A bit of C++

Alexandre Bergel
abergel@dcc.uchile.cl
21/06/2010

Roadmap

1. C++ vs C
2. C++ vs Java
3. References vs pointers
4. C++ classes: Orthodox Canonical Form
5. A quick look at STL — The Standard Template Library

Roadmap

1. **C++ vs C**
2. C++ vs Java
3. References vs pointers
4. C++ classes: Orthodox Canonical Form
5. A quick look at STL — The Standard Template Library

Essential C++ Texts

Bjarne Stroustrup, *The C++ Programming Language (Special Edition)*, Addison Wesley, 2000.

Stanley B. Lippman and Josee LaJoie, *C++ Primer*, Third Edition, Addison-Wesley, 1998.

Scott Meyers, *Effective C++*, 2d ed., Addison-Wesley, 1998.

James O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

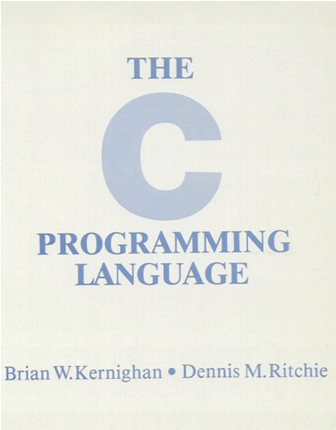
David R. Musser, Gilmer J. Derge and Atul Saini, *STL Tutorial and Reference Guide*, 2d ed., Addison-Wesley, 2000.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.

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*"



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

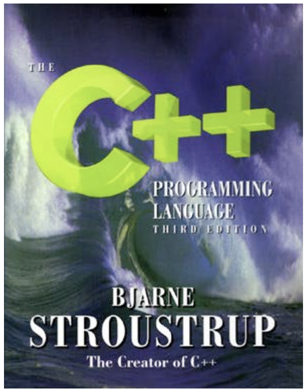
Indicate correct termination

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++ 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++ vs C

C++ encourages a different style of programming:

- avoid procedural programming
- model your domain with classes and templates

Roadmap

1. C++ vs C
2. **C++ vs Java**
3. References vs pointers
4. C++ classes: Orthodox Canonical Form
5. A quick look at STL — The Standard Template Library

Hello World in Java

```java
package cc3002;
// My first Java program!
public class HelloMain {
    public static void main(String[] args) {
      System.out.println("hello world!");
   }
}
```

“Hello World” in C++

Use the standard namespace

Include standard iostream classes

A C++ comment

```cpp
using namespace std;
#include <iostream>
// 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!)

Makefiles / Managed Make in CDT

You could compile it all together by hand:

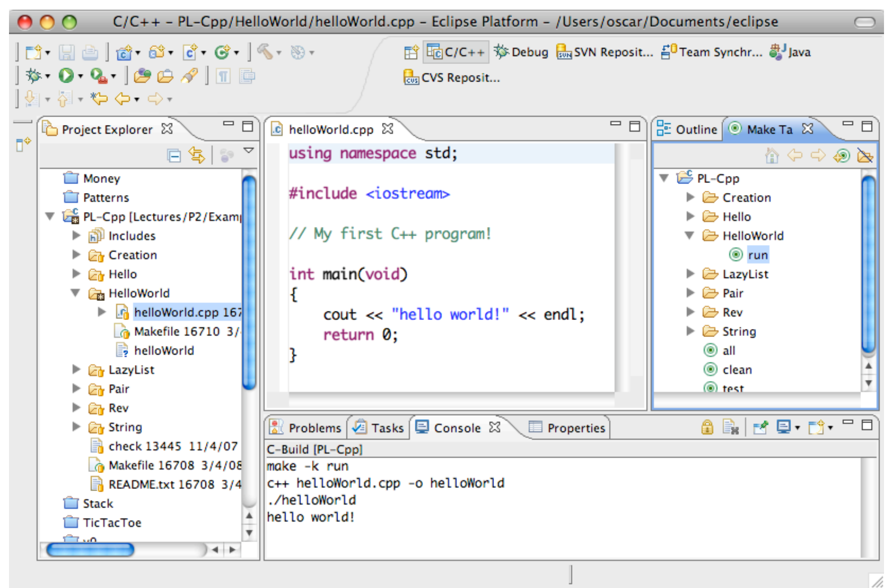
```
c++ helloWorld.cpp -o helloWorld
```

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

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

