C++ Program Design
-- Best Practice

Junjie Cao @ DLUT

Summer 2021

https://github.com/jjcao-school/c

Structure of a program

Statement语句

- **An statement is analogous to sentence.**
- We write sentences in order to convey an idea. We write statements in order to convey to the compiler that we want to perform a task.
- **Statements in C++ are terminated by a semicolon**.

- Many types:
  - Simple statements execute a single task

```
#include <iostream> // include statement, no ;
int x; // declaration statement
x = 5+3; // assignment statement
std::cout << x; // output statement
```

Compound statements

- Many types:
  - Simple statements execute a single task
  - **Compound statements or *blocks* are brace-enclosed sequences of statements.**

```
if (x > 5)              // start of if statement
{                       // start of block
    int n = 1;          // declaration statement
    std::cout << n; // expression statement
}                       // end of block, end of if statement
```

  - Note: a block is not terminated by a semicolon

Types of statements

1) expression statements;
2) compound statements;
3) selection statements;
4) iteration statements;
5) jump statements;
6) declaration statements;
7) try blocks;
8) atomic and synchronized blocks

Expression表达式

- An **expression** is a mathematical entity that evaluates to a value.
- An **expression** is analogous to **phrase**. 词组

- It is a sequence of ***operators*** and their ***operands***, that specifies a computation.

5+3

x=3

- Every expression yields a **result**.
  - In the case of an expression with no operator, the result is the operand itself, e.g., a literal constant or a variable.

4

"Hello World!"

- Statement is composed of one or multiple expressions.
- An expression followed by a semicolon is a statement.

cout<<"Hello World!“ => cout<<"Hello World!“;

Types of Expressions

- Primary expressions
- Scope resolution operator
- Postfix expressions
- Expressions with unary operators
- Expressions with binary operators
- Conditional operator
- Constant expressions
- Expressions with explicit type conversions
- Casting operators
- Run-time type information

Token 标记

- A token is analogous to **word**. It is the smallest element of a C++ program that is meaningful to the compiler.

- The C++ parser recognizes **these kinds of tokens**: identifiers, keywords, literals, operators, punctuators, and other separators.

100 // literal 字面量

::func // a global function

::operator + // a global operator function, 运算符

MyClass // a identifier 标识符

cout // a identifier

Types of Tokens

Token type	Description/Purpose	Examples
Keywords	Words with special meaning to the compiler	`int, double, for, auto`
Identifiers	Names of things that are not built into the language	`cout, std, x, myFunction`
Literals	Basic constant values whose value is specified directly in the source code	`"Hello, world!", 24.3, 0, 'c'`
Operators	Mathematical or logical operations	`+, -, &&, %, <<`
Punctuation/Separators	Punctuation defining the structure of a program	`{ } ( ) , ;`
Whitespace	Spaces of various sorts; ignored by the compiler	Spaces, tabs, newlines, comments

Token标记

- The parser separates tokens from the input stream by creating the **longest token possible** using the input characters in **a left-to-right scan**. Consider this code fragment:

```
a = i+++j;
```

- The programmer who wrote the code might have intended either of these two statements:

```
a = i + (++j) ;
a = (i++) + j ;
```

- Because the parser creates the longest token possible from the input stream, it chooses the second interpretation, making the tokens i++, +, and j.

Statement, expression & token

- **Statement is analogous to sentence.**
  - **A compound statement is analogous to paragraph. brace-enclosed**
  - Not every statement is an expression. It makes no sense to talk about the value of an **#include statement**, for instance.

- **Expression** is analogous to **phrase**.
- **Token** is analogous to **word**.

- sin(5+3) + cos(3); // expression statement
- Expressions: sin, cos, 5+3, 3, (5+3), sin(5+3), …
- Tokens: sin, +, 5, 3, (,)

Functions

- Statements are typically grouped into units called functions.
  - A **function** is a collection of statements that executes sequentially.

- Every C++ program must contain a special function called **main**.
  - When the C++ program is run, execution starts with the first statement inside of function main.

- Functions are typically written to do a very specific job.
  - For example, a function named “max” might contain statements that figures out which of two numbers is larger.
  - A function named “calculateGrade” might calculate a student’s grade.
  - We will talk more about functions later.

Classes / Types

- Contains many functions
- Combine data and the functions
- One class per file: A.h + A.cpp

Libraries and frameworks

- A **library** is a collection of functions that serves one particular purpose.
  - For example, if you were writing a game, you'd probably want to include a sound library and a graphics library.
  - Namespace, directory
- A **framework** is a collection of patterns and **libraries** to help with building an application.

- The C++ core language is actually very small and minimalistic.
- However, C++ also comes with a library called the **C++ standard library** that provides additional functionality for your use.
  - iostream, containers, algorithms, …

