

Basic Concepts

Advanced Programming
Brent van Bladel



Introduction

C++

- Flexibility towards multiple Programming Paradigms

Procedural, Object-Oriented, Generic paradigms

- Developer has control

Allows for optimization of memory management and CPU usage

- Compiled language

Faster, more efficient, compiler optimizations

Compiled vs Interpreted Languages

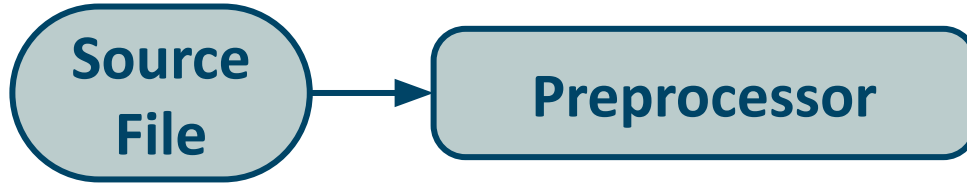
Compiled

Source code is converted to executable machine code before runtime.

Interpreted

Source code is converted to executable machine code at runtime.

C++ Compiler Phases: Preprocessor



Source code is preprocessed using **macros** (start with #).

```
#include <iostream>

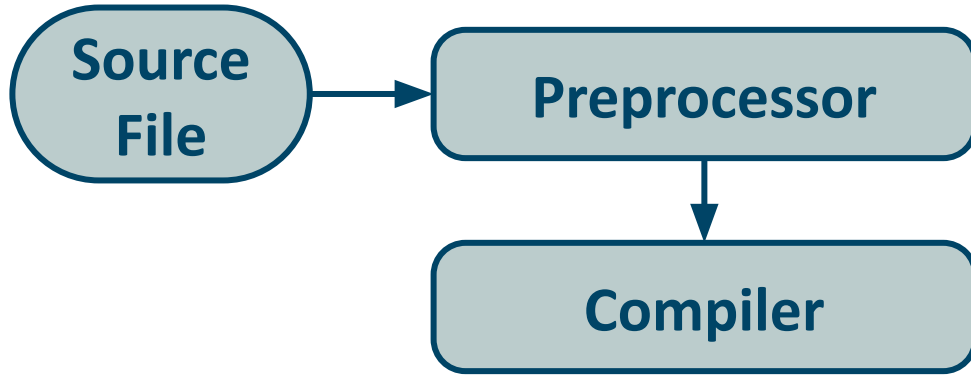
#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {

    std::cout <<"The minimum is " << MIN(10, 25) << std::endl;

    return 0;
}
```

