynamic class loading
> no header files — can define `interface`
> no operator overloading — only method overloading
> no member initialization lists — call `super` constructor
> no preprocessor — `static final` constants and automatic inlining
> no structs, unions — typically not needed

New Keywords

In addition to the keywords inherited from C, C++ adds:

Exceptions	`catch, throw, try`
Declarations:	`bool, class, enum, explicit, export, friend, inline, mutable, namespace, operator, private, protected, public, template, typename, using, virtual, volatile, wchar_t`
Expressions:	`and, and_eq, bitand, bitor, compl, const_cast, delete, dynamic_cast, false, new, not, not_eq, or, or_eq, reinterpret_cast, static_cast, this, true, typeid, xor, xor_eq`

***(see** http://www.glenmccl.com/glos.htm)*

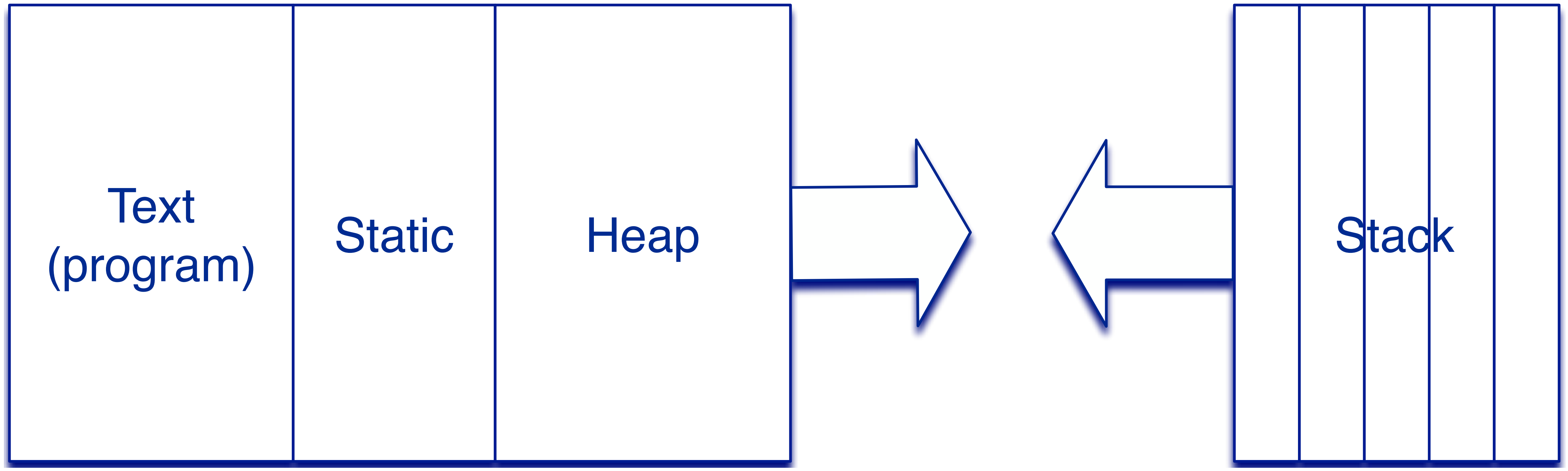
Roadmap

- C++ vs C
- C++ vs Java
- **References vs pointers**
- C++ classes: Orthodox Canonical Form
- A quick look at STL — The Standard Template Library

Memory Layout

The address space consists of (at least):

Text:	executable program text (not writable)
Static:	static data
Heap:	dynamically allocated global memory (grows upward)
Stack:	local memory for function calls (grows downward)



In a classical 32-bit architecture, a 32-bit C or C++ pointer can address up to 4GB of memory. Modern architectures support 64 bit pointers, which overcome the 4GB limitation.

The actual memory used by a program is far less than that which could theoretically be addressed. Normally statically and dynamically allocated heap space start with low addresses (i.e., starting from 0) and increase on need. The run-time stack starts at high addresses (2^{64} - 1) and decreases.

Low address starts with program “text” (i.e., space for the program itself), then *static* memory for global data following the program text. Finally as the program runs, *heap space* is allocated dynamically through calls to `new` (or `malloc`).