- OpenGL, QT, …

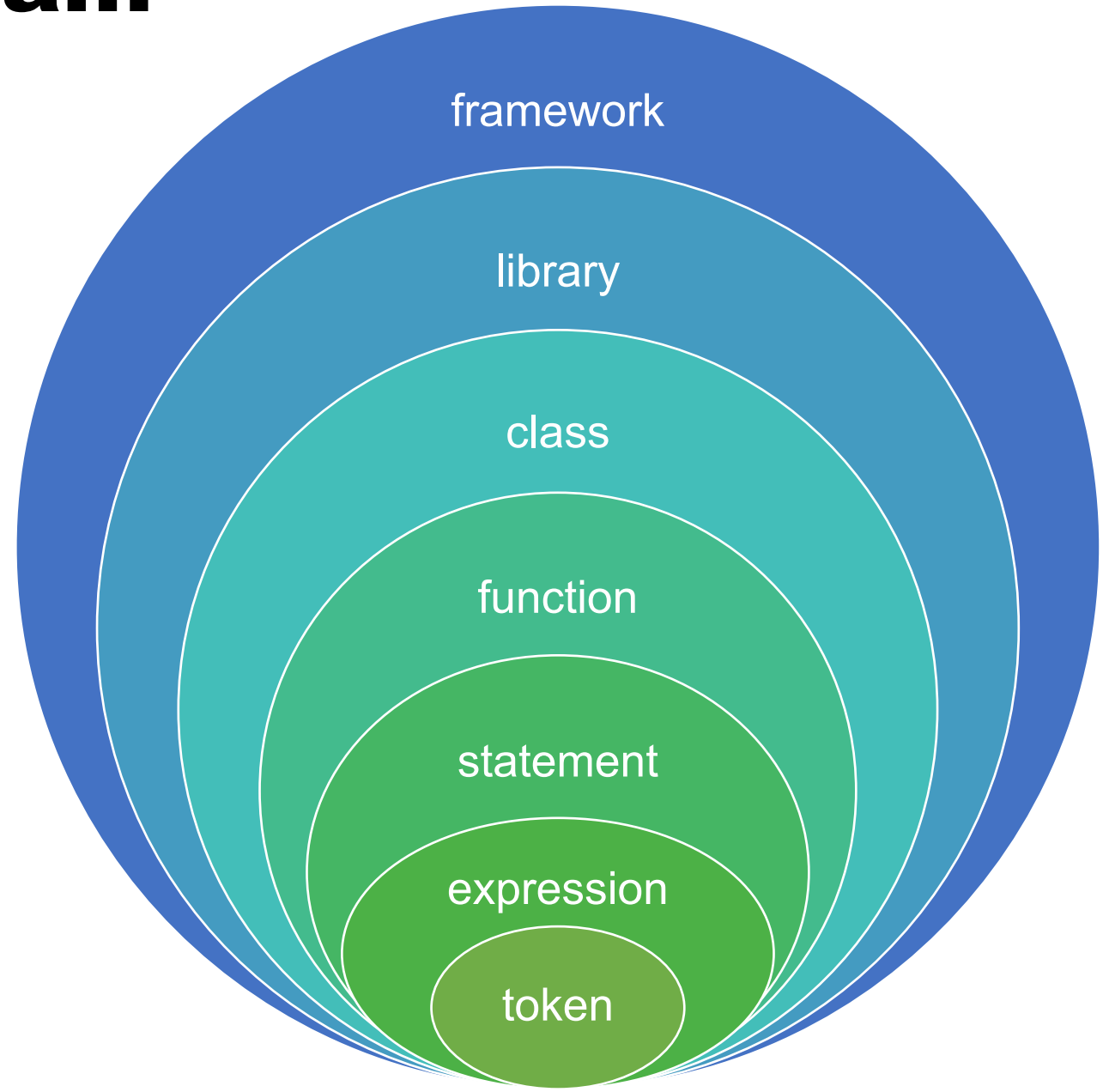
Structure of a program

- Token标记, word,
- a = i+++j;

  a = i + (++j) ;

  a = (i++) + j ;
- Longest token possible, left-to-right

- Expression表达式, phrase, value
- Statement语句, sentence, ;
- Reuse begin with function & class;



Taking a look at the sample program

- Code Reuse
  - Library: std
  - Class: ostream
    - extern std::ostream cout; //a global object / instance of class ostream
    - Member function: ostream::operator<<()
- Code Organization
  - Function: main()
  - Compound statement: {…}
  - Statement:…
  - Expression: …
  - Token: …

```
// A Hello World program
# include <iostream>
int main() {
    std::cout << "Hello, world!\n";
    return 0;
}
```

Syntax句法 and syntax errors

- In English, sentences are constructed according to specific grammatical rules – syntax
- C++ has a syntax too.
- the compiler is responsible for making sure your program follows the syntax of the C++ language. If you violate a rule, the compiler will issue you a **syntax error**.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Hello world!"
6      return 0;
7  }
```

- Visual studio produces the following error:
- c:\users\apomeranz\documents\visual studio 2013\projects\test1\test1\test1.cpp(6): error C2143: syntax error : missing ';' before 'return‘
- In this case, the error is actually at the end of line 5. Often, the compiler will pinpoint the exact line where the syntax error occurs for you. However, sometimes it doesn’t notice until the next line.
- Syntax errors are common when writing a program. Fortunately, they’re often **easily fixable**. The program can only be fully compiled (and executed) once all syntax errors are resolved.

Quiz

The following quiz is meant to reinforce your understanding of the material presented above.

1) What is the difference between a statement and an expression?
2) What is the difference between a function and a library?
3) What symbol do statements in C++ end with?
4) What is a syntax error?

Basic formatting

Whitespace and basic formatting

- **Whitespace** is a term that refers to characters that are used for formatting purposes.
  - spaces, tabs, (sometimes) newlines.
- The C++ compiler generally ignores whitespace, with a few minor exceptions.

