Software Languages Lab

Structuur van Computerprogramma's 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/

Vrije Universiteit Brussel - Faculty of Science and Bio-Engineering Sciences - Computer Science Department - Pleinlaan 2 - 1050 Brussels - Belgium

Software Languages.Lab

Chapter 1 - Introduction

Structuur van Computerprogramma's 2

C++ Facts & Figures

Scheme vs. C++

```
(define (fac n)
  (if (= 0 n)
    1
    (* n (fac (- n 1)))))

>>>fac
```

```
(fac 5)


>>>120
```

A simple matter of getting used to the syntax?

... not really

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

int fac(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * fac(n - 1);
   }
}

int main() {
   cout << fac(5);
   return 0;
}
```

fac.cpp

```
g++ -Wall -o fac fac.cpp
./fac
>>>120
```

C++ Design Principles & Characteristics

- **Statically typed** general-purpose programming language
- **Preprocessor**, **Compiler** and **Linker**
- Focus on **run-time efficiency**
  - Zero-overhead principle
- Focus on **portability**
- Focus on **interoperability**
- **Multiparadigm** programming support
  - structured programming, object-oriented programming, generic programming, data abstraction

> *"C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming"*
>
> *B. Stroustrup, A History of C++: 1979-1991*

Memory Management?

> The possibility of automatic garbage collection was considered on several occasions before 1985 and deemed unsuitable for a language already in use for real-time processing and hard-core systems tasks such as device drivers. In those days, garbage collectors were less sophisticated than they are today and the processing power and memory capacity of the average computer were small fractions of what today's systems offer. My personal experience with Simula and reports of other GC-based systems convinced me that GC was unaffordable by me and my colleagues for the kind of applications we were writing. Had C with Classes (or even C++) been defined to require automatic garbage collection it would have been more elegant, but stillborn.

[STROU93]

Option I : Manual memory management

- Be careful and mindful, don't introduce memory leaks !

Option II : Use a library-based approach (e.g., libgc)

- Automated protection against memory leaks
- No free's and delete's in your program

More about this later...

GCC - GNU Project Compiler Collection

Terminal — grotty — 87×27

```
GCC(1)                          GNU                          GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] [@file] infile...

       Only the most useful options are listed here; see below for the
       remainder.  g++ accepts mostly the same options as gcc.

       In Apple's version of GCC, both cc and gcc are actually symbolic li
       to a compiler named like gcc-version.  Similarly, c++ and g++ are
       links to a compiler named like g++-version.

       Note that Apple's GCC includes a number of extensions to standard G
       (flagged below with "APPLE ONLY"), and that not all generic GCC
       options are available or supported on Darwin / Mac OS X.  In
       particular, Apple does not currently support the compilation of
       Fortran, Ada, or Java, although there are third parties who have ma
:
```

Terminal — grotty — 87×27

```
OPTIONS
       Option Summary

       Here is a summary of all the options, grouped by type.  Explanations
       are in the following sections.

       Overall Options
           -c  -S  -E  -o file  -combine -pipe  -pass-exit-codes -ObjC (APPLE
           ONLY) -ObjC++ (APPLE ONLY) -arch arch (APPLE ONLY) -Xarch_arch
           option (APPLE ONLY) -fsave-repository=file -x language  -v  -###
           --help  --target-help  --version @file

       C Language Options
           -ansi  -std=standard  -fgnu89-inline -aux-info filename -faltivec
           (APPLE ONLY) -fasm-blocks (APPLE ONLY) -fno-asm -fno-blocks
           -fno-builtin  -fno-builtin-function -fhosted  -ffreestanding
           -fopenmp -fms-extensions -trigraphs  -no-integrated-cpp
           -traditional  -traditional-cpp -follow-single-precision
           -fcond-mismatch -flax-vector-conversions -fconstant-cfstrings
           (APPLE ONLY) -fnon-lvalue-assign (APPLE ONLY)
           -fno-nested-functions -fpch-preprocess (APPLE ONLY)
           -fsigned-bitfields  -fsigned-char -Wno-#warnings (APPLE ONLY)
           -Wextra-tokens (APPLE ONLY) -Wnewline-eof (APPLE ONLY)
           -Wno-altivec-long-deprecated (APPLE ONLY)
           -fglobal-alloc-prefer-bytes (APPLE ONLY)
           -fno-global-alloc-prefer-bytes (APPLE ONLY) -funsigned-bitfields
:
```

Take a look at the man pages of gcc:

man gcc

g++ is the same as **gcc** but handles the input as C++
(i.e. you can achieve the same result with **gcc** but then you need to set the correct options manually)

Example IDE's (1)

Terminal — nano — 88×29

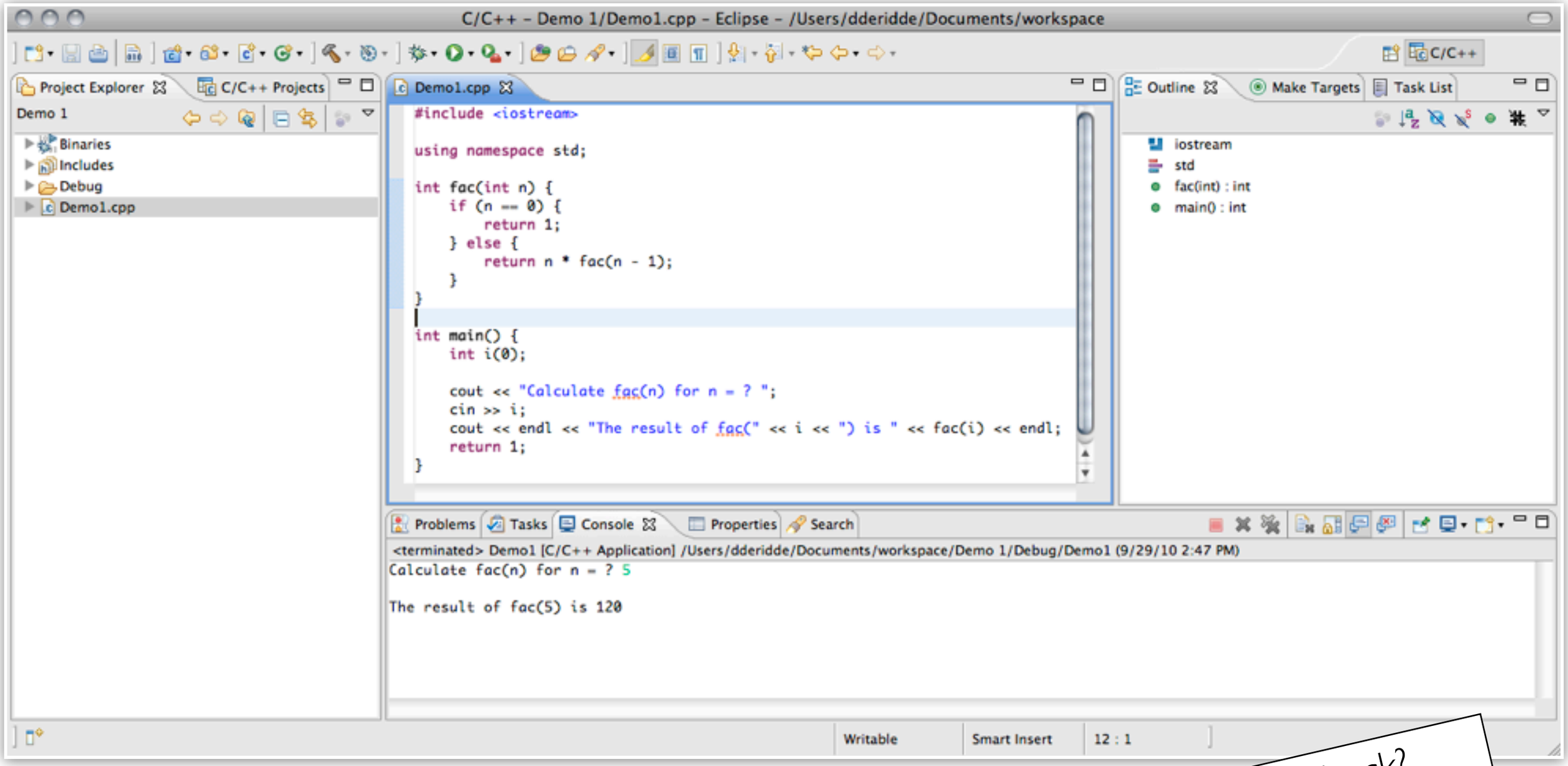
```
 GNU nano 2.0.6                File: demo1.cpp

// Compile using g++ -v -o demo1 demo1.cpp
// Run using ./demo1

#include <iostream>

using namespace std;

int fac(int n){
  if(n==0) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}

int main() {
 int i(0);

 cout << "Calculate fac(n) for n = ? " ;
 cin >> i;
 cout << endl << "The result of fac(" << i << ") is " << fac(i) << endl;
 return 1;
}


^G Get Help    ^O WriteOut    ^R Read File    ^Y Prev Page    ^K Cut Text    ^C Cur Pos
^X Exit        ^J Justify     ^W Where Is     ^V Next Page    ^U UnCut Text  ^T To Spell
```

Terminal — bash — 86×9

```
Balderwiz:Demo1 dderidde$ g++ -Wall -o demo1 demo1.cpp
Balderwiz:Demo1 dderidde$ ./demo1
Calculate fac(n) for n = ? 5