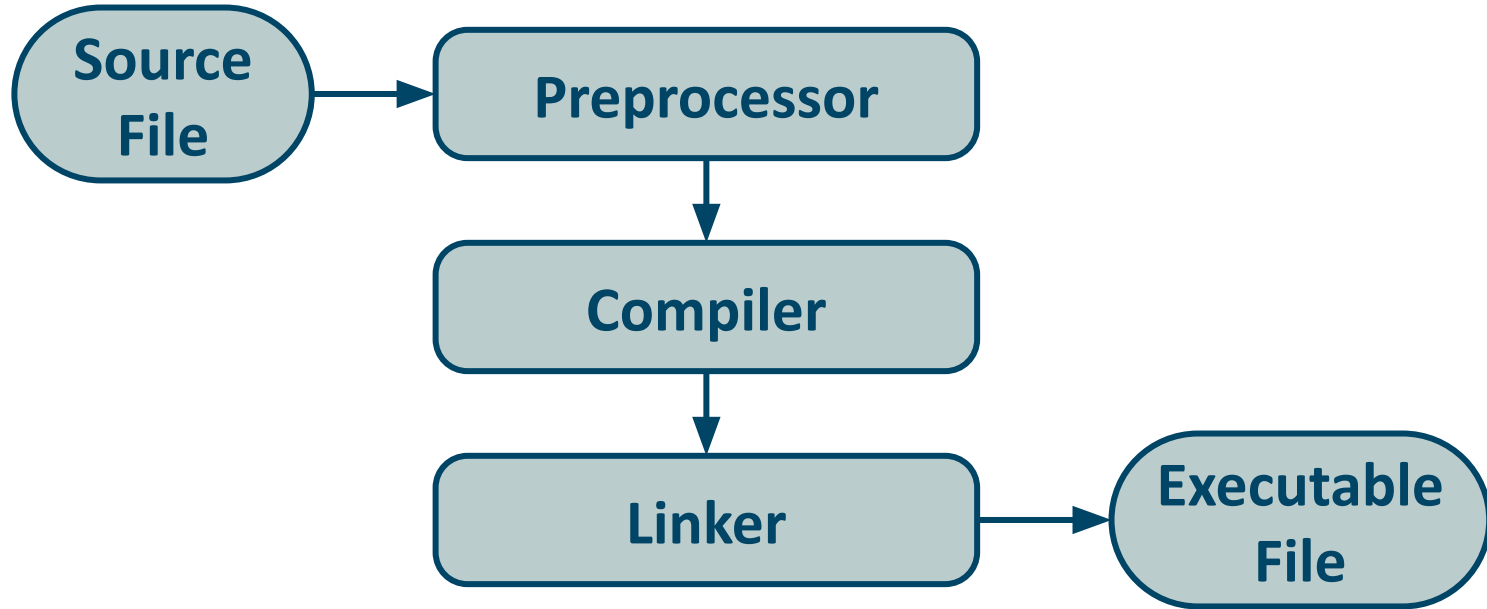
C++ Compiler Phases: Compiler



Preprocessed source code is converted to executable machine code, typically object files.

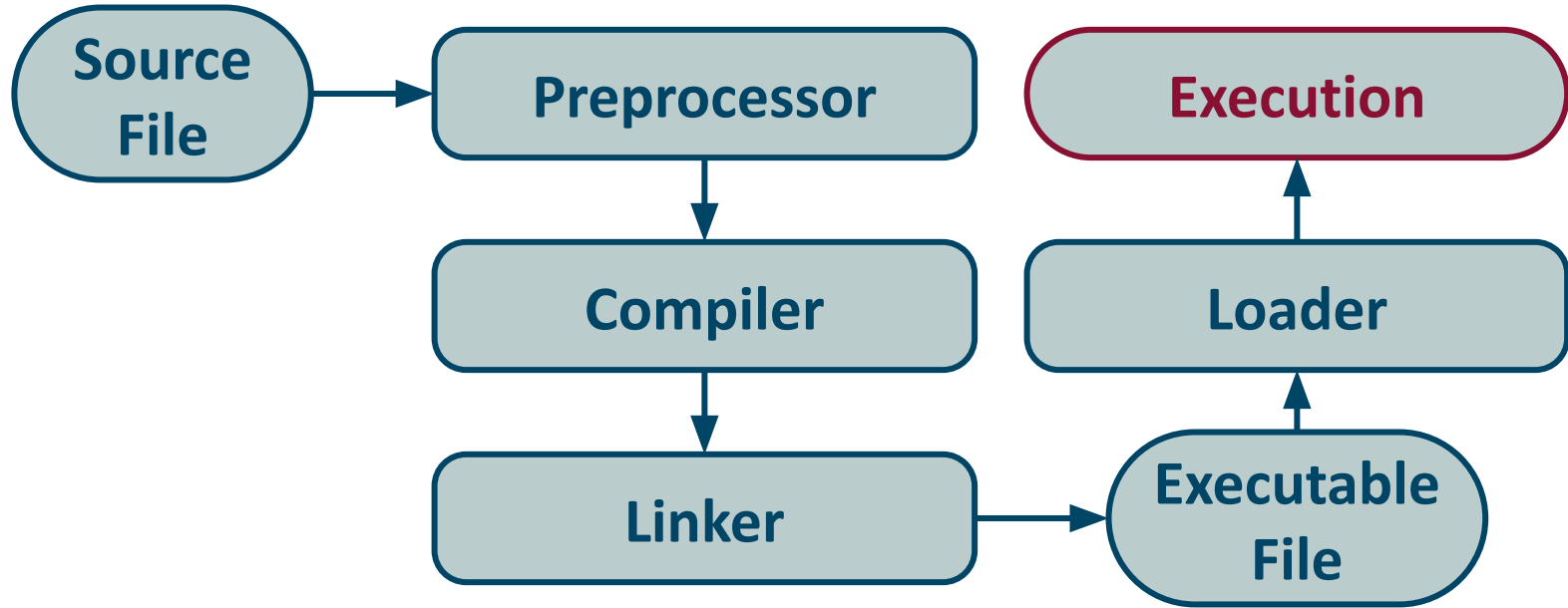
One object file for each source file is created.

