C++ for Java Programmers – 1

WL101.1a

©Wilhansen Li 2010 <wil@byimplication.com>

Kindly prepare tissues for epic nosebleeds.

Outline

- A Simple Program
- Basic Data Types
- Type Modifiers
- Type system
- Variable declaration
- Typedef's
- bool Translation
- Constants
- Statics
- Functions
- The #include
- The Preprocessor

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

```
Comp:~ User$ make hello_world

Comp:~ User$ g++ -o hello_world hello_world.cpp

C:/> cl hello_world.cpp
```

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

- "We'll be using these libraries..."
- Similar to "import" in Java.
- Details in session 2.

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

- If not for this, we'll be prefixing everything with std::
- Details in session 2.

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

- Starting point of execution.
- For Java, it's like `public static void main(String [] args)`

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

Declare a variable named “name” which is a string.

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

- Read a string from input.
- Automatic type deduction: if `name` was declared as an `int`, it would read it as an `int`.
- Equivalent to `Scanner.nextXXX()` in Java.

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

- Print stuff to the output like `System.out.print()`.
- Chain outputs with `<<`.
- It will automagically convert any variable to a string (even if it's not a string).

A Simple Program

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

using namespace std;

int main() {
    string name;
    cin >> name;
    cout << "Hello " << name << "!\n";
    return 0;
}
```

- Return from the function (i.e. exit the program).
- 0 usually means “success”.
- Lengthy, unnecessary explanation so just follow it...

Basic Data Types

Type	Values	Literal
bool	0/1, true/false	true, false, 10, -2
char	-127 to 128	0xFF,'C'
short	-32767 to 32768	0x10, 0777, 421, 'BEEF'
int/long	~ -2 billion to 2 billion	
long long	$-2^{63}-1$ to 2^{63}	124LL
float	~7 digits of precision	1.0f, 2.f, 1.2e10f
double	~15 digits of precision	1.0, 2., 1.2e10
long double	~ 40 digits of precision	1.0LD, 2.LD, 1.2e10LD

Unsigned Variants

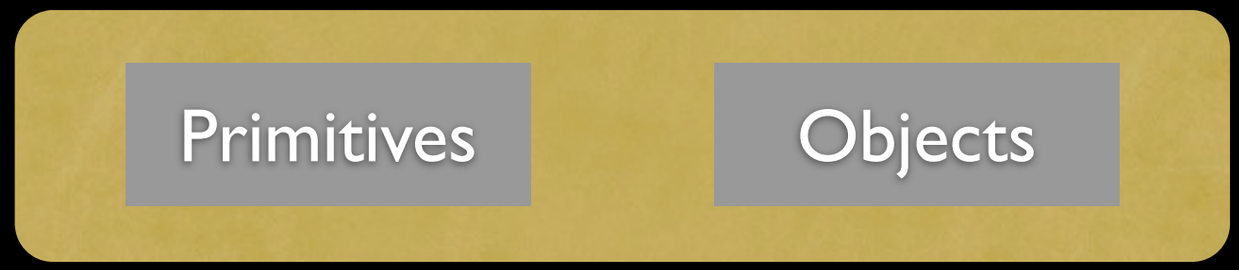
Type	Values	Literal
unsigned char	0 to 255	‘C’
unsigned short	0 to 65535	0x10U, 0777U, 421U, ‘BEEF’
unsigned int	0 to ~4 billion	
unsigned long long	0 to 2^{64}-1	124ULL

Type modifiers

- `unsigned` (discussed)
- `const` (later)
- `static` (later)
- `extern`
- `*`
- `&`

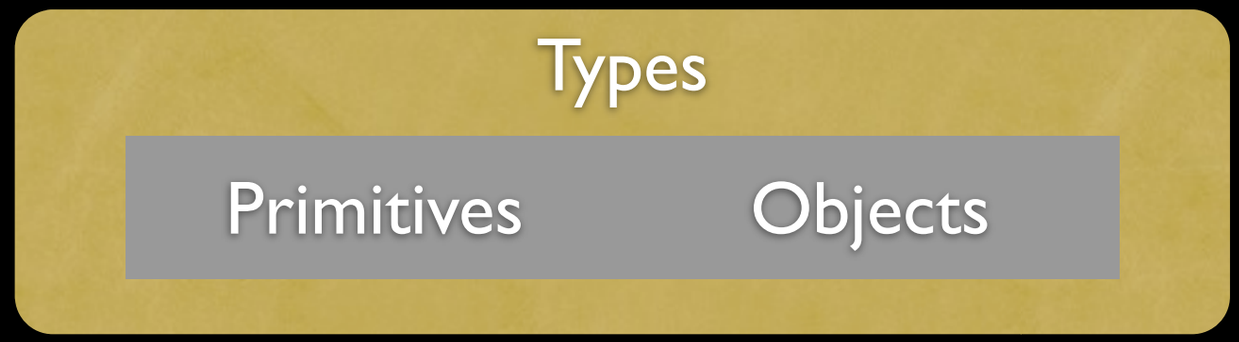
Type System

Java:



Latter OOP (i.e. Ruby):

C++:



Treat objects as if they are primitives!

Variable Declaration

- Similar to Java
- Caveat: uninitialized primitive variables have undefined content (and don't turn it into a source for random numbers!)