The result of fac(5) is 120
Balderwiz:Demo1 dderidde$
```

What's the benefit/drawback?

Example IDE's (2)



What's the benefit/drawback?

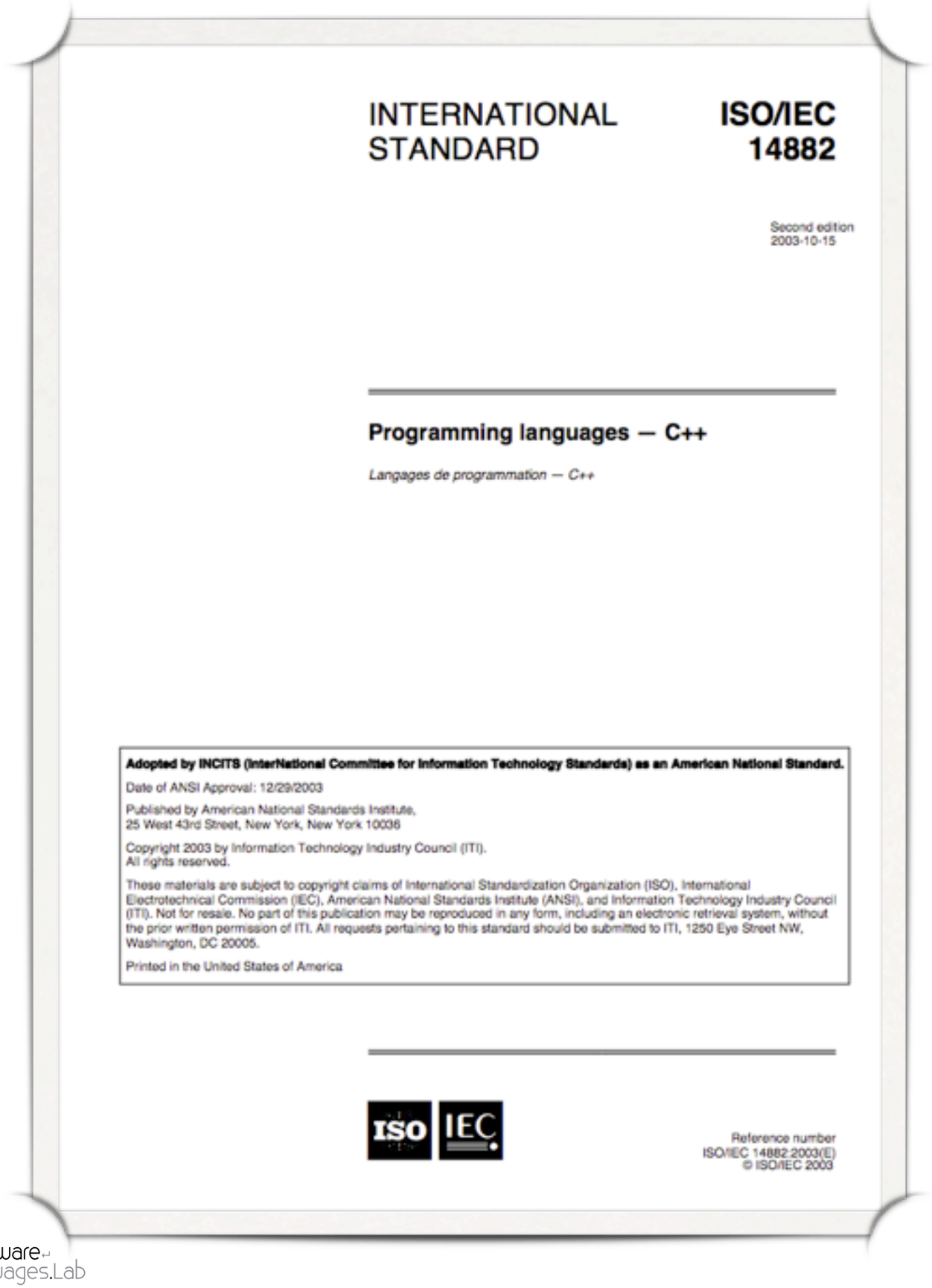
Brief C++ History

- Originally designed and implemented by Bjarne Stroustrup
  - 1979: start at AT&T Bell Labs, basis was **C**
  - 1980: "**C with Classes**" TR in SIGPLAN Notices
  - 1983: officially named **C++**
  - 1985: first commercial version & book
  - late 80ies: templates & exception handling
  - 1998: standard **ISO/IEC 14882 + STL**
  - 2003: corrigendum ISO/IEC 14882
  - Today: standard **C++0x**

Read [STROU93] for a good insight in C++ design considerations

http://www2.research.att.com/~bs/homepage.html

ISO/IEC 14882:2003



INTERNATIONAL STANDARD

ISO/IEC 14882

Second edition
2003-10-15

Programming languages — C++

Langages de programmation — C++

Adopted by INCITS (InterNational Committee for Information Technology Standards) as an American National Standard.

Date of ANSI Approval: 12/29/2003

Published by American National Standards Institute,
25 West 43rd Street, New York, New York 10036

Printed in the United States of America

ISO IEC

Reference number
ISO/IEC 14882:2003(E)

- Standard used in this course
- Reference work
- 786 pages !
  - **Save a tree, don't print it**

ISO
International Standard Organization

IEC
International Electrotechnical Commission

ANSI
American National Standard Institute

What's the benefit/drawback?

Started in 1979? C++ is old so why should I care...

source - http://www.tiobe.com (1-sep-2010)

Position Aug 2010	Position Aug 2009	Delta in Position	Programming Language	Ratings Jul 2010	Delta Jul 2009	Status
1	1	=	Java	17.994%	-1.53%	A
2	2	=	C	17.866%	+0.65%	A
3	3	=	C++	9.658%	-0.84%	A
4	4	=	PHP	9.180%	-0.21%	A
5	5	=	(Visual) Basic	5.413%	-3.07%	A
6	7	↑	C#	4.986%	+0.54%	A
7	6	↓	Python	4.223%	-0.27%	A
8	8	=	Perl	3.427%	-0.60%	A
9	19	↑↑↑↑↑↑↑↑↑↑	Objective-C	3.150%	+2.54%	A
10	11	↑	Delphi	2.428%	+0.0	
11	9	↓↓	JavaScript	2.401%	-0.4	
12	10	↓↓	Ruby	1.979%	-0.5	
13	12	↓	PL/SQL	0.757%	-0.2	
14	13	↓	SAS	0.715%	-0.1	
15	20	↑↑↑↑↑	MATLAB	0.627%	+0.0	B
16	18	↑↑	Lisp/Scheme/Clojure	0.626%	0.00%	B
17	16	↓	Pascal	0.622%	-	
18	15	↓↓↓	ABAP	0.616%	-	
19	14	↓↓↓↓↓	RPG (OS/400)	0.606%	-	
20	-	↑↑↑↑↑↑↑↑↑↑	Go	0.603%	0.00%	B

C(++) top-ranked !

source - http://sourceforge.net (1-sep-2010)

Programming Language

Java	(395)
C++	(4,776)
PHP	(177)
C	(2,383)
C#	(550)
Python	(342)
Show all	

sourceforge

C++ for open source development !

Category	Ratings Aug 2010	Delta Aug 2009
Object-Oriented Languages	54.9%	+0.6%
Procedural Languages	40.5%	-1.0%
Functional Languages	3.1%	+0.2%
Logical Languages	1.5%	+0.2%

C++ is multi-paradigm !

Category	Ratings Aug 2010	Delta Aug 2009
Statically Typed Languages	62.2%	+2.5%
Dynamically Typed Languages	37.8%	-2.5%

C++ is statically typed !

Well-known C++ Applications (*)

source - B. Stroustrup - june 2010 - http://www2.research.att.com/~bs/applications.html

Unix CDE Desktop

Mozilla Firefox

Google Chromium

MacOS X Finder

iPod GUI

Amazon.com

Bloomberg

Adobe Acrobat

Mozilla Thunderbird

Doxygen

AppleWorks

Linux KDE

Autodesk

Windows XP

BeOS

MySQL

SETI@home

CERN Data Analysis

Adobe Illustrator

Adobe Photoshop

Internet Explorer

ILOG

NASA JPL Mars Rover

Sun HotSpot Java VM

...

(*) no guarantees about the accuracy, some applications are only partially written in C++

Objectives (Recap)

- Study a language **closer to the hardware and to the operating system**
  - Program in a language which requires a **profound knowledge of all the rules and exceptions**
  - **No overprotective programmer parenthood** enforced by the language
    - Unsafe areas are open for exploration,
    - Reckless and sloppy programmers will shoot their own feet (yes, both of them!)
      - **Debugging beyond "get the algorithm right"**
        - Instead a focus on **"get the right code for the algorithm"**

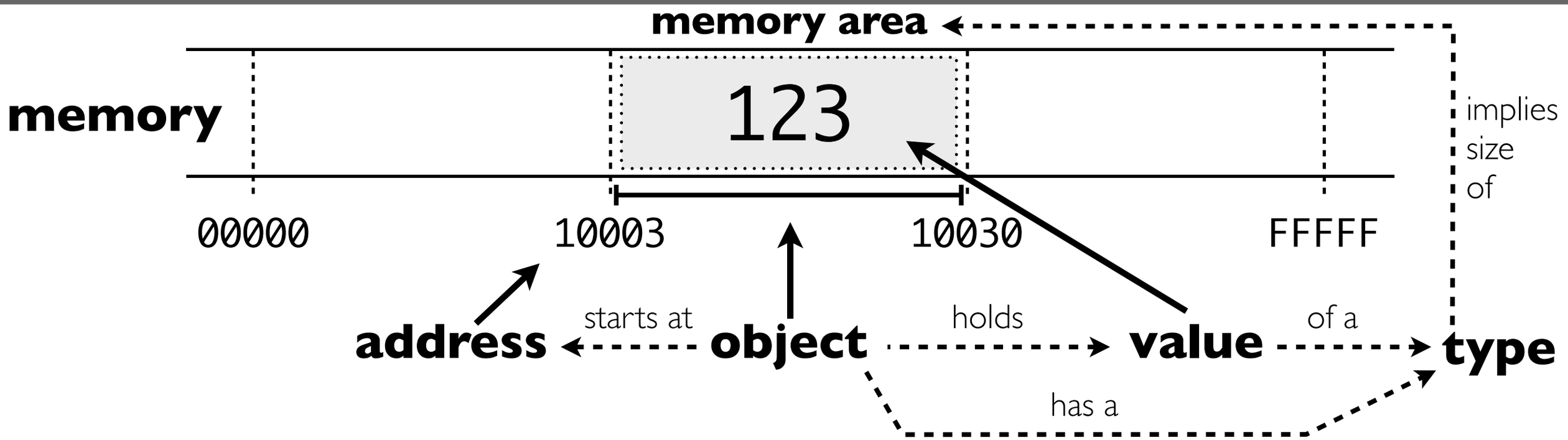
- Introduce **different programming paradigms**
  - **Structured** programming
  - **Object-Oriented** programming
  - **Generic** programming
- Study an **industry-strength language**

C++ Fundamentals

Overview of Concepts

- Objects, values, and types
- Variables, constants, and expressions
- Lvalue, rvalue
- Object definition, and object manipulation
- Reference variable
- Function definition, function body, formal parameters, actual parameters, return type, return value
- Function calling, function call frame
- Parameter passing, call by value, call by reference
- Identifiers, keywords

Objects, Values, and Types



- An **object** is a memory area with an **address**
- The contents of an object is a **value**
- An object has a **type** (e.g., `int`) restricting the set of allowed values
- A type implies a **size** used to reserve the amount of memory
  - `sizeof(`object`)` = `sizeof(`type`)` = # of bytes to store the object
- A type supports a set of **operations** (e.g., **+**)
- To create an object of a certain type, you need to:
  - Allocate an object (memory area) of the corresponding size
  - Fill the object with the initial value

Variables, Constants, and Expressions

- How to **reference** a value?
  - A **variable** which refers to an object that holds a value

  - A **constant** which refers to a value