```
1  cout << "Hello world!";
2
3  cout                 <<              "Hello world!";
4
5          cout <<          "Hello world!";
6
7  cout
8      << "Hello world!";
```

  - "Hello world!“ is different than "Hello world!“
  - Newlines are not allowed in quoted text:

```
1  cout << "Hello
2       world!" << endl; // Not allowed!
```

The following functions all do the same thing:

```
1  int add(int x, int y) { return x + y; }
2
3  int add(int x, int y) {
4      return x + y; }
5
6  int add(int x, int y)
7  {    return x + y; }
8
9  int add(int x, int y)
10 {
11     return x + y;
12 }
```

Basic formatting

- trust the programmer! => many formatting methods => disagreement
- rule of thumb
  - produce the most readable code
  - provide the most consistency.

- Braces are on their own lines
- Using tab

```
1  int add(int x, int y) { return x + y; }
2
3  int add(int x, int y) {
4      return x + y; }
5
6  int add(int x, int y)
7  {    return x + y; }
8
9  int add(int x, int y)
10 {
11     return x + y;
12 }
```

Lines should not be too long.

- 72, 78, or 80 characters is the maximum length a line should be.
- If a line is going to be longer, it should be broken (at a reasonable spot) into multiple lines

```
1  int main()
2  {
3      cout << "This is a really, really, really, really, really, really, really, " <<
           "really long line" << endl; // one extra indentation for continuation line
4
       cout << "This is another really, really, really, really, really, really, really, " <<
5              "really long line" << endl; // text aligned with the previous line for continuation line
6
       cout << "This one is short" << endl;
7  }
```

Lines should not be too long.

- If a long line that is broken into pieces is broken with an operator (eg. << or +), the operator should be placed at the end of the line, not the start of the next line:

```
1       cout << "This is a really, really, really, really, really, really, really, " <<
                "really long line" << endl;
2
```

Not

```
1       cout << "This is a really, really, really, really, really, really, really, "
2               << "really long line" << endl;
```

Use whitespace to make your code easier to read.

Harder to read:

```
nCost = 57;
nPricePerItem = 24;
nValue = 5;
nNumberOfItems = 17;
```

Easier to read:

```
nCost           = 57;
nPricePerItem   = 24;
nValue          = 5;
nNumberOfItems = 17;
```

Harder to read:

```
cout << "Hello world!" << endl; // cout and endl live in the iostream library
cout << "It is very nice to meet you!" << endl; // these comments make the code hard to read
cout << "Yeah!" << endl; // especially when lines are different lengths

```

Easier to read:

```
cout << "Hello world!" << endl;                    // cout and endl live in the iostream library
cout << "It is very nice to meet you!" << endl;   // these comments are easier to read
cout << "Yeah!" << endl;                          // especially when all lined up
```

Harder to read:

```
// cout and endl live in the iostream library
cout << "Hello world!" << endl;
// these comments make the code hard to read
cout << "It is very nice to meet you!" << endl;
// especially when all bunched together
cout << "Yeah!" << endl;
```

Easier to read:

```
// cout and endl live in the iostream library
cout << "Hello world!" << endl;

// these comments are easier to read
cout << "It is very nice to meet you!" << endl;

// when separated by whitespace
cout << "Yeah!" << endl;
```

Forward declarations and definitions

Forward declarations

- What would you expect this program to produce?

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
7      return 0;
8  }
9
10 int add(int x, int y)
11 {
12     return x + y;
13 }
```

add.cpp(6) : error C3861: 'add': identifier not found
add.cpp(10) : error C2365: 'add' : redefinition; previous definition was 'formerly unknown identifier'

Forward declarations

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
7      return 0;
8  }
9
10 int add(int x, int y)
11 {
12     return x + y;
13 }
```

- compiler reads files sequentially.
- That produces the first error (“identifier not found”).
- ***Rule: When addressing compile errors in your programs, always resolve the first error produced first.***

Option 1. Define add() before main()

```cpp
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
    return 0;
}
```

- Many functions => tedious to figure out caller & **callee**, and declare them sequentially

Option 1. Define add() before main()

- Many functions => tedious to figure out caller & **callee**, and declare them sequentially

- this option is not always possible.
  - A call B and B call A.

Option 2. Use a forward declaration

- A **forward declaration**
  - tell the compiler existence of an identifier *before* actually defining it.
  - even if it doesn't yet know how or where the function is defined.
  - when the compiler encounters an identifier, it'll understand we're making a function call, & can check to ensure we're calling the function correctly
- declaration statement: **function prototype原型**.
  - return type
  - Name
  - Parameters
  - but no function body

```
int add(int x, int y); // function prototype includes return type, name, paramet
ers, and semicolon.  No function body!
```

```cpp
#include <iostream>

int add(int x, int y); // forward declaration of add() (using a function prototype)

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is: " << add(3, 4) << endl; // this works because we forward declared add() above
    return 0;
}

int add(int x, int y) // even though the body of add() isn't defined until here
{
    return x + y;
}
```

- function prototypes do not need to specify the names of the parameters

```cpp
int add(int, int);
```

Forgetting the function body

- what happens if we declare a function but do not define it?
- it depends.
  - If the function is never called, the program will compile and run fine.
  - if the function is called
    - compile okay,
    - linker will complain: can’t resolve the function call.

```
Compiling...
add.cpp
Linking...
add.obj : error LNK2001: unresolved external symbol "int __cdecl add(int,int)" (?add@@YAHHH@Z)
add.exe : fatal error LNK1120: 1 unresolved externals
```

Declarations vs. definitions

- A **definition** actually implements or instantiates (causes memory to be allocated for) the identifier.
  - You can only have one definition per identifier.
  - A definition is needed to satisfy the **linker**.