```
int a; //uninitialized
unsigned int b = 0;
string str; //initialized to ""

cout << b; //prints 0
cout << a; //prints a "random" number
```

Variable Declarations

- For class types...

Java
```
SomeClass a = new SomeClass(param);
```

C++
```
SomeClass a(param);
```

Note that

```
SomeClass a();
```
function declaration

≠

```
SomeClass a;
```
variable declaration

Typedef's

- Assign an alias to a type.
- Declaration is similar to variable declaration.
- The identifier specified becomes the name of the new type.

```
typedef unsigned long long ulolo;
ulolo a = 0;
ulolo foo() { return 0; }
```

bool translation

- In Java, a `boolean` is its own type.
- In C++ however, any integer type and some other types are automatically translated to a boolean when needed.
- `0` is `false`, any other value is `true` (even negative numbers)

```
//won't work in java
if (1) {}
int a = 0;
if (a) {}
```

Checkpoint 1.0

Figure out what the code does.

```
#include <iostream>
using namespace std;
typedef long long T;

int main() {
  T a, b = 0;
  cin >> a;
  while ( !b ) {
    cin >> b;
  }
  cout << a / b << '\n';
  return 0;
}
```

Checkpoint 1.1

What's wrong with this code?

```
#include <iostream>
using namespace std;
typedef long long T;

int main() {
  T a, b;
  cin >> a;
  while ( !b ) {
    cin >> b;
  }
  cout << a / b << '\n';
  return 0;
}
```

constants

- Variables that can't be modified.
- Similar to Java's `final` keyword.
- Judicious use can prevent programming mistakes.

```
//Usual immutable variable
const double PI = 3.14159265358979;
PI = 42; //not allowed!
```

`static`s

- Means different things depending on scope:
  - File: Variable is only visible in the file. Can be combined with `const`.
  - Function: Variable initializes ONCE on first encounter.

```
static const int SCREEN_WIDTH = 640;

int incr() {
 static int i = 0;
 return i++;
}
```

Functions

- Unlike Java where all methods have to be in classes, C++ can have functions in the global scope.
- Functions "declared out there" are called *global functions*.
- Functions declared in classes are called *member functions*.
- Function signature similar to Java:

```
return_type name(parameters);
```

Declaration v.s. Definition

- Usually you would write functions like this:

```
void foo() {
    //stuff here
}
```

- This is called a *definition* since you state that this function exists and this is what it does.

Declaration v.s. Definition

- On the other hand, you *declare* a function like this:

```
void foo();
```

  - You write the function signature but not the actual contents.
  - In Java, it's just like interface or abstract methods.
  - You simply say that it exists, not what it does.
  - You still have to define the function somewhere in the code.

Ordering Principle

- Unlike Java, the C++ compiler scans the source file from top to down.
- That means, you have to order your variables and functions carefully.
- The rule is: *Declare* before using.

```
int main() {
    foo();
}

void foo() {
}
```

```
void foo() {
}

int main() {
    foo();
}
```

```
void foo();

int main() {
    foo();
}

void foo() {
}
```

Circular Dependence

- What if you have two or more functions that rely on each other?
- Declare one first.

```
void foo() {
    bar();
}

void bar() {
    foo();
}

int main() {
    foo();
}
```

```
void bar();

void foo() {
    bar();
}

void bar() {
    foo();
}

int main() {
    foo();
}
```

Functions + const

- Note that passed arguments are "cast" to parameters.
- This applies to variables passed to constants.