```
123
```

  - An **expression** which refers to a value

```
x + 123
```

- Two kinds of values:
  - An **lvalue** refers to a value that has an address

```
x
```

  - An **rvalue** refers to a value without an address

```
x + 123
```

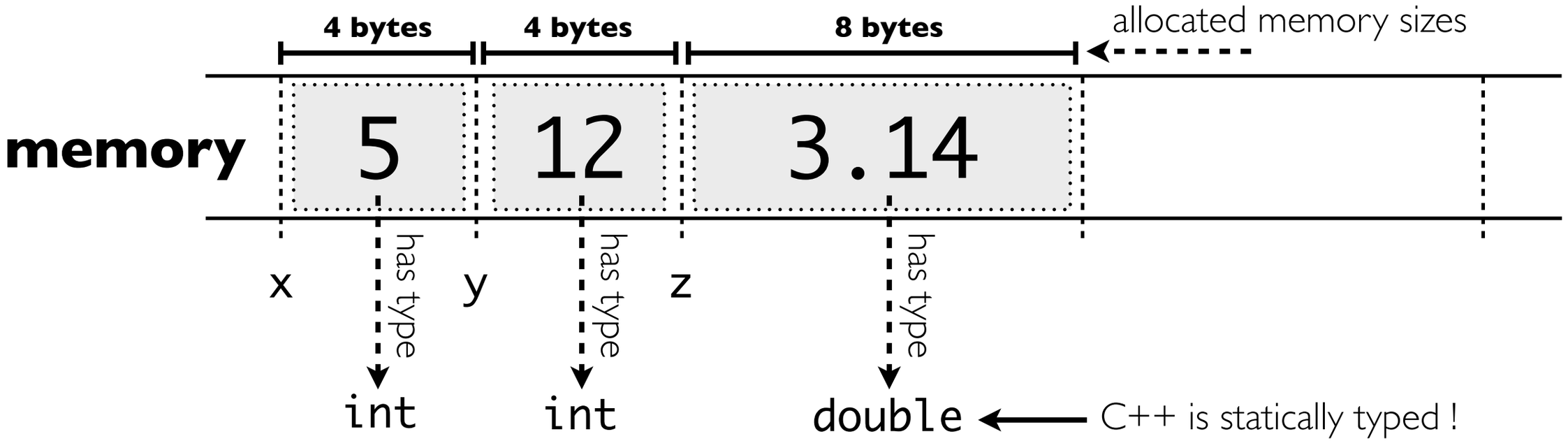
Object Definition

```
NameOfType NameOfVariable(InitialValue);
```

```
int x(5);
int y(x+7);
double z(3.14);
```

initial value provided by a **constant**

initial value provided by an **expression**



Alternative definitions:

```
int x(5), y(12);
double z = 3.14;
```

multiple definitions on one line

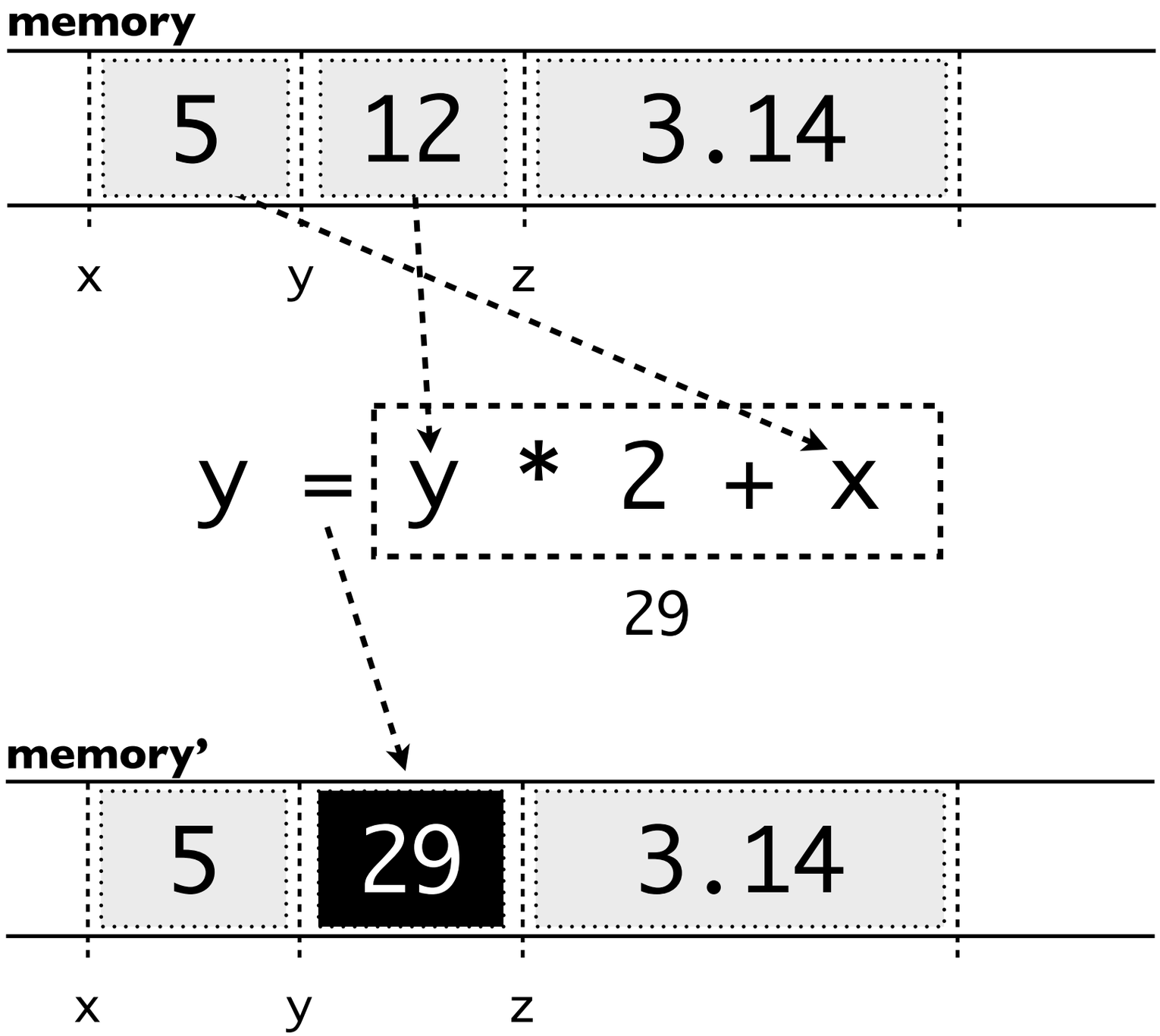
use of = for assigning initial value

Object Manipulation: Assignment

```
LeftExpression = RightExpression
```

Note: the `LeftExpression` must always evaluate to an **lvalue**

```
int x(5);
int y(x+7);
double z(3.14);

y = y * 2 + x;
```



Reference Variables (1)

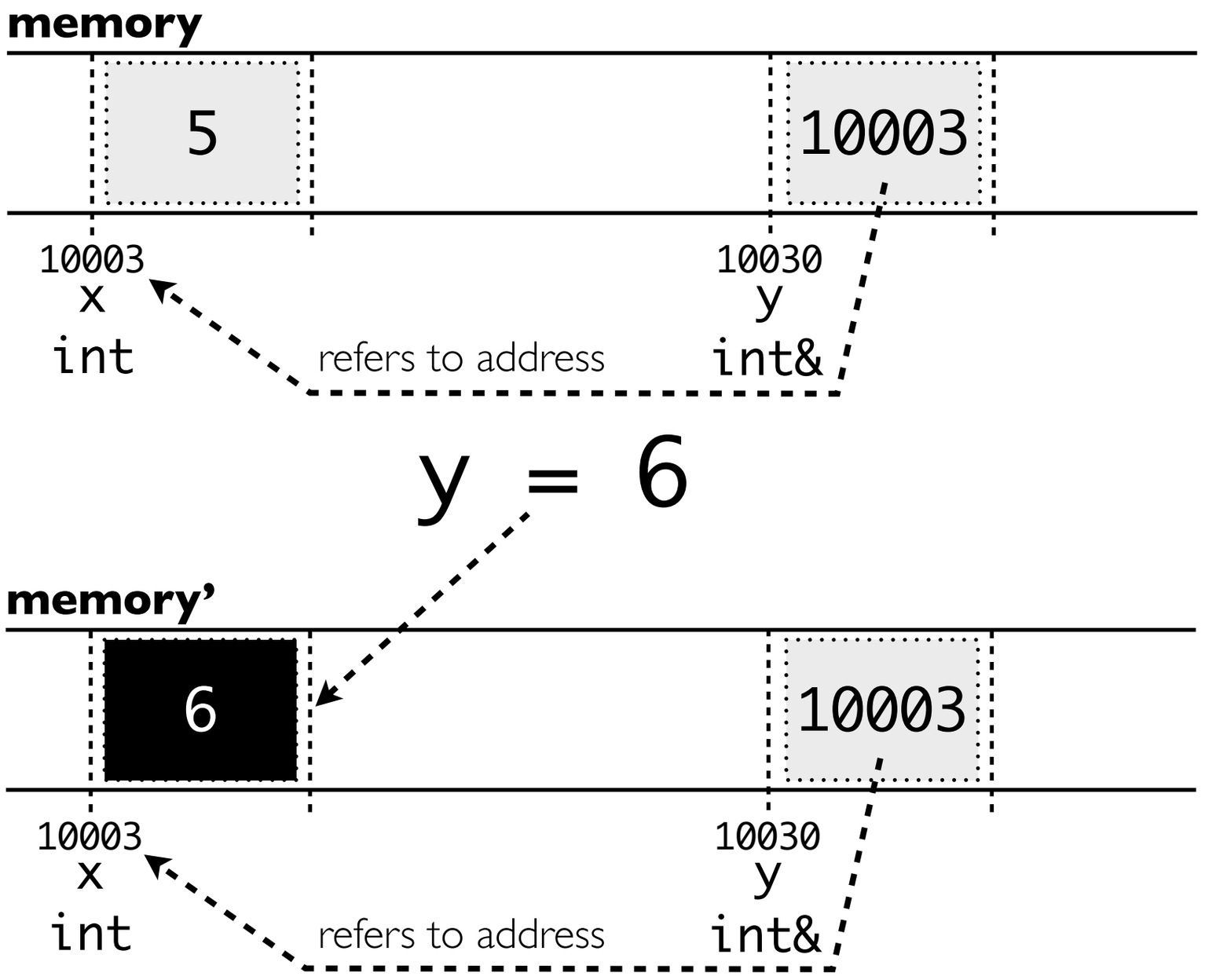
`NameOfType& NameOfVariable(Expression);`

Note: the `Expression` must yield an **lvalue** of type `NameOfType`