```
1  int add(int x, int y) // defines function add()
2  {
3      return x + y;
4  }
5
6  int x; // instantiates (causes memory to be allocat
   ed for) an integer variable named x
```

- A **declaration** is a statement that defines an identifier (variable or function name) and its type.
  - A declaration is all that is needed to satisfy the compiler.

```
1  int add(int x, int y); // declares a function named
2  "add" that takes two int parameters and returns an
   int.  No body!
   int x; // declares an integer variable named x
```

Pure declarations

- In C++, all definitions also serve as declarations.
- Since “int x” is a definition, it’s by default a declaration too.
- This is the case with most declarations.

- Pure declarations: a small subset of declarations that are not definitions, such as function prototypes
  - forward declarations for variables
  - class declarations
  - type declarations

Quiz

- Write the function prototype for this function:

```
1  int doMath(int first, int second, int third, int fo
2  urth)
3  {
4      return first + second * third / fourth;
   }
```

Quiz

- state whether they fail to compile, fail to link, or compile and link.

```
1  #include <iostream>
2  int add(int x, int y);
3
4  int main()
5  {
6      using namespace std;
7      cout << "3 + 4 + 5 = " << add(3, 4, 5) << endl;
8      return 0;
9  }
10
11 int add(int x, int y)
12 {
13     return x + y;
14 }
```

Quiz

- state whether they fail to compile, fail to link, or compile and link.

```
1   #include <iostream>
2   int add(int x, int y);
3
4   int main()
5   {
6       using namespace std;
7       cout << "3 + 4 + 5 = " << add(3, 4, 5) << endl;
8       return 0;
9   }
10
11  int add(int x, int y, int z)
12  {
13      return x + y + z;
14  }
```

Quiz

- state whether they fail to compile, fail to link, or compile and link.

```
1   #include <iostream>
2   int add(int x, int y);
3
4   int main()
5   {
6       using namespace std;
7       cout << "3 + 4 + 5 = " << add(3, 4) << endl;
8       return 0;
9   }
10
11  int add(int x, int y, int z)
12  {
13      return x + y + z;
14  }
```

Quiz

- state whether they fail to compile, fail to link, or compile and link.

```
1   #include <iostream>
2   int add(int x, int y, int z);
3
4   int main()
5   {
6       using namespace std;
7       cout << "3 + 4 + 5 = " << add(3, 4, 5) << end
8   l;
9       return 0;
10  }
11
12  int add(int x, int y, int z)
13  {
14      return x + y + z;
    }
```

Programs with multiple files

Adding files to your project in Visual Studio

- Add new file
  - right click on “Source Files” in the Solution Explorer window on the left, and choose Add -> New Item. Make sure you have “C++ File (.cpp)” selected. Give the new file a name, and it will be added to your project

- Add existing file
  - right click on “Source Files” in the Solution Explorer, choose Add -> Existing Item, and then select your file.

- When you compile your program, the new file will be automatically included, since it’s part of your project.

A multi-file example

- add.cpp:

```cpp
//#include "stdafx.h" // uncomment if using Visual Studio

int add(int x, int y)
{
    return x + y;
}

```

- main.cpp:

```cpp
//#include "stdafx.h" // uncomment if using Visual Studio
#include <iostream>

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
    return 0;
}
```

```cpp
//#include "stdafx.h" // uncomment if using Visual Studio
#include <iostream>

int add(int x, int y); // needed so main.cpp knows that add() is a function declared elsewhere