```
make helloWorld
```

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



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*

C++ Features

C with Classes	Classes as structs Inheritance; virtual functions Inline functions
C++ 1.0 (1985)	Strong typing; function prototypes new and delete operators
C++ 2.0	Local classes; protected members Multiple inheritance
C++ 3.0	Templates Exception handling
ANSI C++ (1998)	Namespaces RTTI (Runtime Type Information)

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

Some Java Extensions:
- garbage collection
- standard abstract machine
- standard classes (came later to C++)
- packages (now C++ has namespaces)
- final classes
- autoboxing
- generics instead of templates

Java Simplifications of C++

no pointers — *just references*

no functions — *can declare static methods*

no global variables — *use public static variables*

no destructors — *garbage collection and finalize*

no linking — *dynamic class loading*

no header files — *can define interface*

Java Simplifications of C++

no operator overloading — *only method overloading*

no member initialization lists — *call super constructor*

no preprocessor — *static final constants and automatic inlining*

no multiple inheritance — *implement multiple interfaces*

no structs, unions, enums — *typically not needed*

New Keywords

In addition 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)

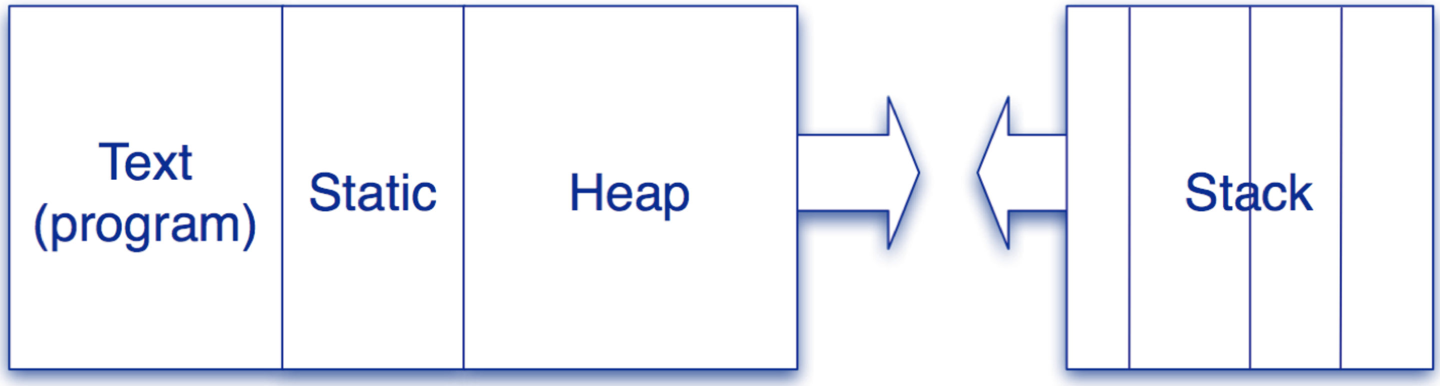
Roadmap

1. C++ vs C
2. C++ vs Java
3. **References vs pointers**
4. C++ classes: Orthodox Canonical Form
5. 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
Stack:	local memory for function calls (grows downward)



Pointers in C++

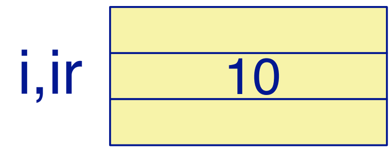
```
int i;
int *iPtr; // a pointer to an integer

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

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

References

A reference is an alias for another variable:




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

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
}
```

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

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
```

Constructors and destructors

Include standard iostream and string classes

Use initialization list in constructor

Specify cleanup in destructor

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

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

```
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
}
```

```
#include "MyClass.h"


using namespace std;


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

   MyClass aClass("d");
   MyClass& bClass = aClass.start();

   aClass.finish(bClass);
   return 0;

}
```

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

Roadmap

1. C++ vs C
2. C++ vs Java
3. References vs pointers
4. **C++ classes: Orthodox Canonical Form**
5. 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:
      ...
};
```

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

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 ...

A Simple String.h

```
class String
{
   friend ostream& operator<<(ostream&, const String&);
public:
   String(void);                       // default construct
   ~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
};
```

Returns a reference to ostream

Operator overloading

A friend function prototype declaration of the String class

Operator overloading of =

inline

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!
```

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*!

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

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

A few remarks ...

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 abstraction boundary is a class, not an object. Within a class, all private members are visible (as is copy._s)

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

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

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 ...

We declare:

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

And define:

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

Roadmap

1. C++ vs C
2. C++ vs Java
3. References vs pointers
4. C++ classes: Orthodox Canonical Form
5. **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.

Containers store *collections of objects*

- vector, list, deque, set, multiset, map, multimap

Iterators *traverse containers*

- random access, bidirectional, forward/backward ...

Function Objects encapsulate *functions as objects*

- arithmetic, comparison, logical, and user-defined ...

Algorithms implement *generic procedures*

- search, count, copy, random_shuffle, sort, ...

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();
   }
}
```

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

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?

License

http://creativecommons.org/licenses/by-sa/2.5