```
int x(5);
int& y(x);

y = 6;
```

A **reference variable** holds the address of another object as its value

The address itself is not accessible to the programmer, the compiler handles the "referencing"



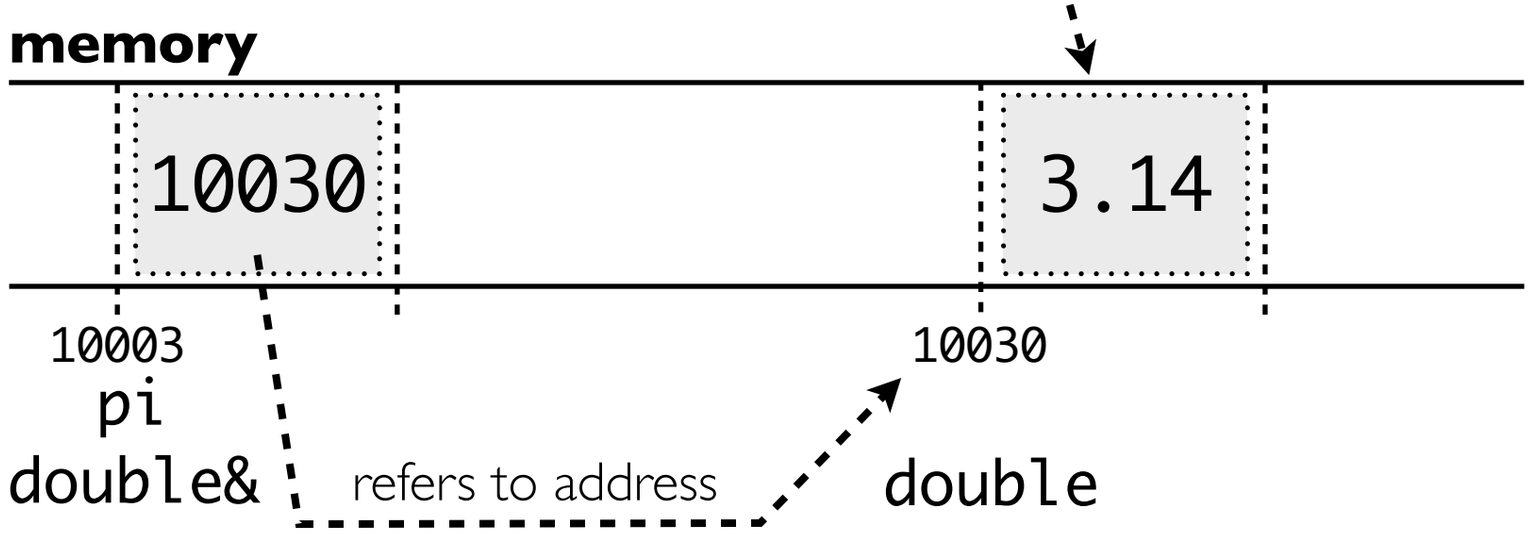
Reference Variables (2)

```
NameOfType& NameOfVariable(Expression);
```

What if `Expression` **yields an rvalue?**

```
double& pi(3.14);
```

1. A fresh object containing the value is created
2. NameOfVariable will refer to this object

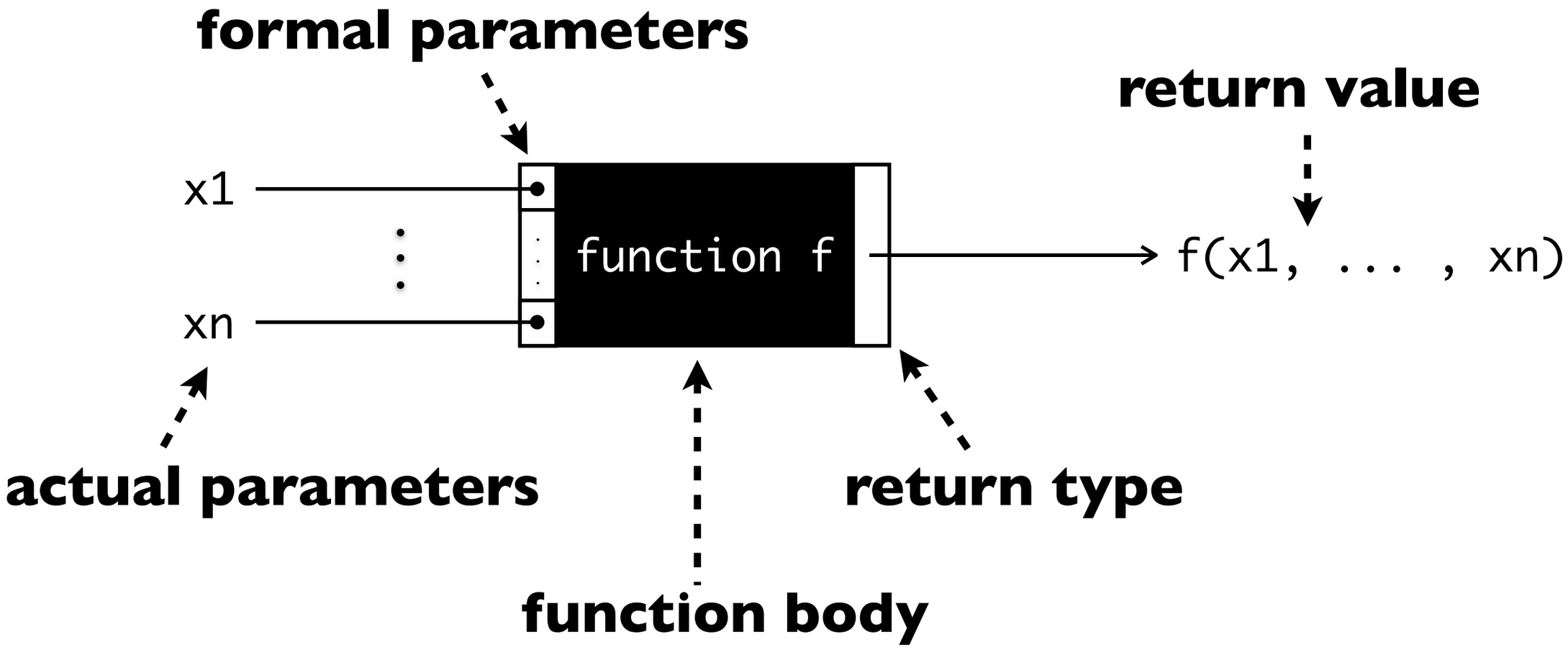


What if the object referenced belongs to a different scope?

This is allowed, see functions and parameter passing (a.k.a. **call-by-reference**)

Careful with reference variables and deallocated memory (see later)

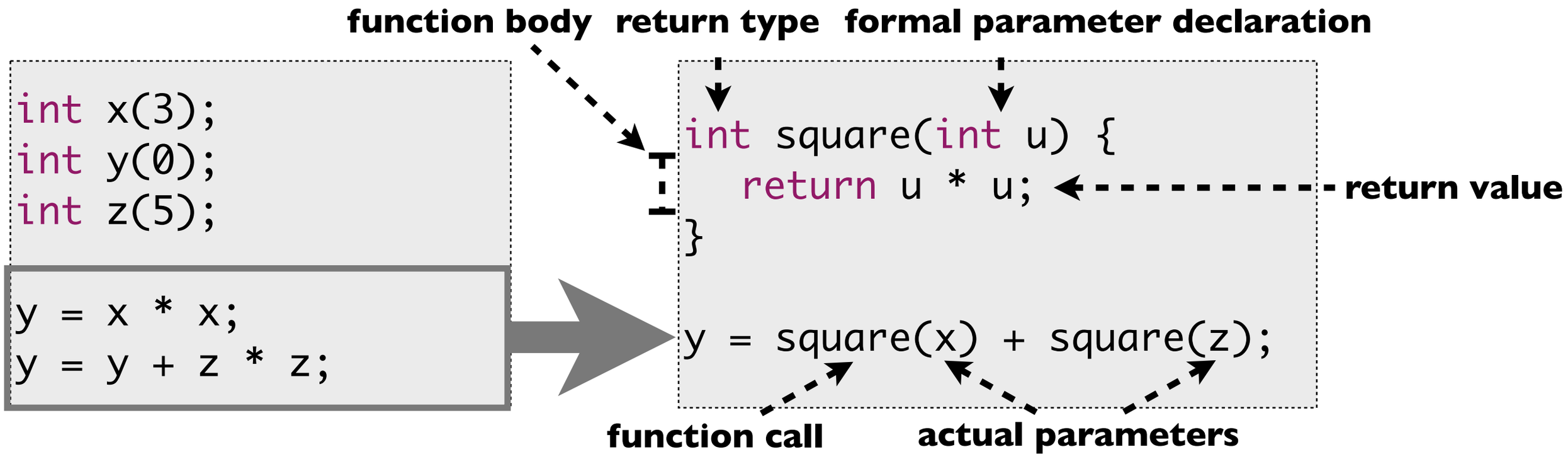
Functions



- Basic means for modularising an implementation
  - Structured programming
- Reuse instead of code duplication
  - Decrease code size and eliminate potential duplication of errors: easier to maintain!

Function Definition

```
ReturnType
FunctionName(FormalParamDeclarationList){
     StatementList
}
```



Function Calling and Function Call Frames

```
int x(3);
int y(0);
int z(5);

int square(int u) {
   return u * u;
}

y = square(x + 1);
```

function call

1. Initialize a fresh **local parameter object** based on the formal parameter declaration and store the value of the actual parameter expression as its value

```
int u(4);
```

2. Execute the statements of the **function body** within a **function call frame** (environment) containing local (`u`) and non-local names (`x, y, z`)

3. The return statement is evaluated by creating a fresh object which is initialized with the value of the associated expression