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
    return 0;
}
```

- Does forward declaration work?
- Yes

Header files

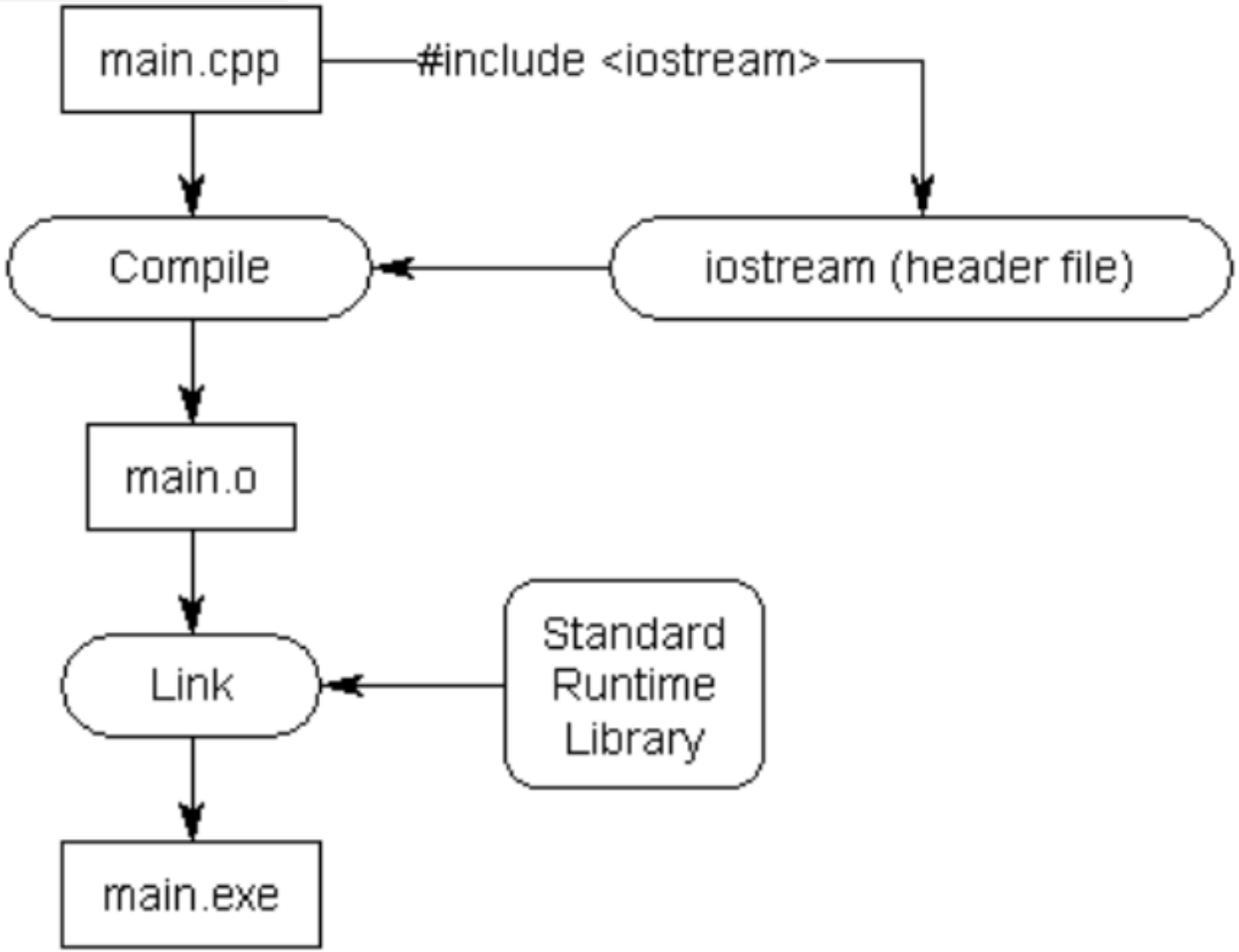
Headers, and their purpose

- As programs grow larger and larger (and include more files),
  - tedious to have to forward declare every function you want to use that lives in a different file.
  - Why not put all your declarations in one place?

- **header file/include file**
  - .h/.hpp extension or no extension at all.
  - Purpose: hold declarations for other files to use.

Using standard library header files

```
1  #include <iostream>
2  int main()
3  {
4      using namespace std;
5      cout << "Hello, world!"
6      return 0;
7  }
```



Writing your own header files

```
1  // This is start of the header guard.  ADD_H can be any unique name.  By conventio
2  n, we use the name of the header file.
3  #ifndef ADD_H
4  #define ADD_H
5
6  // This is the content of the .h file, which is where the declarations go
7  int add(int x, int y); // function prototype for add.h -- don't forget the semicol
8  on!
9
   // This is the end of the header guard
   #endif
```

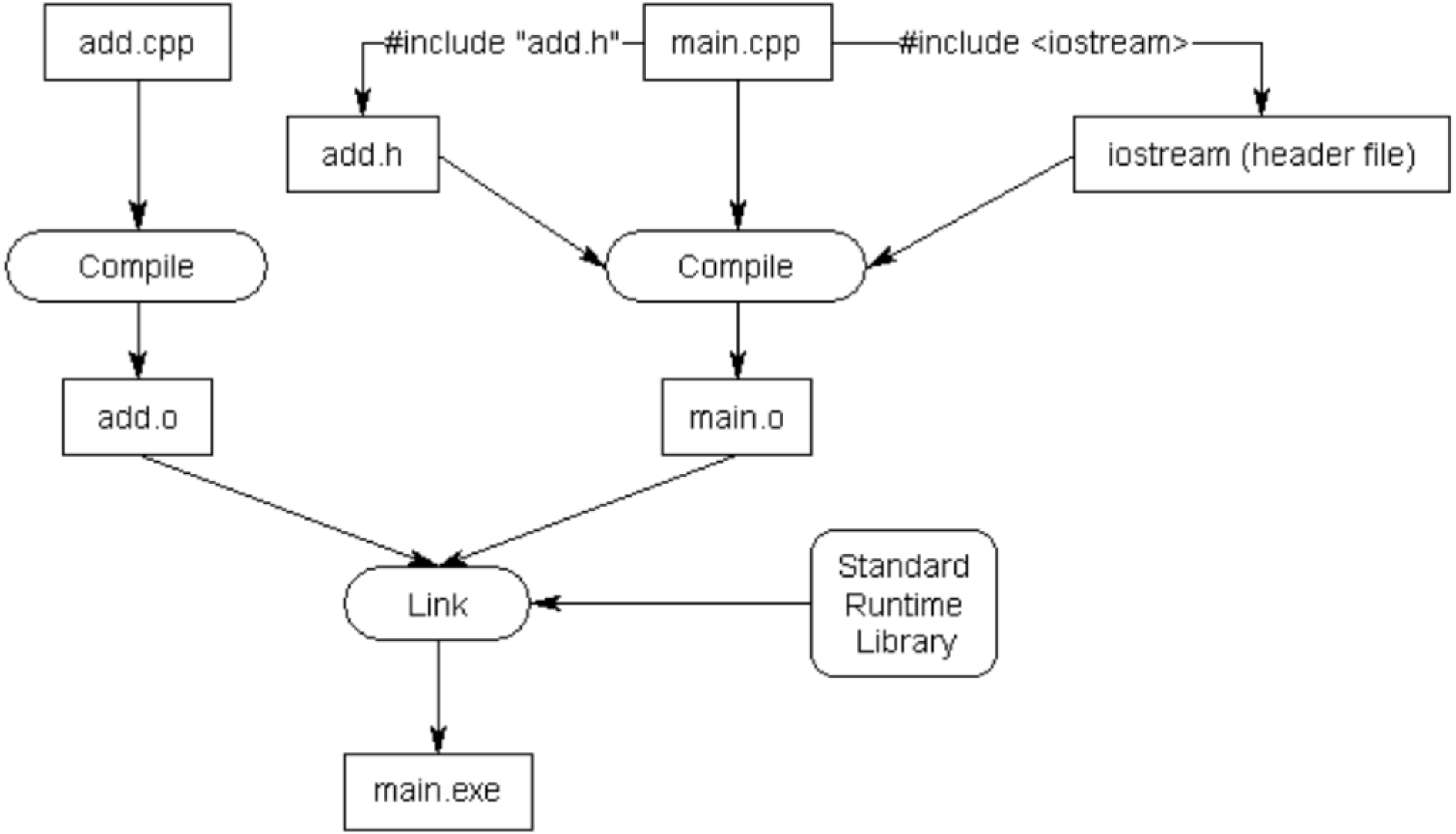
- **header guard**
  - prevent a given header file from being #included more than once from the same file.
  - Discuss it later

Use your own header files