```
void foo(const int param) {
 //param cannot be modified!
}

int a;
cin >> a;
//'int' implicitly casted to 'const int'
foo(a);
```

Functions + const

- Functions can also return const values.
- Useless for now.

```
const int foo() {}
```

Checkpoint 1.2

What is the output?

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

int incr(const int a) {
  static int i = 0;
  return a * i++;
}

int main() {
  int mult;
  cin >> mult;
  for ( int i = 0; i < 5; ++i ) {
     cout << incr(mult) << '\n';
  }
  return 0;
}
```

Parameter Tricks

- Parameter names can be omitted, causing them to be "ignored parameters".

```
void foo(int) { /*...*/ }
```

- It is possible to have "default parameters" which are automatically passed if the corresponding parameters are omitted.

```
void add(int a, int b = 0) { return a + b; }
add(1); //same as calling add(1, 0)
```

The #include

- `#include <fileName>` instructs the preprocessor (to be discussed later) to find the file called fileName in the "search paths" and literally, copy-paste its contents to where it's invoked.
- `#include "fileName"` does the same thing but the preprocessor starts looking at the directory of the origin.

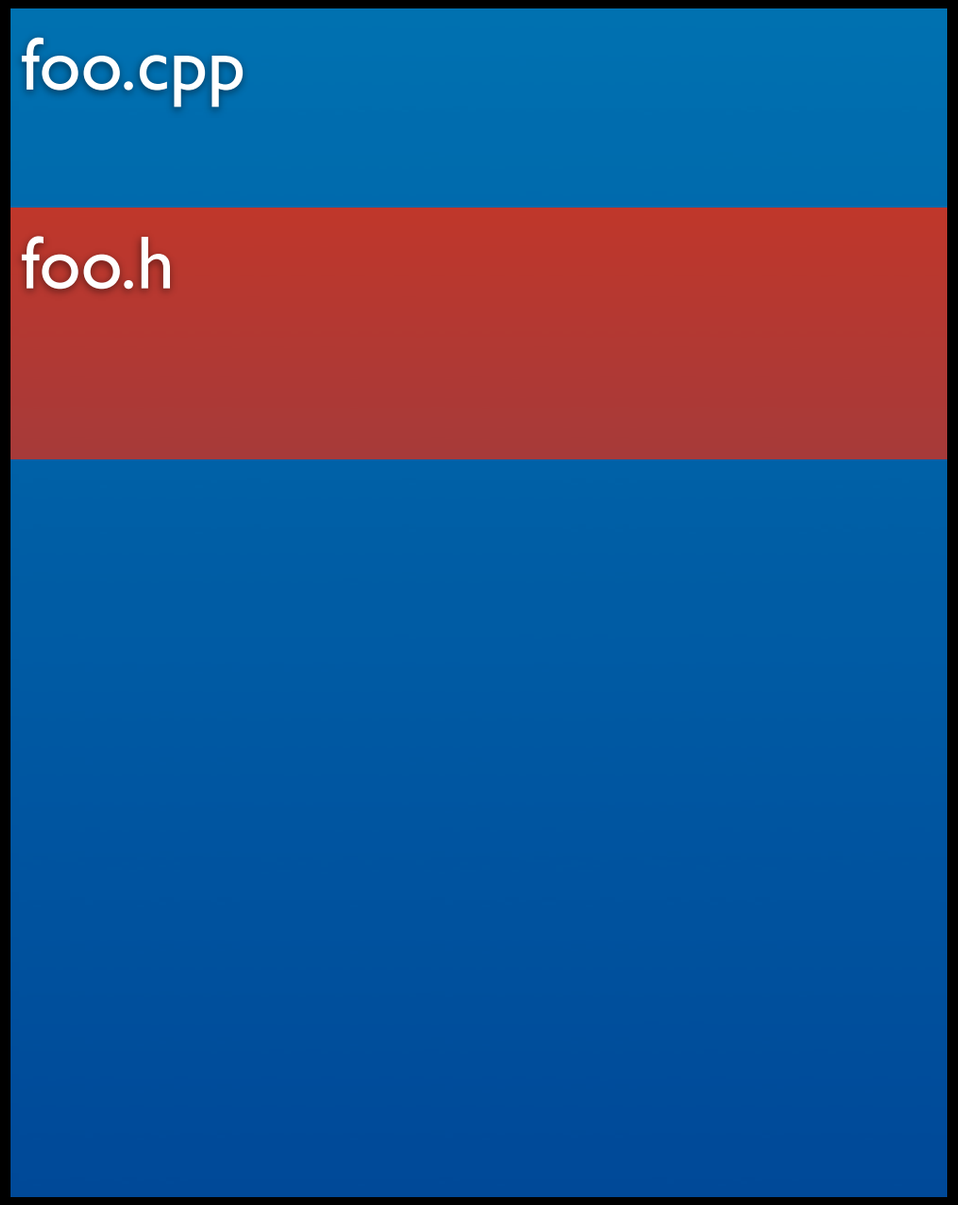
#include Illustration

foo.cpp

```
#include "foo.h"
```

foo.h

#include Illustration



Orly? (try it out!)

Save in Header.h☛

```
//Header.h

"Hello There!\n"
```

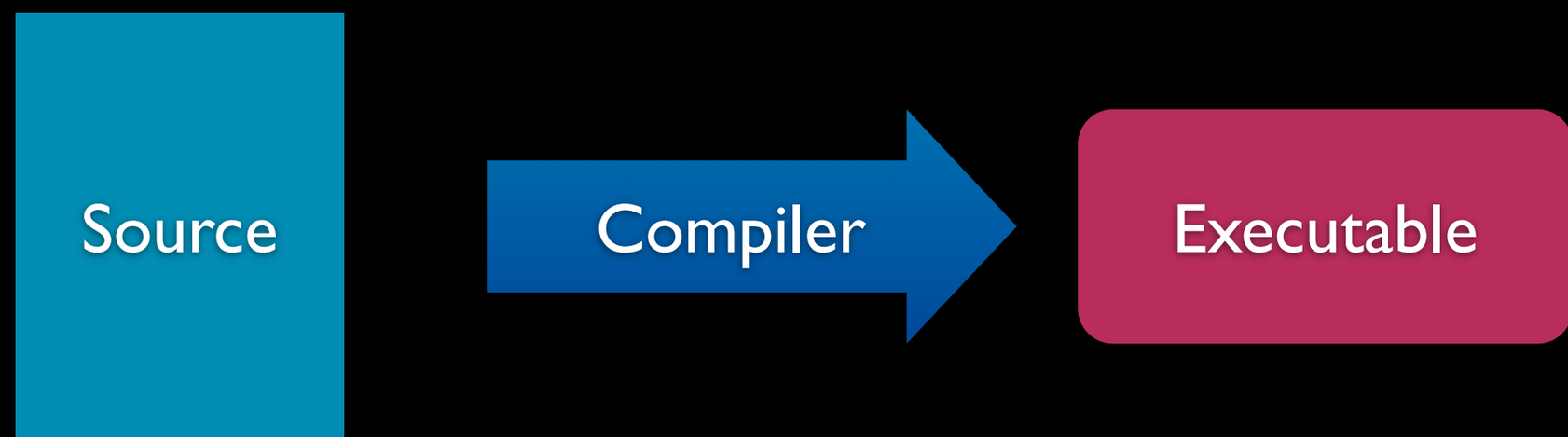
Save in includeTest.cpp☛

```
#include <iostream>

using namespace std;

int main() {
    cout <<
#include "Header.h"
    <<
#include "Header.h"
    << '\n';
    return 0;
}
```