Pointers in C and C++

```
int i;
int *iPtr; // a pointer to an integer

iPtr = &i; // iPtr contains the address of I
*iPtr = 100;
```

variable	value	Address in hex
	...	
i	100	456FD4
iPtr	456FD4	456FD0
	...	

A *pointer* holds the address of an object in memory.

Here `iPtr` is a variable holding the address of an integer object in memory. The size of a pointer is *always the same*, depending on the machine architecture (on modern computers normally 64 bits). It does not depend on the object pointed to.

In C and C++, you use the `&` operator to take the address of an object. The address can be assigned to a pointer variable.

To dereference a pointer, you use the `*` operator.

Note that `*` is used here both in the *definition* of `iPtr` (its type is `int *`), and to dereference the pointer (`*iPtr`).

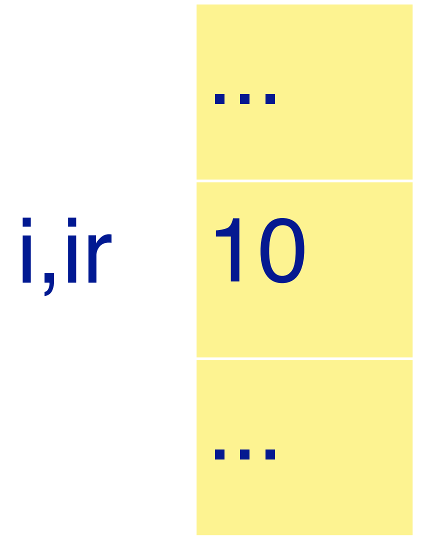
Note also that `*iPtr` is used here as an *lvalue*, i.e., a value that can be assigned to on the left-hand side of an assignment, as opposed to an *rvalue*, which is a value used in an expression or an assignment.

References

A reference is an ***alias*** for another variable:

```
int i = 10;
int &ir = i;    // reference (alias)
ir = ir + 1;    // increment i
```

Once initialized, references cannot be changed.



References are especially useful in ***procedure calls*** to avoid the overhead of passing arguments by value, without the clutter of explicit pointer dereferencing (y = *ptr;)

```
void refInc(int &n)
{
   n = n+1; // increment the variable n refers to
}
```

The there *reference* in most languages (e.g., Java) means something similar to a pointer. It is normally implemented as a kind of safe pointer to an object.

In C++, a *reference* is not a pointer, but an *alias*. This means that it occupies no memory of its own, but simply provides a new name for an already existing object. You can use the reference just as you would the original name, so no pointer dereferencing is needed to read or write the referred object.

The syntax for declaring a reference is unfortunately that same as the syntax for taking the address of an object in memory. However the reference syntax is only used in definitions, whereas an address may only be taken as part of an expression.

In the previous slide, `&` is only used to define `ir` and `n`, not to take the address of an object.

References vs Pointers

References should be preferred to pointers ***except*** *when:*

> manipulating dynamically allocated objects

—**`new`** returns an object pointer

> a variable must range over a set of objects

—use a **pointer** to walk through the set

References are inherently safer than pointers, since they involve no pointer arithmetic. However there are two situations where pointers are needed.

The first occurs when a new objects is created (with `new`). This always returns a pointer. To avoid dealing with pointers, the pointer can be immediately dereferenced and assigned to a reference variable:

```
Cat &myCat = *(new Cat());
```

To delete it, however, we must get the pointer back:

```
delete &myCat;
```

Note the two different uses of `&` above (the first to declare a reference and the second to take the address of the reference).

The second situation arises when you need to iterate over a collection. In this case, however, C++ provides *iterators* that eliminate the need too work with low-level pointers.

C++ Classes

C++ classes may be instantiated either *automatically* (on the stack):

```
MyClass oVal;     // constructor called
                  // destroyed when scope ends
```

or *dynamically* (in the heap)

```
MyClass *oPtr;         // uninitialized pointer

oPtr = new MyClass;  // constructor called
                     // must be explicitly deleted
```

Whereas in Java all objects exists exclusively in the heap, in C++ (as in C), objects may also be allocated automatically in the run-time stack.