C++ Compiler Phases: Linker



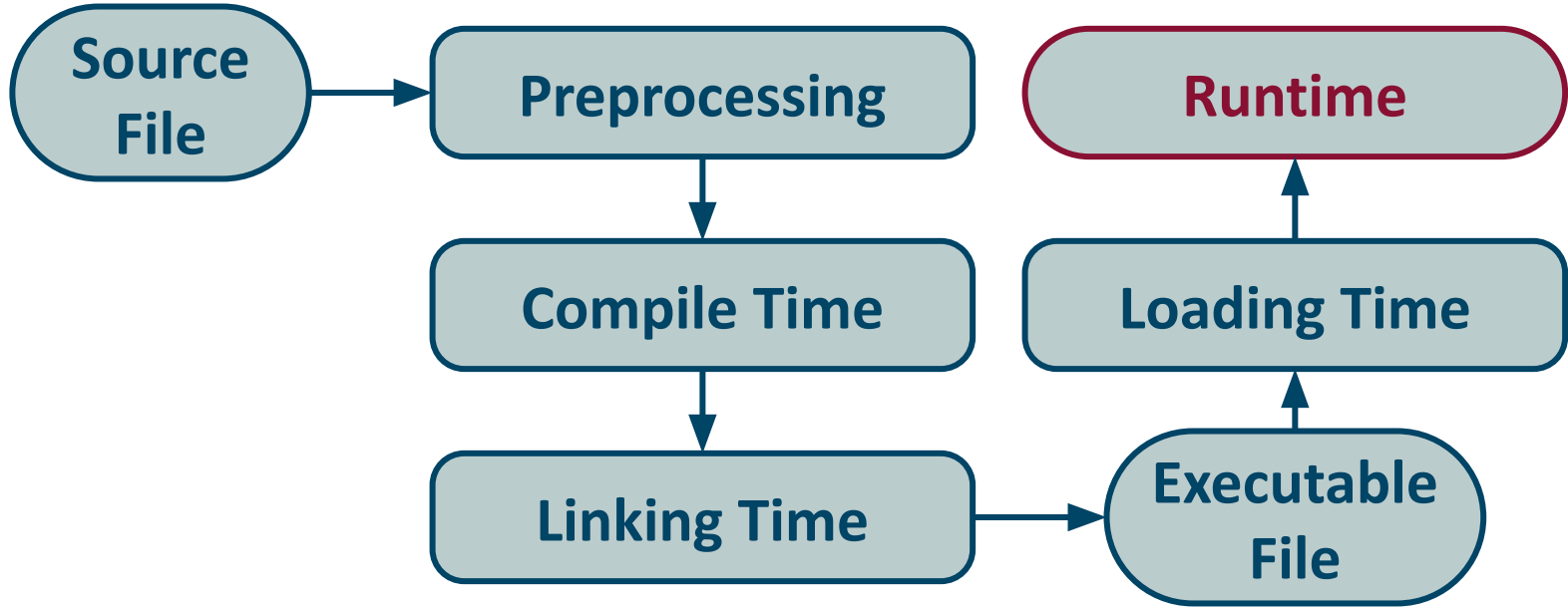
Object files are linked to create one executable file.

C++ Compiler Phases: Loader



The executable code is loaded into memory, after which it can be run.

C++ Timings

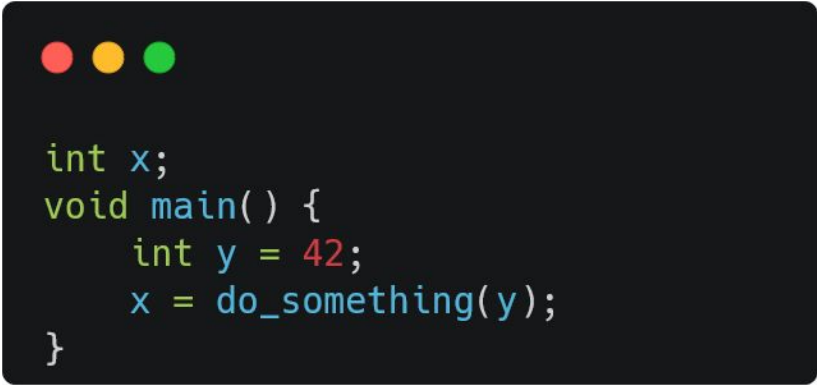


Binding

Binding is the act of associating properties with names.

Note that different properties are associated at different times.

Binding Time



```
int x;  
void main() {  
    int y = 42;  
    x = do_something(y);  
}
```

- When is the type of `x` bound? **Compile time**
- When is the reference to `do_something*` bound? **Linking time**
- When is the memory location for `x` bound? **Loading time**
- When is the memory location for `y` bound? **Runtime**
- When is the value of `x` bound? **Runtime**

Static vs Dynamic Binding

Static Binding occurs before runtime and remains unchanged during the program execution.

Dynamic Binding occurs during program execution and can change during the program execution.