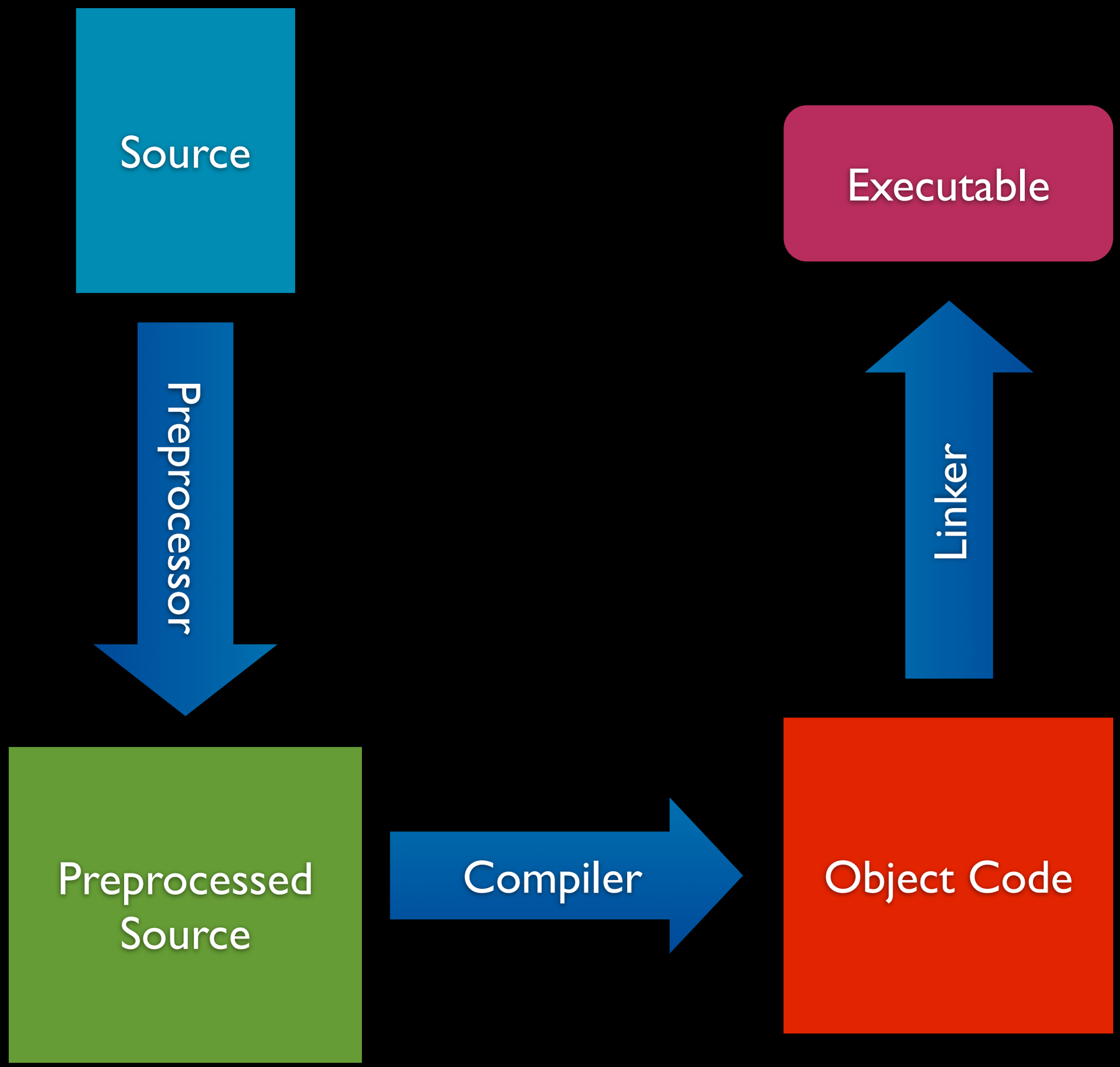
☑Try using `#include<Header.h>` and see what happens.

The “Preprocessor”

- The `#include <>` directive is part of the preprocessor
- Prepares the source code before handing it off to the *actual* compiler.
- All preprocessor directives start with a # at the beginning of the line.
- Doesn’t use a semicolon to end the line.



What you see.



What actually happens.

Other Directives

- #define, #undef
- #if, #else, #elif, #endif
- #ifdef
- #pragma
- #warning
- #error

} Focus on these.

#define

- Defines a preprocessor constant or macro.
- You can define a constant with no value.
- Use defined tokens as normal identifiers.
- Like the **#include**, it also does a copy-paste or specifically, find-and-replace.

```
#define TOKEN
#define Author "Shikababa Kaul"
#define Mag(x,y) sqrt(x*x + y*y)
```

#define Illustration

#define Illustration

```
#define Author "Shikababa Kaul"
```

#define Illustration

```
#define Author "Shikababa Kaul"
```

```
cout << Author << ' ';
```

#define Illustration

```
#define Author "Shikababa Kaul"
```

+

```
cout << Author << ' ';
```