As in Java, objects in the heap are created using the `new` keyword. Such objects must be explicitly deleted (i.e., using `delete`).

Objects in the stack are allocated automatically (hence the term *automatic* memory), when the functions they are defined in are executed.

The way a variable is declared and initialized tells us where it is:

```
Thing y; // automatic, on the stack
Thing *yp = &y; // a pointer to y on the stack
Thing &yr = y; // a reference to y on the stack
Thing *xp = new Thing; // a pointer to the heap
Thing &xr = *xp; // a reference to the heap
```

Constructors and destructors

Include standard iostream and string classes

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

class MyClass {
private:
   string name;
public:
   MyClass(string name) : name(name) {        // constructor
      cout << "create " << name << endl;
   }
   ~MyClass() {
      cout << "destroy " << name << endl;
   }
};
```

Use initialization list in constructor

Specify cleanup in destructor

Constructors in C++ may include an initialization list that initializes instance variables using their own constructors, and taking as arguments those specified in the initialization list. The initialization name(name) is analogous to:

```
 this.name = new string(name);
```

C++ class also support explicit *destructors*, which provide the details of how the memory used by the object is to be released. As a general rule, every piece of memory allocated with `new` must be released some where with a corresponding `delete`.

(The example on the previous slide is degenerate, since nothing is allocated or released; it simply reports when the constructor and destructor are called, needed for the following example.)

Automatic and dynamic destruction

```
#include "MyClass.h”
using namespace std;
int main (int argc, char **argv) {
   MyClass aClass("d");
   finish(start());
   return 0;
}
```

```
create d
create a
create b
destroy a
destroy b
destroy d
```

```
MyClass& start() {                   // returns a reference
   MyClass a("a");                   // automatic
   MyClass *b = new MyClass("b");    // dynamic
   return *b;                        // returns a reference (!) to b
}                                    // a goes out of scope
void finish(MyClass& b) {
   delete &b;                        // need pointer to b
}
```

Here we see when exactly constructors and destructors are called for objects on the stack and the heap.

Objects `a` and `d` are automatic, and are created and destroyed when their scopes, i.e., `main` and `start`, begin and end.

Object `b` is dynamic. It is explicitly created in the `start()` function, and then explicitly deleted in the `finish()` function.

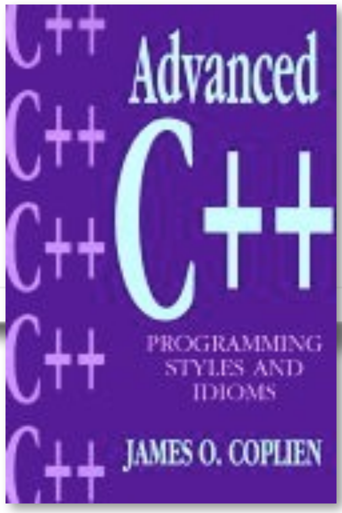
Roadmap

> C++ vs C
> C++ vs Java
> References vs pointers
> **C++ classes: Orthodox Canonical Form**
> A quick look at STL — The Standard Template Library

Orthodox Canonical Form

Most of your classes should look like this:

```
class myClass {
public:
   myClass(void);                         // default constructor
   myClass(const myClass& copy);          // copy constructor
      ...                                 // other constructors
   ~myClass(void);                        // destructor
   myClass& operator=(const myClass&);    // assignment
      ...                                 // other public member
functions
private:
      ...
};
```



In order to ensure that your objects are cleanly created and destroyed, that is, to make sure you have no memory leaks, and no dangling pointers, you should design your classes as promoted by Coplien.

This means you need to implement not only the default constructor and the destructor, but also a *copy constructor* and an *assignment operator*.

Why OCF?

If you don’t define these four member functions, *C++ will generate them:*

> ***default constructor***
> — will call default constructor for each data member
>
> ***destructor***
> — will call destructor of each data member
>
> ***copy constructor***
> — will *shallow copy* each data member
> — pointers will be copied, not the objects pointed to!
>
> ***assignment***
> — will *shallow copy* each data member

The default constructor (as in Java) takes no arguments.

The destructor should release (`delete`) all memory that has been indirectly allocated by the object (i.e., pointed to by its instance variables).

The copy constructor takes a reference to an object of the same type, and returns a pointer to a new object that is a copy of the original.

The assignment operator specifies how the representation of the argument (rhs, or right-hand side) is to be copied to the receiver (lhs, or left-hand side).

If you do not specify these four methods, C++ will generate them, assuming a shallow-copy strategy. This is very often the wrong thing to do, so it is better to specify them yourself.

Example: A String Class

We would like a `String` class that protects C-style strings:

> strings are indistinguishable from `char` pointers

> string updates may cause memory to be corrupted

Strings should support:

> creation and destruction

> initialization from char arrays

> copying

> safe indexing

> safe concatenation and updating

> output

> length, and other common operations ...

C++ actually has such a class (called `string`), which hides all the details of the underlying character array representation. The example illustrates all the main points of the orthodox canonical form.

A Simple String.h

Returns a reference to ostream

Operator overloading

A friend function prototype declaration of the String class