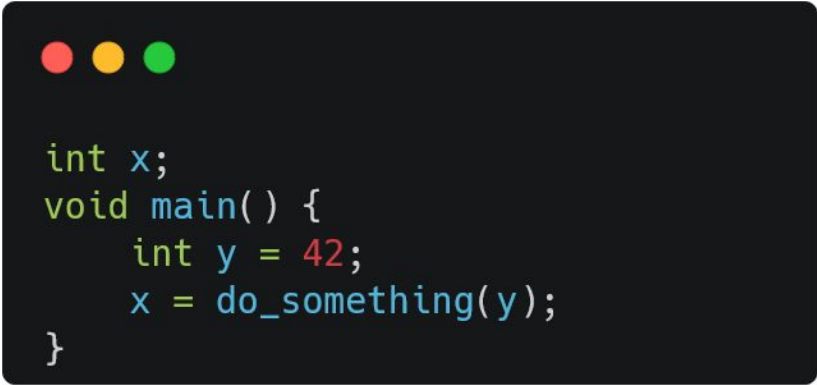
Binding Time

Static Binding

```
int x;  
void main() {  
    int y = 42;  
    x = do_something(y);  
}
```

- When is the type of *x* bound? **Compile time**
- When is the reference to *do_something** bound? **Linking time**
- When is the memory location for *x* bound? **Loading time**
- When is the memory location for *y* bound? **Runtime**
- When is the value of *x* bound? **Runtime**

Binding Time



```
int x;  
void main() {  
    int y = 42;  
    x = do_something(y);  
}
```

- When is the type of `x` bound? **Compile time**
- When is the reference to `do_something` bound? **Linking time**
- When is the memory location for `x` bound? **Loading time**

Dynamic Binding

- When is the memory location for `y` bound? **Runtime**
- When is the value of `x` bound? **Runtime**

Static vs Dynamic Semantics

Static Property

- Applies to the source code, not to a execution.
- Can be determined from source code.

Dynamic Property

- Occurs during program execution.
- Can be inferred from the execution of a program.

Static vs Dynamic Types

Static Type

The type of an expression, which type results from analysis of the program without considering execution semantics.

Dynamic Type

The type of the most derived object to which the expression refers.

Note that every variable has both a static type and a dynamic type.

Static vs Dynamic Types Example



```
struct Base {    virtual string name(){ return "Base";    } };  
struct Derived : Base { string name(){ return "Derived"; } };  
  
void print(Base& b) { cout << b.name(); }  
  
int main(int argc, char* argv[]){  
    Base base;  
    Derived derived;  
    if (argc == 1)  
        print(base);  
    else  
        print(derived);  
}
```

Variable *base*:

- **Static type:** Base
- **Dynamic type:** Base

Static vs Dynamic Types Example



```
struct Base {    virtual string name(){ return "Base";    } };  
struct Derived : Base { string name(){ return "Derived"; } };  
  
void print(Base& b) { cout << b.name(); }  
  
int main(int argc, char* argv[]){  
    Base base;  
    Derived derived;  
    if (argc == 1)  
        print(base);  
    else  
        print(derived);  
}
```

Variable *derived*:

- **Static type:** Derived
- **Dynamic type:** Derived

Static vs Dynamic Types Example

```
struct Base { virtual string name(){ return "Base"; } };  
struct Derived : Base { string name(){ return "Derived"; } };  
  
void print(Base& b) { cout << b.name(); }  
  
int main(int argc, char* argv[]){  
    Base base;  
    Derived derived;  
    if (argc == 1)  
        print(base);  
    else  
        print(derived);  
}
```

Variable *b*:

- **Static type:** Base
- **Dynamic type:** either Base or Derived

Static vs Dynamic Types Example



```
struct Base {    virtual string name(){ return "Base";    } };  
struct Derived : Base { string name(){ return "Derived"; } };  
  
void print(Base& b) { cout << b.name(); }  
  
int main(int argc, char* argv[]){  
    Base base;  
    Derived derived;  
    print(base);  
    print(derived);  
}
```

Variable *b*:

- **Static type:** Base
- **Dynamic type:** Base, then Derived


Declarations & Definitions

A **declaration** introduces an identifier and binds its static type.


It is what the compiler needs to accept references to that identifier.

A **definition** instantiates/implements an identifier.

It's what the linker needs in order to link references to those entities.



```
extern int bar;  
extern int g(int, int);  
double f(int, double);  
class foo;
```



```
int bar;  
int g(int a, int b) {return a * b;}  
double f(int x, double y) {return x+y;}  
class foo {};
```

Note that a definition can also be a declaration.


Declarations & Definitions

An identifier can be **declared** as often as you want.



```
int foo(int i);  
int foo(int i);  
  
int foo(int i){ return i*2; }
```

An identifier must be **defined** exactly once!



```
int foo(int i){ return i*2; }  
  
int foo(int i){ return i*2; }  
// compilation error
```

One Definition Rule

- No translation unit shall contain more than one definition of any variable, function, class type, enumeration type or template.
- Every program shall contain exactly one definition of every non-inline function or variable that is used.
- Exactly one definition of a class is required if any translation unit where the class is used, requires it to be complete.

No definition: unknown what to do.

Multiple definitions: unknown which to choose.