```
#include <iostream>
#include "add.h"

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is " << add(3, 4) << endl;
    return 0;
}
```

Use your own header files

- *When you #include a file, the entire content of the included file is inserted at the point of inclusion.*



Angled brackets vs quotes

```
1  #include <iostream>
2  #include "add.h"
```

- angled brackets
  - tell the compiler that we are including a header file that was included with the compiler,
  - it should look for that header file in the system directories.
- double-quotes
  - tell the compiler that this is a header file we are supplying,
  - it should look for that header file in the current directory containing our source code first.
  - Then check any other include paths that you've specified as part of your compiler/IDE settings.
  - That failing, it will fall back to checking the system directories.

Including header files from other directories

- One (bad) way to do this is to include a relative path to the header file you want to include as part of the #include line.

```
1  #include "headers/myHeader.h"
2  #include "../moreHeaders/myOtherHeader.h"
```

- The downside
  - it requires you to reflect your directory structure in your code.
  - If you ever update your directory structure, your code won't work any more.

- setting an "include path" in your IDE project settings.
  - right click on your project in the Solution Explorer,
  - choose "Properties",
  - "VC++ Directories" tab.
  - "Include Directories".

Can I put function definitions in a header file?

- C++ won't complain if you do, but generally speaking, you shouldn't.

- for larger projects, this can make things take much longer to compile (as the same code gets recompiled each time it is encountered)
- could significantly bloat the size of your executable.

- If you make a change to a definition in a code file, only that .cpp file needs to be recompiled.
- If you make a change to a definition in a header file, every code file that includes the header needs to be recompiled.
- One small change can cause you to have to recompile your entire project!

Header file best practices

1. Always include header guards.

2. Do not define variables in header files unless they are constants. Header files should generally only be used for declarations.

3. Do not define functions in header files.

Header file best practices

4. Each header file should have a specific job, and be as independent as possible.
    1. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h.
    2. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.

5. Give your header files the same name as the source files they're associated with (e.g. grades.h goes with grades.cpp).

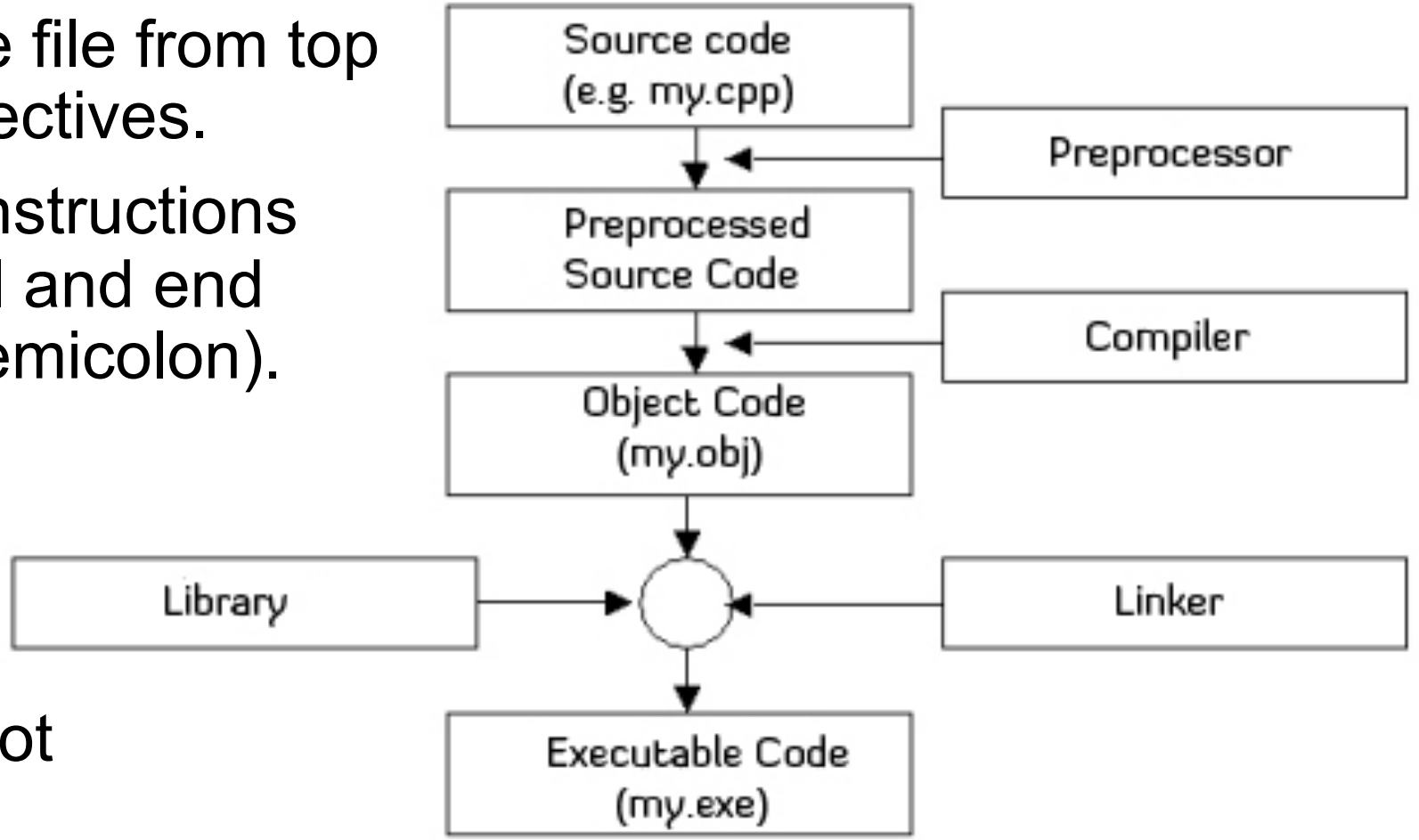
6. Try to minimize the number of other header files you #include in your header files. Only #include what is necessary.

7. Do not #include .cpp files.

A first look at the preprocessor

preprocessor

- scans through each code file from top to bottom, looking for directives.
- **Directives** are specific instructions that start with a # symbol and end with a newline (NOT a semicolon).