```
int tmp
```

4. The caller processes the returned object

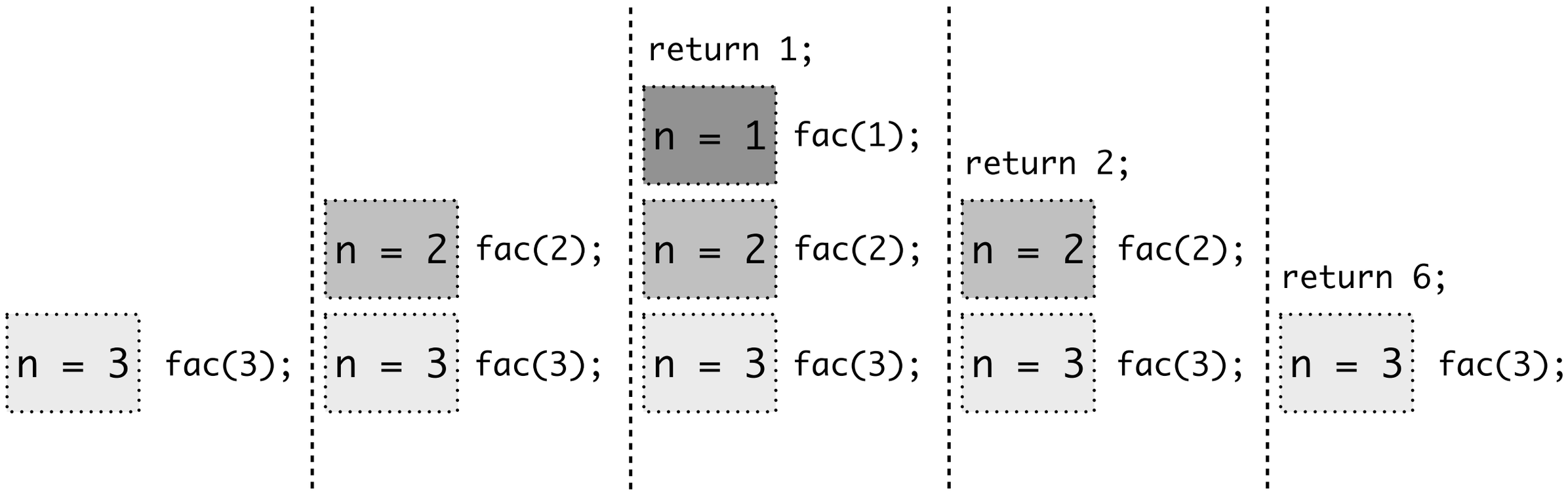
```
y = 16;
```

5. Deallocate the function call frame

Recursion: Function Call Frame Stack

```
int fac(int n) {
   if (n < 2)
      return 1;
   else
      return n * fac(n - 1);
}

fac(3);
```

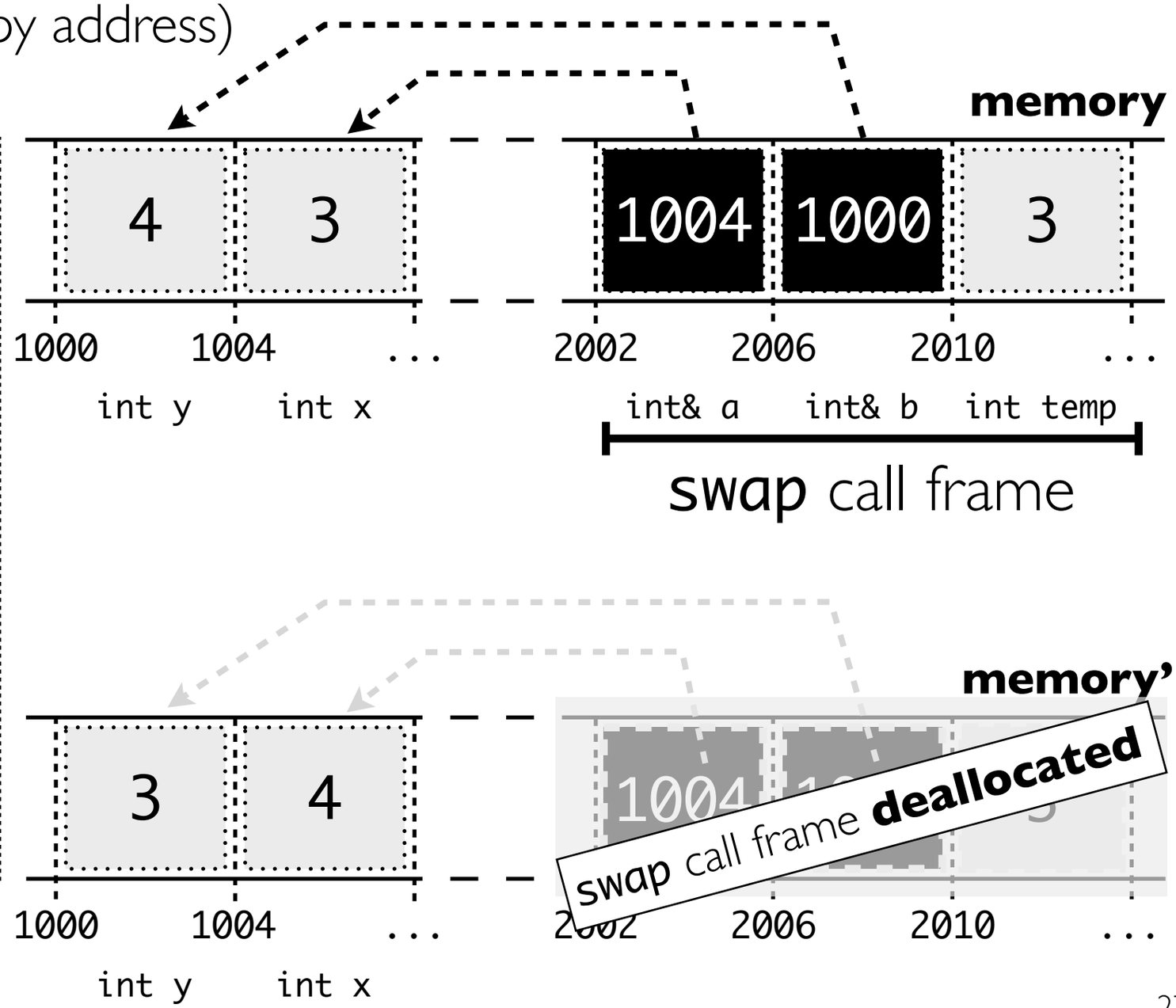
1. Every time you enter fac, initialize a fresh **local parameter object** based on the formal parameter declaration and store the value of the actual parameter expression as its value
2. Execute the statements of the **function body** within a **function call frame** (environment) containing local (n) and non-local names
3. The return statement is evaluated by creating a fresh object which is initialized with the value of the associated expression
4. The caller processes the returned object
5. Deallocate the function call frame



Function Calling: Parameter Passing (1)

- C++ only supports **call by value**, so the value of the actual parameters is always copied into the local parameter objects
- Problematic if you pass around large objects
- Call by value on parameters of reference type is equivalent to **call by reference** (a.k.a. call by address)

```
int x(3);
int y(4);


void swap(int& a, int& b) {

   int temp(a);
   a = b;
   b = temp;
}
swap(x, y);
```



Function Calling: Parameter Passing (2)

What happens if we execute the following program?

```
int x(3);
int y(4);
int& z(x);


int& square(int u) {
   int result(0);
   result = u * u;
   return result;
}
```

```
z = square(y);
```

```
swap(square(x), square(y));
```

Once the call frame of `square(x + 1)` is popped from the stack, **the reference to the memory location of the local variable is no longer trustworthy !**

A bug that is **difficult to trace/reproduce**

The compiler generates a **warning** (not an error) not to be ignored !

```
../src/mathFunctions.cpp:57: warning:
   reference to local variable 'result'
   returned
```

The Anatomy and Execution of a C++ Program

Overview of Concepts

- Translation unit
- Function and variable declarations
- Function and variable definitions
- Preprocessor, compiler, linker
- Header files, source files, object files, executable files
- Directives
- Macro processor, macro language
- #ifndef wrappers
- Controlling/guard macro's
- Object code relocation
- Scopes, namespaces
- Lexical scoping, dynamic scoping

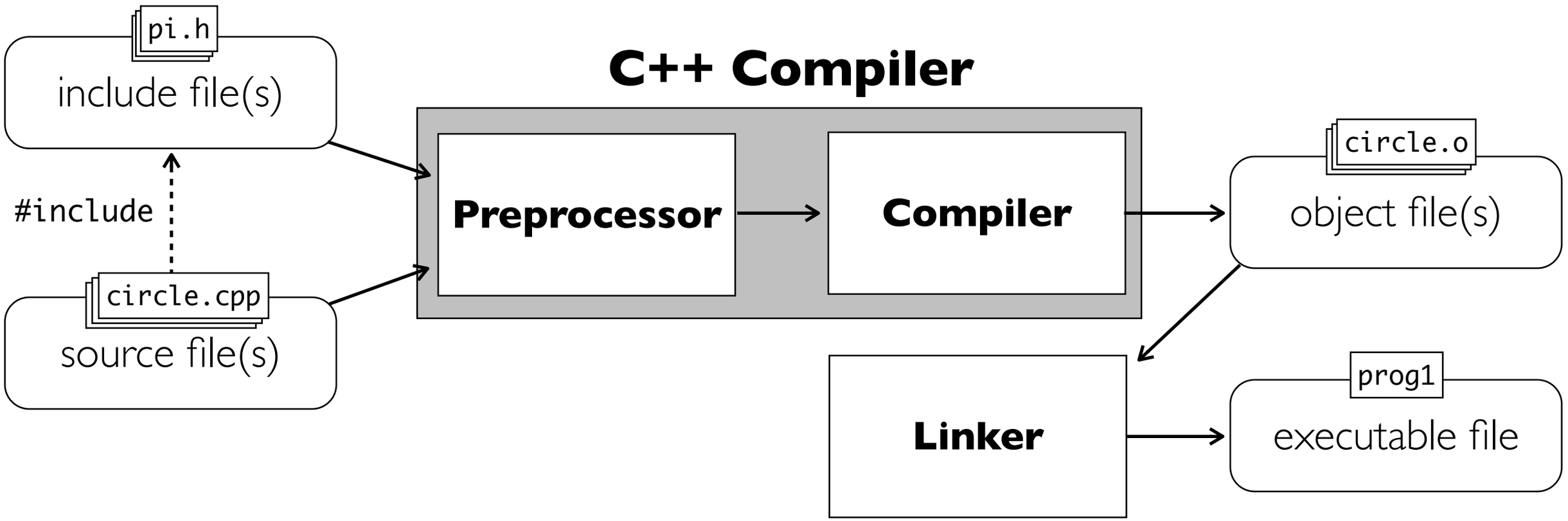
Program Organization

- A C++ program consists of **translation units**
- A translation unit contains **definitions** and/or **declarations of types, variables, and functions**
- A **declaration** tells the compiler about the existence of something so you can start to use its name (even before it is defined)