```
class String
{
    friend ostream& operator<<(ostream&, const String&);
public:
    String(void);                           // default constructor
    ~String(void);                          // destructor
    String(const String& copy);             // copy constructor
    String(const char*s);                   // char* constructor
    String& operator=(const String&);          // assignment

    inline int length(void) const { return ::strlen(_s); }
    char& operator[](const int n) throw(exception);
    String& operator+=(const String&) throw(exception);   // concatenation
private:
    char *_s; // invariant: _s points to a null-terminated heap string
    void become(const char*) throw(exception); // internal copy function
};
```

Operator overloading of =

inline

C and C++ distinguish *declarations* from *definitions*. A declaration specifies, or "declares" variables and functions needed to be able to use a component (i.e., a class or a package). Declarations simply state that something exists, but *does not allocate any memory* for it. Declarations go into header (or ".h") files, that can be shared by service providers and their clients.

Definitions, on the other hand, specify the implementations of variables and functions, so *they allocate memory* for them. Definitions go into source files, which may end in ".C", ".cpp" or even ".c++".

This "String.h" file declares (but does not define) the class `String`.

The "friend" function we will see later. It allows us to send `String` instances to an output stream.

In addition to the standard constructors and the destructor, we also declare a `char*` constructor to build a `String` object from a plain old C character array. We further declare a `length` method, an indexing operator `[]` and a concatenation operator `+=`.

Interestingly, we also declare the private representation. This is needed to compute the size of memory that clients need to reserve for `String` instances. In this case, it is no more nor less than a single character pointer into the heap.

Default Constructors

Every constructor should *establish the class invariant:*

Allocate memory for the string

```
String::String(void)
{
    _s = new char[1];           // allocate a char array
    _s[0] = '\0';               // NULL terminate it!
}
```

The *default constructor* for a class is called when a new instance is declared without any initialization parameters:

```
String anEmptyString;           // call String::String()
String stringVector[10];        // call it ten times!
```

While the declarations of the String methods belongs to the String.h file, their definitions (implementations) belong in the String.cpp file. After the header file is included, each method is implemented using its full name, i.e., `String::`*`method`*

The class invariant for our `String` class is that the `_s` variable should point to a valid character array (i.e., a C string) in the heap. This array must not be shared with any other object!

We initialize a new `String` to point to an empty string. In C, this is a character array of length 1, terminated by an ASCII NULL (0).

Destructors

The `String` destructor must *explicitly free* any memory allocated by that object.

```
String::~String (void)
{
   delete [] _s;
}
```

free memory

Every new must be matched somewhere by a delete!

> use `new` and `delete` for *objects*

> use `new[]` and `delete[]` for *arrays*!

Since the constructor creates a `char` array in the heap, it must be destroyed by the constructor. (Every object created by `new` must be destroyed somewhere by a `delete`.)

Caveat: arrays are created not by `new` but by `new[]`. They must accordingly be destroyed not by `delete` but by `delete []`.

Copy Constructors

Our `String` copy constructor must create a *deep copy:*