#define Illustration

```
#define Author "Shikababa Kaul"
```

+

```
cout << Author << ' ';
```

```
cout << "Shikababa Kaul" << ' ';
```

#define Illustration

```
#define Author "Shikababa Kaul"
```

+

```
cout << Author << ' ';
```

```
cout << "Shikababa Kaul" << ' ';
```

```
#define TOKEN
```

#define Illustration

```
#define Author "Shikababa Kaul"
```

+

```
cout << Author << ' ';
```

```
cout << "Shikababa Kaul" << ' ';
```

```
#define TOKEN
```

```
cout << TOKEN << ' ';
```

#define Illustration

```
#define Author "Shikababa Kaul"
```

+

```
cout << Author << ' ';
```

```
cout << "Shikababa Kaul" << ' ';
```

```
#define TOKEN
```

+

```
cout << TOKEN << ' ';
```

#define Illustration

```
#define Author "Shikababa Kaul"
```

+

```
cout << Author << ' ';
```

```
cout << "Shikababa Kaul" << ' ';
```

```
#define TOKEN
```

+

```
cout << TOKEN << ' ';
```

```
cout <<  << ' ';
```

Try it (using GCC)

- Use the below test file.
- Open the command line and navigate to the directory of the source file.
- Type: `g++ -E source.cpp -o source.pre.cpp`

```
#define Combine(x,y,op) x + x op y + y
#define Linear(x,y) Combine(x,y,*)

double a, b;
Combine(a,b,-);
Linear(a,b);
```

#define Example

```
#include <iostream>
#include <cmath>

#define Author "Shikababa Kaul"
#define Mag(x,y) sqrt(x*x + y*y)
#define Magic cout << Author << ' ' << Mag(a,b) << endl

using namespace std;
int main() {
   double a = 10, b = 20;
   Magic;
}
```

#define Example

```
#include <iostream>
#include <cmath>

#define Author "Shikababa Kaul"
#define Mag(x,y) sqrt(x*x + y*y)
#define Magic cout << Author << ' ' << Mag(a,b) << endl

using namespace std;
int main() {
   double a = 10, b = 20;
   Magic;
}
```

```
#include <iostream>
#include <cmath>


using namespace std;
int main() {
   double a = 10, b = 20;
   cout << "Shikababa Kaul" << ' ' << sqrt(a*a + b*b) << endl;
}
```

Caveat

```
#include <iostream>

#define ADD(x,y) x + y

using namespace std;
int main() {
  double res = ADD(1,2) * ADD(3,4);
  cout << res;
}
```

What's the output?

Caveat

```
ADD(1,2) * ADD(3,4)
```

→

```
1 + 2 * 3 + 4
```

So to make it work as expected...

```
#define ADD(x,y) (x + y)
```

→

```
(1 + 2) * (3 + 4)
```

#undef

- Opposite of **`#define`**.
- Removes the definition of the token.

```
#undef TOKEN
```

#if, #endif, #else, #elif

- Preprocessor's equivalent of if statements.
- Limited to testing for token existence and integer comparison.

```
#include <iostream>

using namespace std;
int main() {
#if 1
   cout << "Hello!\n";
#else
   cout << "World!\n";
#endif
}
```

☑Try changing the 1 to a 0

Illustration

```
#define DEBUGGING 0

#if DEBUGGING
cout << "Debugging...\n";
#else
cout << "Not debugging..\n";
#endif
```

```
#if 0
cout << "Debugging...\n";
#else
cout << "Not debugging..\n";
#endif
```

```
cout << "Not debugging..\n";
```

☑Try using **`#if defined(DEBUGGING)`** instead and see what happens when you retain or remove **`#define DEBUGGING`** (doesn't matter what values it uses)

```
g++ -E source.cpp -o source.pre.cpp
```

Checkpoint 1.3

What's the difference between the two?

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

int main() {
  if ( 1 )
    cout << "true\n";
  else
    cout << "false\n";
  return 0;
}
```

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

int main() {
#if 1
    cout << "true\n";
#else
    cout << "false\n";
#endif
  return 0;
}
```

Checkpoint 1.4

What does the following do?

```
#ifdef WIN32
#include <windows.h>
#elif defined(GNOME)
#include <gtk.h>
#elif defined(MAC_OSX)
#include <Cocoa/Cocoa.h>
#endif
```