- it is not smart -- it does not understand C++ syntax;
- simply manipulates text

Includes

- copies the contents of the included file into the including file at the point of the #include directive.
- two forms:
- #include <filename>
  - look for the file in a special place defined by OS where header files for the C++ runtime library are held.
- #include "filename"
  - look for the file in directory containing the source file.
  - If it doesn’t find the header file there, it will check any other include paths that you’ve specified as part of your compiler/IDE settings.
  - That failing, it will act identically to the angled brackets case.

Macro defines

- #define
  - how an input sequence (e.g. an identifier) is converted into a replacement output sequence (e.g. some text).

- object-like macros vs. function-like macros.
- Function-like macros act like functions.
  - their use is generally considered dangerous,
  - should be replaced by an (inline) function.

- Object-like macros can be defined in one of two ways:
  - #define identifier
  - #define identifier substitution_text
- The top definition has no substitution text, whereas the bottom one does.

Object-like macros with substitution text

- all capital letters, using underscores to represent spaces

```
1  #define MY_FAVORITE_NUMBER 9
2
3  std::cout << "My favorite number is: " << MY_FAVORITE_NUMBER << std::endl;
```

```
1  std::cout << "My favorite number is: " << 9 << std::endl;
```

Object-like macros without substitution text

- Unlike object-like macros with substitution text, macros of this form are generally considered acceptable to use.

```
1  #define PRINT_JOE
2
3  #ifdef PRINT_JOE
4  cout << "Joe" << endl;
5  #endif
6
7  #ifdef PRINT_BOB
8  cout << "Bob" << endl;
9  #endif
```

Conditional compilation

- specify under what conditions something will or won't compile.
- #ifdef, #ifndef, and #endif.

```
1  #define PRINT_JOE
2
3  #ifdef PRINT_JOE
4  cout << "Joe" << endl;
5  #endif
6
7  #ifdef PRINT_BOB
8  cout << "Bob" << endl;
9  #endif
```

- Conditional compilation & header guards.

The scope of defines

function.cpp:

```
1  #include <iostream>
2
3  void doSomething()
4  {
5  #ifdef PRINT
6      std::cout << "Printing!"
7  #endif
8  #ifndef PRINT
9      std::cout << "Not printing!"
10 #endif
11 }
```

Not printing!

main.cpp:

```
1  void doSomething(); // forward declaration for function doSomething()
2  int main()
3  {
4  #define PRINT
5
6      doSomething();
7
8      return 0;
9  }
10
```

- Directives are resolved before compilation
- **from top to bottom** on a file-by-file basis, valid from the point of definition to the end of the file
- Once the preprocessor has finished, all directives from that file are **replaced** with ...
- This means?
- How to define it once and use it in multiple files?

Conditional compilation

function.cpp:

```
1  #include <iostream>
2
3  void doSomething()
4  {
5  #ifdef PRINT
6      std::cout << "Printing!"
7  #endif
8  #ifndef PRINT
9      std::cout << "Not printing!"
10 #endif
11 }
```

Not printing!

main.cpp:

```
1  void doSomething(); // forward declara
2  int main()
3  {
4  #define PRINT
5
6      doSomething();
7
8      return 0;
9  }
10
```

- Function.cpp => function.obj

  // sth from iostream

  void doSomething(){

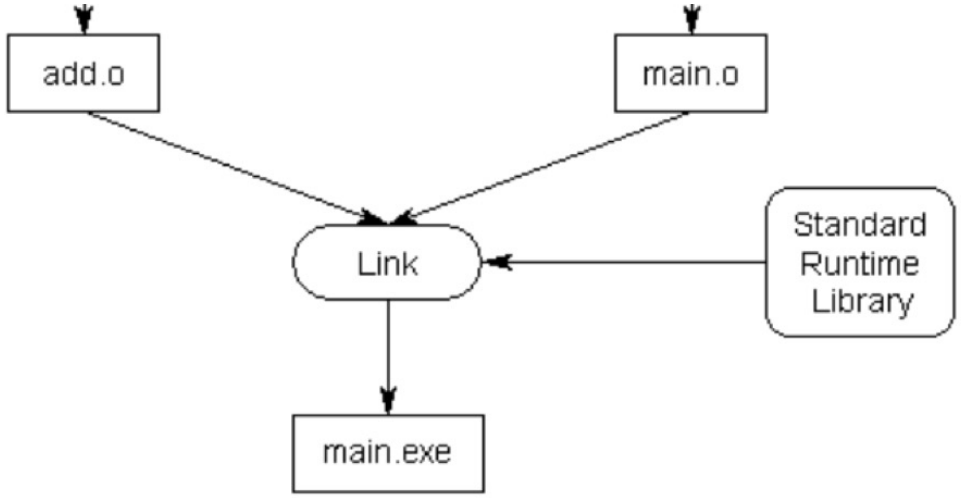
  std::cout << “Not printing!”;}

- Main.cpp => main.obj

  void doSomething();

  int main(){ doSomething();

  Return 0;}



Header guards

The duplicate definition problem

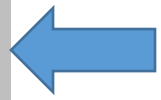
- an identifier can only have one definition

```
1  int main()
2  {
3      int x; // this is a definition for identifier x
4      int x; // compile error: duplicate definition
5
6      return 0;
7  }
```

The duplicate definition problem

- When a header file #includes another header file (which is common).
- How to resolve this issue?

```
int getSquareSides(){// from math.h
    return 4;
}

int getSquareSides(){// from geometry.h
    return 4;
}

int main(){
    return 0;
}
```



math.h:

```
1  int getSquareSides()
2  {
3      return 4;
4  }
```

geometry.h:

```
1  #include "math.h"
```

main.cpp:

```
1  #include "math.h"
2  #include "geometry.h"
3
4  int main()
5  {
6      return 0;
7  }
```

Header guards

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations and definitions here

#endif
```

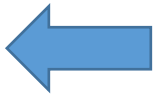
- All header files should have header guards
- SOME_UNIQUE_NAME_HERE: typically the name of the header file with a _H appended to it

math.h:

```
#ifndef MATH_H
#define MATH_H

int getSquareSides()
{
    return 4;
}

#endif
```

Updating our previous example with header guards

```
int getSquareSides(){// from math.h
    return 4;
}

// nothing from geometry.h

int main(){
    return 0;
}
```



math.h

```
1  #ifndef MATH_H
2  #define MATH_H
3
4  int getSquareSides()
5  {
6      return 4;
7  }
8
9  #endif
```

geometry.h:

```
1  #include "math.h"
```

main.cpp:

```
1  #include "math.h"
2  #include "geometry.h"
3
4  int main()
5  {
6      return 0;
7  }
```

Header guards do not prevent a header from being included once into different code files

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

int getSquarePerimeter(int sideLength);

#endif
```

square.cpp:

```
#include "square.h"  // square.h is inclu

int getSquarePerimeter(int sideLength)
{
    return sideLength * getSquareSides();
}
```

main.cpp:

```
#include "square.h" // square.h is also included once here

int main()
{
    std::cout << "a square has " << getSquareSides() << "sides" << std::endl;
    std::cout << "a square of length 5 has perimeter length " << getSquarePerimeter(5) << std::endl;

    return 0;
}
```

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

int getSquarePerimeter(int sideLength);

#endif
```

main.cpp:

```
#include "square.h" // square.h is also included once here

int main()
{
    std::cout << "a square has " << getSquareSides() << "sides" << std::endl;
    std::cout << "a square of length 5 has perimeter length " << getSquarePerimeter(5) << std::endl;

    return 0;
}
```

- Compile!
- but the linker will complain: multiple definitions for identifier getSquareSides!

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides(); // forward declarc
int getSquarePerimeter(int sideLength);

#endif
```

#pragma once

- Many compilers support a simpler, alternate form of header guards using the #pragma directive

- However, #pragma once is not an official part of the C++ language, and not all compilers support it (although most modern compilers do).

- For compatibility purposes, we recommend sticking to header guards.

Summary

- Header guards are designed to ensure that the contents of a given header file are not copied more than once into any single file, in order to prevent duplicate definitions.

- Note that header guards do *not* prevent the contents of a header file from being copied (once) into different code files.

- This is a good thing, because we often need to reference the contents of a given header from different code files.

Summary of all

- Variables
- Cout & cin
- Functions (parameters, arguments, return values)
- Naming identifiers
- Formatting
- Forward declarations
- Multiple files
- Preprocessor
- Header guards
- Scope: {}
- Keywords & Operators

Debugging your code

Six Stages of Debugging

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?