```
extern double pi;
double circle_area(double radius);
```

- A **definition** causes the compiler to:
  - Generate code for a function definition
  - Allocate memory for a variable definition
  - Compute the size and other properties of a newly defined type

The Compilation Process



- An **include file** (a.k.a. header file) contains **declarations** of types, variables, and functions defined in another translation unit
  - As such it declares the **interface** of a particular translation unit
- A **source file** contains **definitions** of types, variables, and functions
- The **preprocessor** is a macro processor that interprets **directives** such as `#include`, `#define`, `#ifdef`, `#ifndef`
- The **compiler** checks the program for errors and translates it to machine code (a.k.a. object code)
- The **linker** connects all the pieces together (e.g. references to libraries)

The C Preprocessor (a.k.a. CPP)

- Implements a **macro language** used to transform C++ programs before they are compiled
- The **macro processor** will make systematic text replacements in the input files provided
- The macro language provides a set of constructs to direct this transformation
  - **Object-like macros**: resemble data objects
    - Similar to a data-object that is being used in code
    - Commonly used to give symbolic names to constants
  - **Function-like macros**: resemble function calls
    - Similar to a function that is being called in code
    - Commonly used to expand larger portions of code

In-depth information at
http://gcc.gnu.org/onlinedocs/cpp/
Read it !

CPP: Object-like Macros

```
#define ObjectLikeMacroName TokenSequence
```

- A simple identifier that is replaced by a code fragment
- The code fragment is a.k.a. the **macro expansion**, **macro body**, or **macro replacement list**
  - There is no restriction, as long as it decomposes into valid preprocessing tokens
  - Non-recursive resolution of macro names
    - So if its name occurs in its body then that name will not be expanded
      - Note: other macro names in the body will be expanded !
    - Avoids infinite expansion of **self-referential macros**
- By convention macro names are uppercase

```
#define BUFFER_SIZE 1024
```

```
foo = (char *) malloc (BUFFER_SIZE);
```

≈

```
foo = (char *) malloc (1024);
```

CPP: Object-like Macros

```
#define NUMBERS 1, \
                2, \
                3
```

macro ends at end of `#define` line, but can be continued with `\` at end of line

```
int x[] = { NUMBERS};
```

≈

```
int x[] = {1, 2, 3};
```

```
foo = X;
#define X 4
bar = X;
```

≈

```
foo = X;
bar = 4;
```

When macro names are expanded, the macro's expansion replaces the invocation of the macro. After that the expansion is checked for containing macro names itself which are also expanded.

CPP: Function-like Macros

```
#define FunctionLikeMacroName() TokenSequence
```

- Function-like macros are only expanded if the macro invocation has parentheses

```
#define lang_init()  c_init()
```

```
lang_init()   ≈   c_init()
```

```
extern void foo(void);
#define foo() /* optimized version, no function call as body */
...
value = foo();
funcptr = foo;
```

expanded version

original version

What happens if you put a space between the name and the parentheses?

It is treated as an object-like macro definition whose body starts with `()`

CPP: Header Files

```
#include <filename>
```

Used for system header files, lookup starts in a standard list of system directories

```
#include “filename”
```

Used for header files of your own program, lookup starts in the current directory, then in the quoted directories, then in the standard list

- CPP will copy the contents of the header file into each source file that requires it
- Avoid redeclarations with a **“wrapper `#ifndef`”** (see example later)
- Benefits of working with header files:
  - Changes in the interface can occur in one place
  - All programs that use an interface will include a new version upon recompilation
  - Inconsistencies are avoided
- Naming convention: `myFile.h`

Program Organization: Circle Example (1)

- **Source files** (translation units) contain definitions
  - `circle.cpp` contains the **definition** of the `circle_area` function
  - `pi.cpp` contains the **definition** of the `pi` variable
  - `prog1.cpp` contains the **definition** of the `main` function
- **Include files** contain declarations
  - `circle.h` contains the **declaration** of the `circle_area` function
  - `pi.h` contains the **declaration** of the `pi` variable
- **File dependencies**
  - `prog1.cpp` needs `circle.h`
  - `circle.cpp` needs `pi.h` (and `circle.h`)
  - `pi.cpp` (needs `pi.h`)

Program Organization: Circle Example (2)

wrapper
`#ifndef`

controlling macro
(a.k.a. guard macro)

single line
comments `//`

```
#ifndef PI_H      // begin conditional inclusion:
// read body until #endif ONLY if PI_H is not defined


#define PI_H
// define variable PI_H, so next time ...


extern double pi; // declaration of variable pi


#endif                  // end of conditional inclusion
```

pi.h

`extern` indicates that the definition is elsewhere

Program Organization: Circle Example (3)

```
#include "pi.h"  // include declaration to ensure that
                 // declaration and definition of pi
                 // are consistent

double pi(3.14); // definition of variable pi
```

pi.cpp

What if the declaration is different from the definition?

If the declaration and definition are not consistent
then the compiler complains:

```
../pi.cpp:11: error: conflicting declaration 'float pi'
../pi.h:12: error: 'pi' has a previous declaration as 'double pi'
make: *** [pi.o] Error 1
```

Program Organization: Circle Example (4)

```c
#ifndef CIRCLE_H
#define CIRCLE_H
// The ifndef define .. endif sequence ensures
// that the following declaration is not read twice
// (see also pi.h)

// declaration of function 'circle_circumference'
double circle_circumference(double radius);

// declaration of function 'circle_area'
double circle_area(double radius);

#endif
```

circle.h

Program Organization: Circle Example (5)

```
#include "pi.h"      // declaration of pi
#include "circle.h" // declaration of functions
                    // that are defined in this file:
                    // including them ensures the consistency of
                    // the following definitions with the declarations

double              // function definition
circle_circumference(double radius) {
   return 2 * pi * radius;
}

double                  // function definition
circle_area(double radius) {
   return pi * radius * radius;
}
```

circle.cpp

Program Organization: Circle Example (6)

`iostream` declares standard stream objects for input/output

```
#include <iostream> // contains needed declarations,
                    // e.g. for operator<<(ostream&, X)

#include "circle.h" // contains needed declaration
                    // of circle_area

int                 // function definition
main() {
   double r(35);    // local variable definition
                    // r will live in the call frame
                    // the next expression writes the area of
                    // the circle to standard output

   std::cout << circle_area(r) << std::endl;

   return 0;        // all is well: return 0
}
```

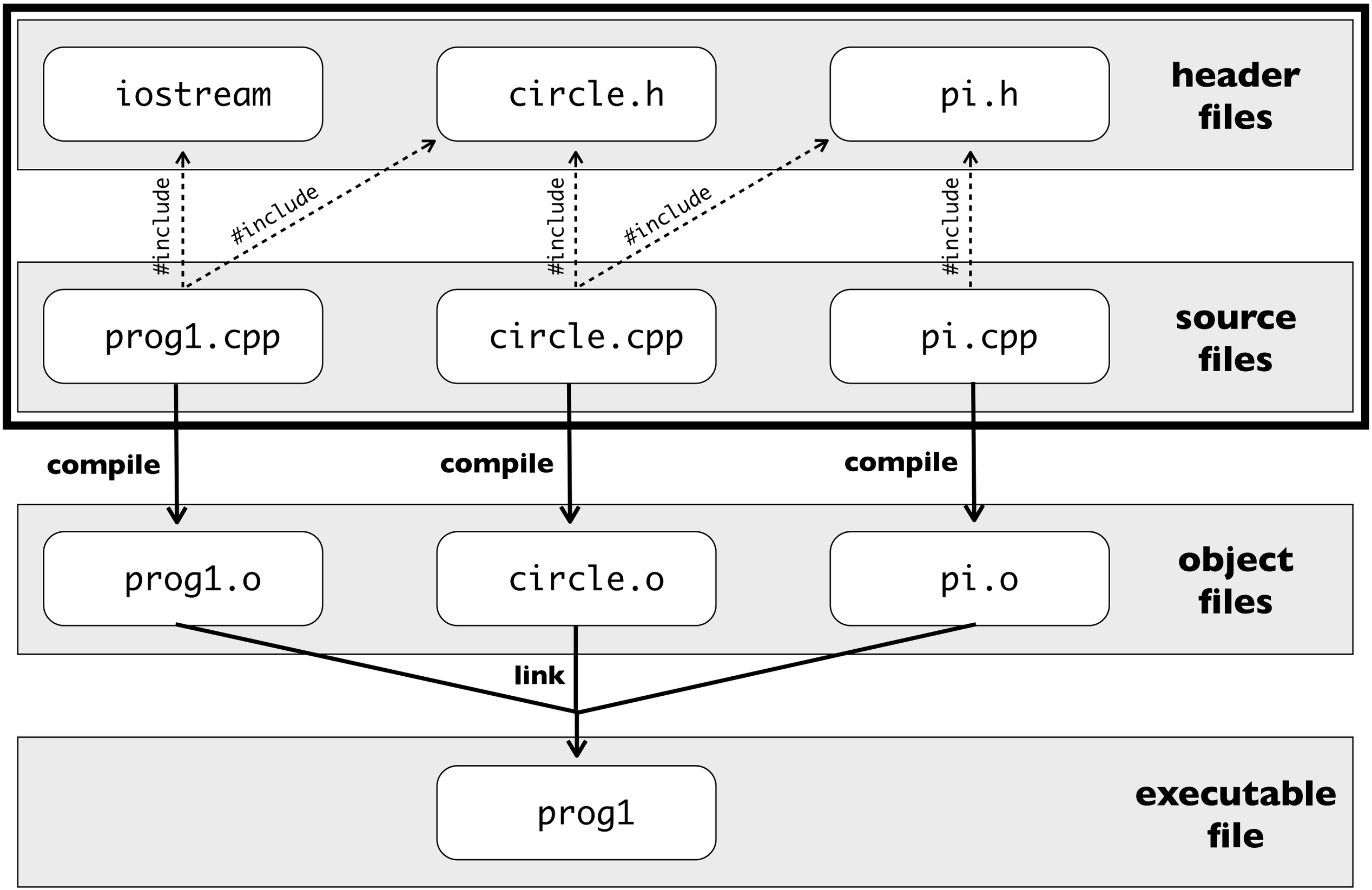
prog1.cpp

`std::cout` is the standard output stream (console)

`<<` is the insertion operator

`std::endl` represents an endline

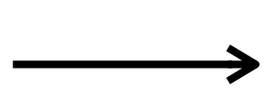
Program Organization: Circle Example (7)



Executing a C++ Program (1)

- When a C++ program is executed, the function **`main`** is automatically called
  - There can only be one **`main`** function in your program
- An **`int`** is returned when the program finishes its execution
  - By convention **`0`** means that the execution was without errors
  - Any other value indicates an abnormal program termination
- Outside the **`main`** function you can use the **`exit(int status)`** function to terminate the program
  - For abnormal termination only, otherwise use exception handling (later) 
- The return value is testable from the outside
  - For example from within a Linux shell:

```
./prog1 || echo " :error level"
```



```
0 :error level
```

Executing a C++ Program (2)

- A main function can have two arguments: **`argc`** and **`argv`**
  - This enables passing arguments when starting a program
    - **`argc`** : the number of arguments supplied
    - **`argc`** : the values of the arguments provided

`argc` and **`argv`** include the program name
`argv[0] = "/Users/dderidde/sayHello"`

An array, more about this later...