```
String::String(const String& copy)
{
   become(copy._s);              // call helper
}

void String::become(const char* s) throw (exception)
{
   _s = new char[::strlen(s) + 1];
   if (_s == 0) throw(logic_error("new failed"));
   ::strcpy(_s, s);
}
```

From std

The copy constructor takes a reference to an existing `String` object, and builds a copy of it. Note that the argument must be a reference. Since C and C++ are call-by-value languages, if the argument were specified simply as a `String`, then its value would have be to copied. But the method for copying `String` values is precisely what we are defining, so that could never work.

In addition, the argument is declared as a “`const`”. This declaration is a promise that we will not attempt to modify the argument in the body of the method. This allows us to create copies of strings declared to be constant. In general C++ requires that copy constructors not modify their arguments.

The implementation calculates the length of the string to be copied, creates a new character array of that length (+1 for the null terminator), and then copies the argument array to the new array.

The `become()` helper function will also be useful later for implementing the assignment operator.

Note that we must check that the result of `new` is not a null pointer (signaling an out-of-memory error).

A few remarks ...

> We **must** define a copy constructor,
… else copies of Strings will *share the same representation!*
— Modifying one will modify the other!
— Destroying one will invalidate the other!

> We **must** declare `copy` as `const`,
… else we won't be able to construct a copy of a `const String`!
— Only const (immutable) operations are permitted on const values

> We **must** declare `copy` as `String&`, not `String`,
… else a *new copy* will be made before it is passed to the constructor!
— Functions arguments are always passed by value in C++
— The "value" of a pointer is a pointer!

> **The encapsulation boundary is a class**, *not an object*. Within a class, **all private members are visible** (as is `copy._s`)

Note that in C++, as in Java, the encapsulation boundary is a class, not an object. Any instance of a given class A can access all the “private” state of other instances of the same class, so we can access `copy._s`, even though it is “private” to `copy`.

(In Smalltalk, the encapsulation boundary is the object.)

Other Constructors

Class constructors may have arbitrary arguments, as long as their signatures are unique and unambiguous:

```
String::String(const char* s)
{
   become(s);
}
```

Since the argument is not modified, we can declare it as **`const`**. This will allow us to construct `String` instances from constant `char` arrays.

Assignment Operators

Assignment is different from the copy constructor because *an instance already exists:*

```
String& String::operator=(const String& copy)
{
   if (this != &copy) {      // take care!
      delete [] _s;
      become(copy._s);
   }
   return *this;            // NB: a reference, not a copy
}
```

> Return **`String&`** rather than void so the result *can be used in an expression*

> Return **`String&`** rather than `String` so the result *won't be copied!*

> **`this`** is a pseudo-variable whose value is a pointer to the current object

—so **`*this`** is the value of the current object, which is *returned by reference*

Although this method is only a few lines long, there are a number of subtle things going on. As before, *the argument must be a reference*, to avoid copying it by value. Similarly, *we return a reference*, to avoid copying it, and to avoid returning a pointer.

All assignment operators should perform a sanity check to make sure that the rhs (right-hand side of the assignment) is not an alias for the lhs, or havoc may ensue! (In this case we would delete the memory of the object and then try to copy it!) Since this in C++ is always a pointer, we must *take the address of the argument* to perform the comparison.

Before performing the `become()`, we must release the old value of the lhs. Since it was allocated with `new[]` it must be freed with `delete[]`.

Finally, to return a reference to the updated object we return `*this` (a reference), not `this` (a pointer).

Implicit Conversion

When an argument of the “wrong” type is passed to a function, the C++ compiler looks for a constructor that will convert it to the “right” type:

```
str = "hello world";
```

is implicitly converted to:

```
str = String("hello world");
```

NB: compare to autoboxing in Java

Operator Overloading (indexing)

Not only assignment, but other useful operators can be “overloaded” provided their signatures are unique:

```
char& String::operator[] (const int n) throw(exception)
{
   if ((n<0) || (length()<=n)) {
      throw(logic_error("array index out of bounds"));
   }
   return _s[n];
}
```

NB: a non-const reference is returned, so can be used as an ***lvalue*** *in an assignment.*

Java allows you to overload methods, but not operators. The principle is the same, however. In both cases there is one syntactic operator or method, but many different implementations, each accepting different static types.

Overloaded operators are resolved at compile-time, based on the static types of the arguments. Consider:

```
"hello" + "there"
```

```
1 + 2
```

In each case, the compiler can statically determine which + operator is to be executed.

NB: Overloading is different from overriding, where methods inherited from a superclass may be overridden by a subclass. In that case, which method is to be executed can only be determined at run time, depending on the dynamic type of the first argument.

Overloadable Operators

The following operators may be overloaded:

+	-	*	/	%	^	&	|
-	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	*=
<<=	>>=	[]	()	->	->*	new	delete

NB: arity and precedence are fixed by C++

Friends

We would like to be able to write:

```
cout << String("TESTING ... ") << endl;
```

But:

- —**It can’t be a member function of `ostream`, since we can’t extend the standard library.**
- —It can’t be a member function of `String` since the target is **`cout`.**
- —But it must have access to `String`’s **private data**

So ... we need a binary *function* << that takes a `cout` and a `String` as arguments, and is a *`friend`* of `String`.

“Friends” are peculiar to C++, but are reminiscent of Java's “package scope”. The idea is that classes that are designed to work closely together may need access to each other’s private state. In Java you would put these together in the same package and declare the state to be of package scope (i.e., without any “`private`” or “`protected`” modifier). In C++ you can declare the privileged functions as “friends”.

Friends ...

*We **declare**:*

```
class String
{
 friend ostream&
        operator<<(ostream&, const String&);
   ...
};
```

*And **define**:*

```
ostream&
operator<<(ostream& outStream, const String& s)
{
   return outStream << s._s;
}
```

Roadmap

- C++ vs C
- C++ vs Java
- References vs pointers
- C++ classes: Orthodox Canonical Form
- **A quick look at STL — The Standard Template Library**

Standard Template Library

STL is a general-purpose C++ library of generic algorithms and data structures.

1. Containers store *collections of objects*
   — `vector, list, deque, set, multiset, map, multimap`
2. Iterators *traverse containers*
   — random access, bidirectional, forward/backward ...
3. Function Objects encapsulate *functions as objects*
   — arithmetic, comparison, logical, and user-defined ...
4. Algorithms implement *generic procedures*
   — `search, count, copy, random_shuffle, sort, ...`
5. Adaptors provide an *alternative interface* to a component
   — `stack, queue, reverse_iterator, ...`

An STL Line Reverser

```
#include <iostream>
#include <stack>                          // STL stacks
#include <string>                         // Standard strings

void rev(void)
{
   typedef stack<string> IOStack;// instantiate the template
   IOStack ioStack;               // instantiate the template class
   string buf;

   while (getline(cin, buf)) {
      ioStack.push(buf);
   }
   while (ioStack.size() != 0) {
      cout << ioStack.top() << endl;
      ioStack.pop();
   }
}
```

Note how the high-level abstractions of strings and stacks completely hide the low-level details of memory management. The equivalent program in C is far more complex, difficult to read and to debug.

Modern C++ pushes this idea even further and approaches a level of abstraction close to that of Java.

What we didn’t have time for ...

- virtual member functions, pure virtuals
- public, private and multiple inheritance
- default arguments, default initializers
- method overloading
- const declarations
- enumerations
- smart pointers
- static and dynamic casts
- Templates, STL
- template specialization
- namespaces
- RTTI
- ...

What you should know!

- ✎ What new features does C++ add to C?
- ✎ What does Java remove from C++?
- ✎ How should you use C and C++ commenting styles?
- ✎ How does a reference differ from a pointer?
- ✎ When should you use pointers in C++?
- ✎ Where do C++ objects live in memory?
- ✎ What is a member initialization list?
- ✎ Why does C++ need destructors?
- ✎ What is OCF and why is it important?
- ✎ What’s the difference between delete and delete[]?
- ✎ What is operator overloading?

Can you answer these questions?

- ✎ *Why doesn’t C++ support garbage collection?*
- ✎ *Why doesn’t Java support multiple inheritance?*
- ✎ *What trouble can you get into with references?*
- ✎ *Why doesn’t C++ just make deep copies by default?*
- ✎ *How can you declare a class without a default constructor?*
- ✎ *Why can objects of the same class access each others private members?*