```cpp
#include <iostream>

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

   if (argc != 2) {
      std::cout << "Error: usage -> sayHello aName"<< std::endl;
   } else {

      std::cout << "Hello " << argv[1] << std::endl;
   }
   return 0;
}
```

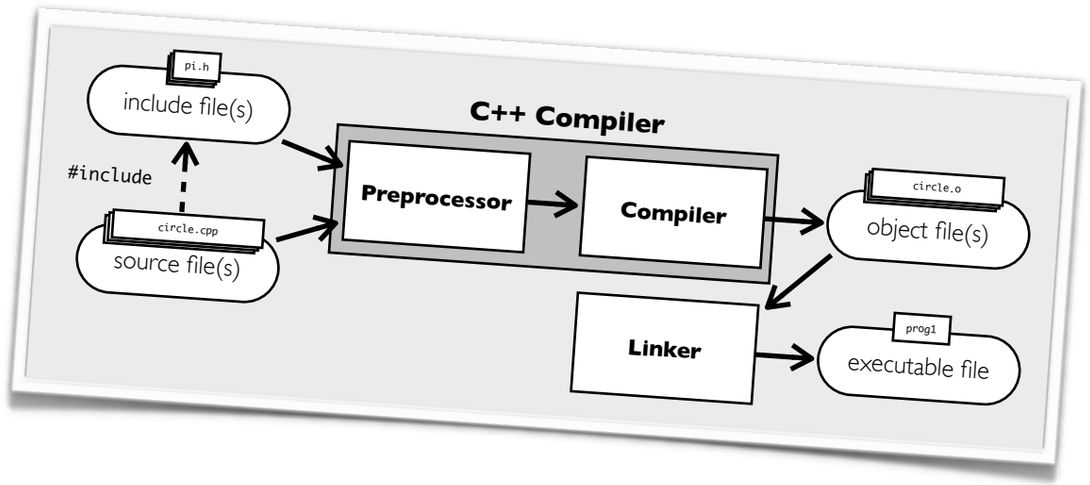
sayHello.cpp

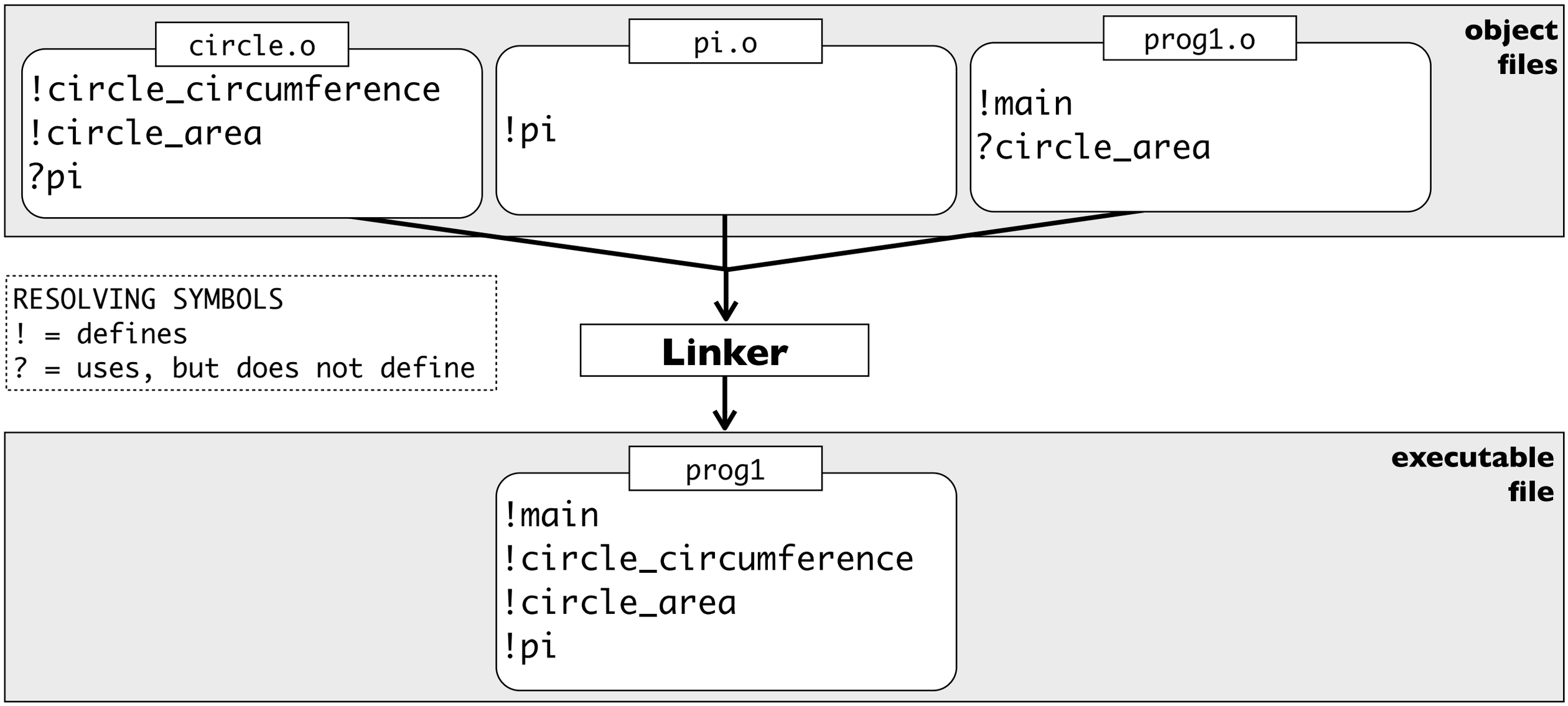
- For example from within a Linux shell:

```
./sayHello "Mister"   ⟶   Hello Mister
```

Linker

- Resolves symbols
  (e.g., `circle_area` in `prog1.o`)
- Relocates object code





Circle Example before Linking - prog1.o

prog1.o

```
!main
?circle_area
```

address in memory, still needs to be relocated

still needs to be resolved

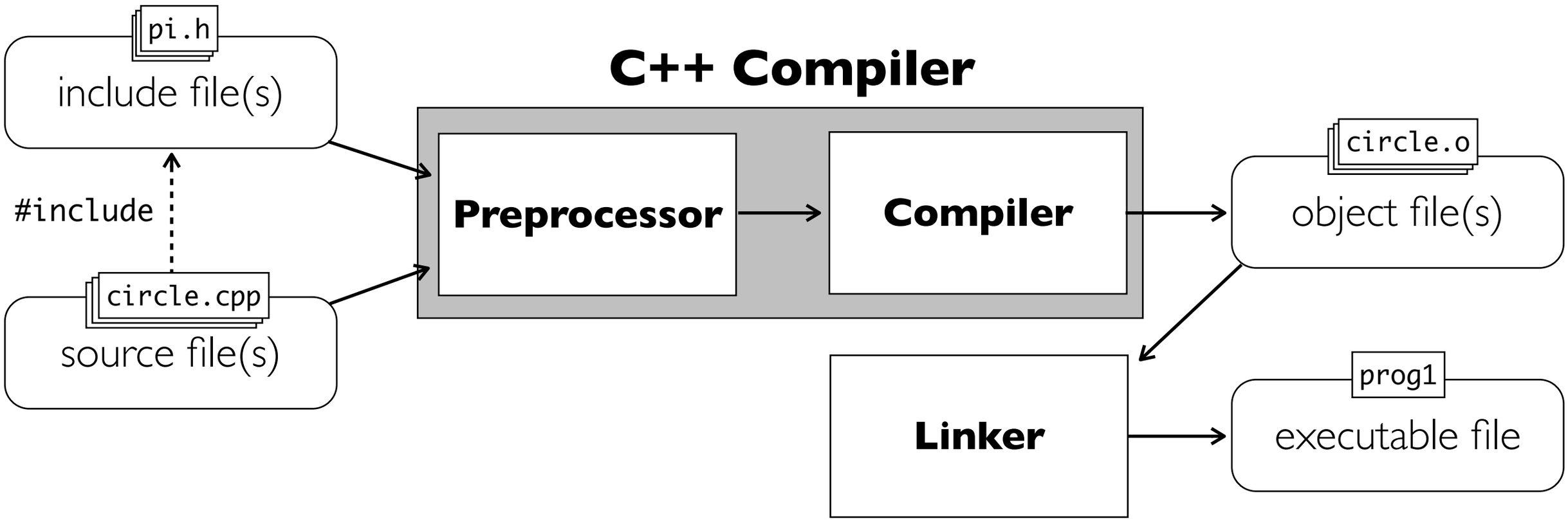
```
00000000 <main>:
      0:   55                      pushl   %ebp

...

     1a:    dd 1c 24               fstpl   (%esp,1)
     1d:    e8 fc ff ff ff         call    1e <main+0x1e>

...
```

Circle Example after Linking - prog1.o

- Resolves symbols
  (e.g., `circle_area` in `prog1.o`)
- Relocates object code

```
08048710 <main>:
8048710:  55                    pushl   %ebp

..

804872a:  dd 1c 24              fstpl   (%esp,1)
804872d:  e8 36 00 00 00        call    8048768 <circle_area(double)>

...

08048768 <circle_area(double)>:
8048768:  55                    pushl   %ebp
```

The Compilation Process: Overview



C++ Lexical Considerations

- Name of defined type/function/variable: **identifier:**
  - Start with letter or underscore _
  - Contains letters, digits, or underscore _
  - Convention: `MyClass`, `my_variable, my_simple_function`, `MY_CONSTANT`
  - Case sensitive: `student ≠ Student`
  - Not a **keyword** (see list on next slide)

Choosing good names is crucial for understandable programs

- Comments: `//` until end of line
  - Should be consistent with the code
  - Should have an added value

Important documentation
for code users and code maintainers (and yourself)

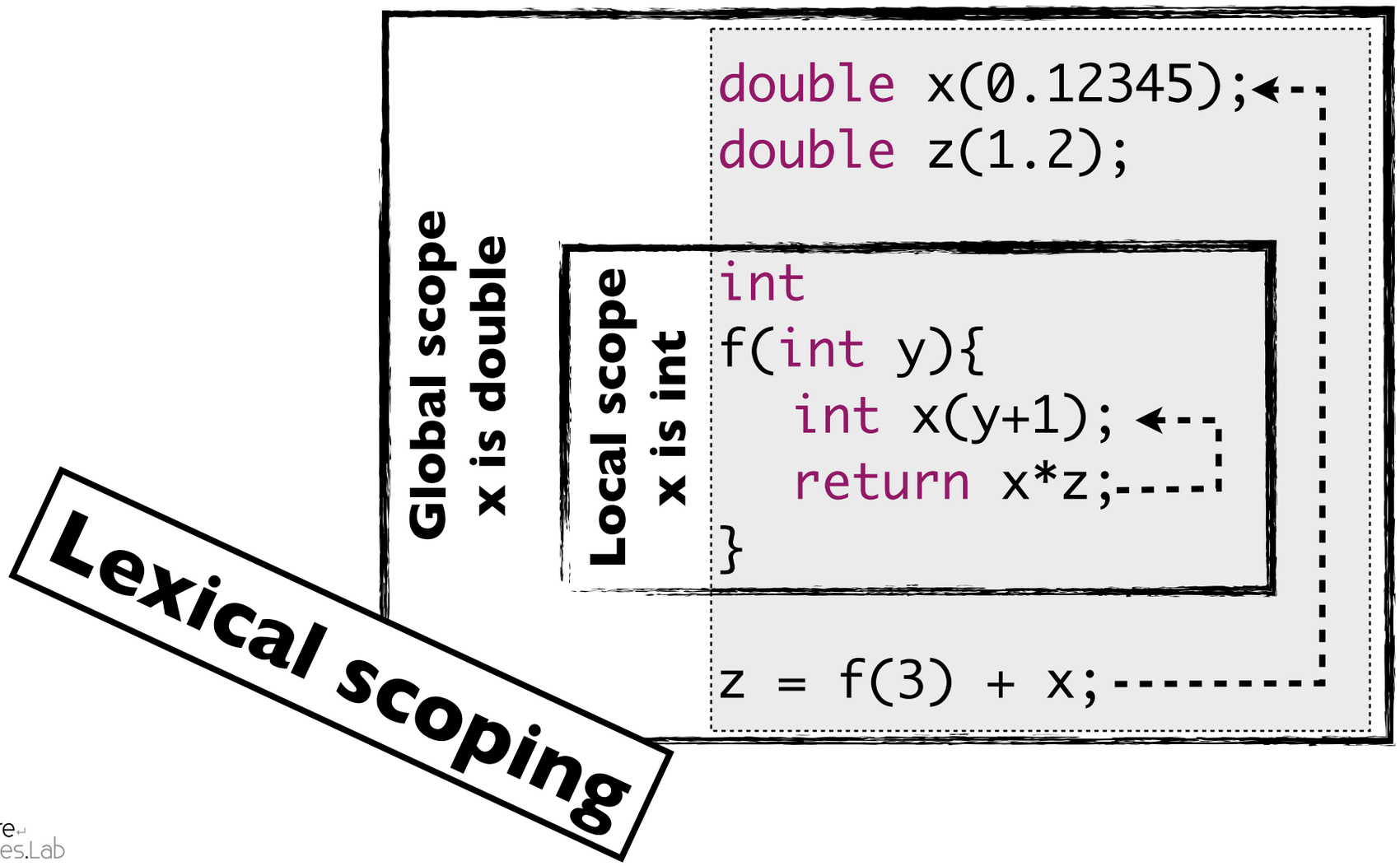
C++ Keywords

Keywords are reserved as part of the C++ language, they have a fixed meaning which cannot be changed by the programmer

asm	else	operator	throw
auto	enum	private	true
bool	explicit	protected	try
break	extern	public	typedef
case	false	register	typeid
catch	float	reinterpret_cast	typename
char	for	return	union
class	friend	short	unsigned
const	goto	signed	using
const_cast	if	sizeof	virtual
continue	inline	static	void
default	int	static_cast	volatile
delete	long	struct	wchar_t
do	mutable	switch	while
double	namespace	template	
dynamic_cast	new	this	

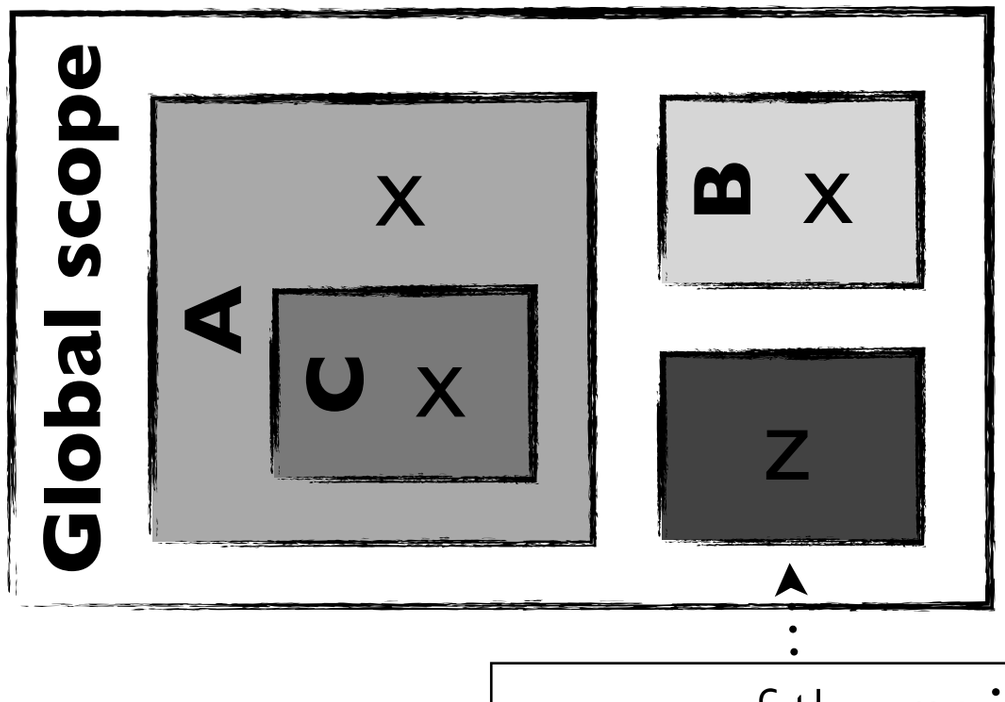
Scopes

- C++ programs may consist of thousands of translation units
  - Name conflicts may arise (e.g., when reusing a library)
- Names are organized hierarchically in (nested) **scopes**
- Some constructs, e.g., a class type or a function body automatically define a new scope (nested in the encompassing scope)



Namespaces: User-Defined Scopes

```
namespace A {
   int x;
   namespace C { int x; }
}

namespace B { int x; }

int
main() {
   int z(2);
   //error: which x?
   z = x;
}
```

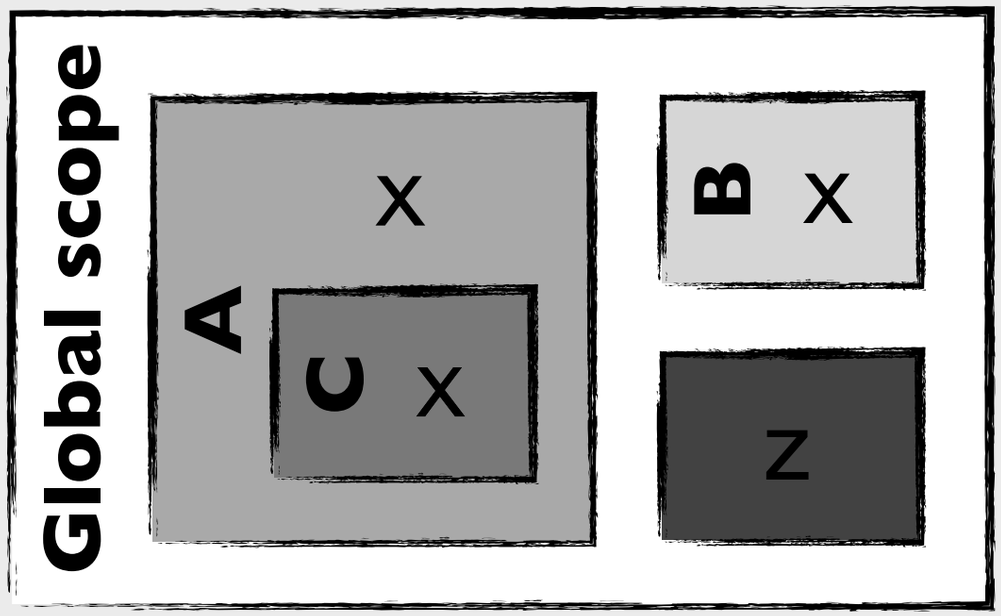


scope of the **main** function definition

```
// possible correction for z=x
z = A::C::x; // ok
// other possibility:
using A::C::x;
z = x;
```

Using Namespaces

```
namespace A {
      namespace C {
            extern int x;
            } // only a declaration
}

namespace B {
      int x;
}
```




```
int A::C::x(5); // definition of name "x" declared in ::A::C
namespace A {
      int x;
      } // continuation of namespace A

int main() {
   int z(2);
   using namespace A::C;
   z = x; // thus "x" refers to A::C::x
}
```

Recommended Reading

- **[STROU-93] A History of C++: 1979-1991**
  Bjarne Stroustrup, ACM HOPL-II, 1993
  http://www.research.att.com/~bs/hopl2.pdf

- **[STROU-07] Evolving a Language in and for the Real World: C++ 1991-2006**
  Bjarne Stroustrup, ACM HOPL-III, 2007
  http://www.research.att.com/~bs/hopl-almost-final.pdf

- **The C Preprocessor**
  http://gcc.gnu.org/onlinedocs/cpp/