

C++ Program Design **-- C++ Basics**

Junjie Cao @ DLUT

Summer 2016

<http://jjcao.github.io/cPlusPlus>

Structure of a program

Statement

- **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

- An **expression** is analogous to **phrase**.
- 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](#)

- Primary expressions are the building blocks of more complex expressions. They are literals, names, and names qualified by the scope-resolution operator (::). Examples of primary expressions include:

100 // literal

::func // a global function

::operator + // a global operator function

(i + 1) // a parenthesized expression

MyClass // a identifier

cout // a identifier

- Postfix expressions consist of primary expressions or expressions in which postfix operators follow a primary expression.

func(i+1)

Token

- 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.
- Tokens are usually separated by *white space*.

Types of Tokens

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

Token

- 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`.

Operators

1. Mathematical: +, -, *, /, and parentheses.
2. % (the modulus operator) takes the remainder of two numbers:
 - 6%5 evaluates to 1.
3. Logical: used for “and,” “or,” and so on. More on those in the next lecture.
 - ==, !=, >, <=, &&, ||
4. Bitwise: used to manipulate the binary representations of numbers.
 - We will not focus on these

Operator precedence

1	() [] -> . ::	Grouping, scope, array/member access
2	! ~ * & sizeof (type cast) ++ -	(most) unary operations, sizeof and typecasts
3	* / %	Multiplication, division, modulo
4	+ -	Addition and subtraction
5	<< >>	Bitwise left and right shift
6	< <= > >=	Comparisons: less than, etc.
7	== !=	Comparisons: equal and not equal
8	&	Bitwise AND
9	^	Bitwise exclusive OR
10		Bitwise inclusive (normal) OR
11	&&	Logical AND
12		Logical OR
13	?:	Conditional expression (ternary operator)
14	= += -= *= /= %=, etc.	Assignment operators
15	,	Comma operator

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

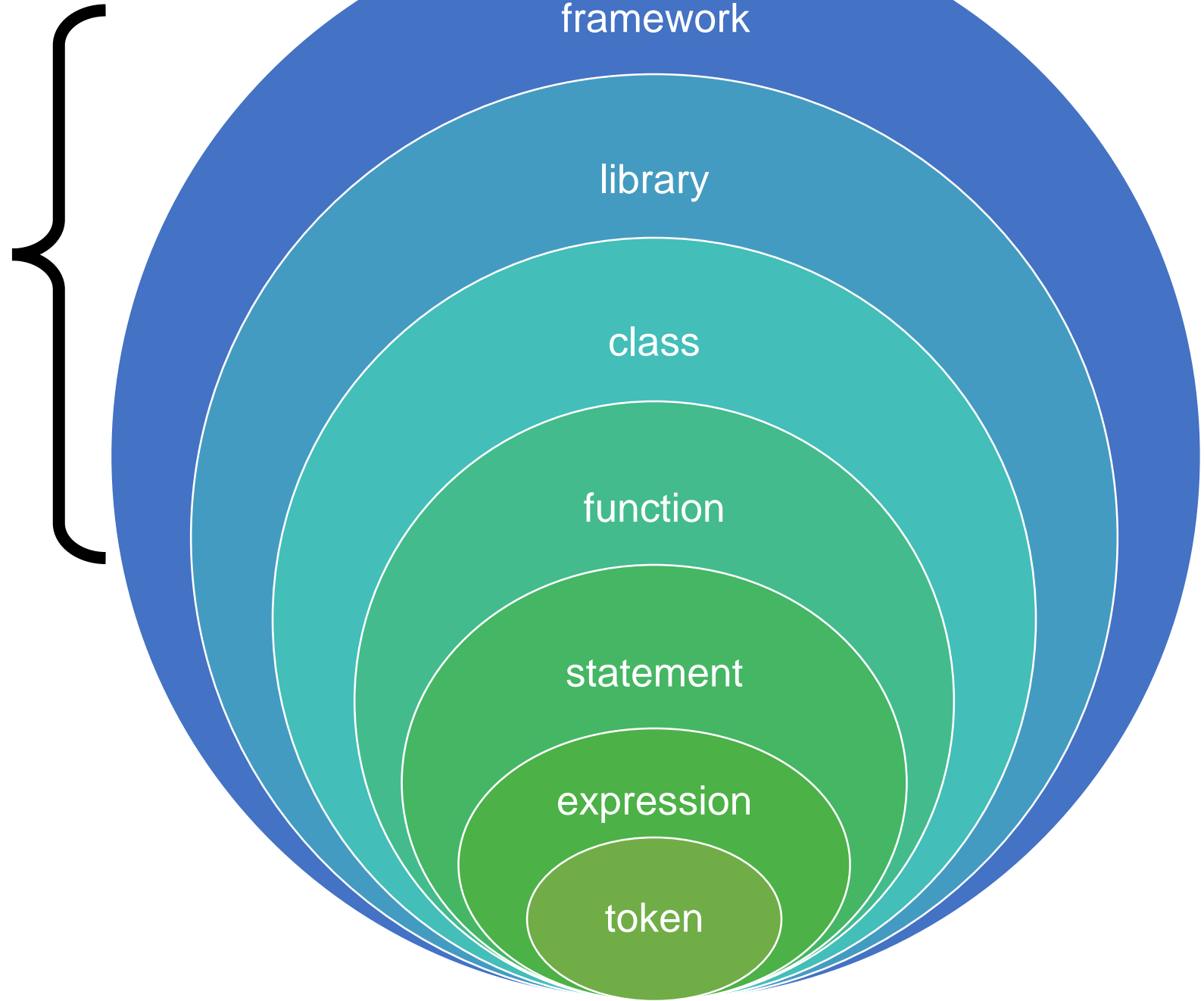
- combine data and the functions
- Contains many functions
- One class per file: A.h + A.cpp

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

Reuse



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 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?

Overview of C

Variables

Variables

- An expression:
 - $4 + 2$
- We might want to give a value a name so we can **refer to it later**. We do this using variables. A variable is a **named location in memory**
 - `x = 5; //a statement`
- All computers have memory, called RAM (random access memory), that is available for programs to use. When a variable is defined, a piece of that memory is set aside for the variable.

Declaration

- An integer variable is a variable that holds an integer value.
- In order to define a variable, we generally use a **declaration statement**.
 - `int x;`
- We must tell the compiler what **type** x will be so that it knows how much **memory to reserve** for it and what kinds of **operations may be performed** on it.
- When this statement is executed by the CPU, a piece of memory from RAM will be set aside (called **instantiation**).
- For the sake of example, let's say that the variable x is assigned memory location 140. Whenever the program sees the variable x in an expression or statement, it knows that it should look in memory location 140 to get the value.

Initialization

- **initialization** of `x`, where we specify an initial value for it. This introduces **assignment operator**: `=`
 - `x=5;`
- When the CPU executes this statement, it translates this to “put the value of 5 in memory location 140”.
- Later in our program, we could print that value to the screen using `std::cout`:
 - `std::cout << x; // prints the value of x (memory location 140) to the console`

I-values and r-values

- variables are a type of I-value (pronounced ell-value).
- An **I-value** is a value that has an **address** (in **memory**).
- The name I-value came about because I-values are the only values that **can be on the left side of an assignment statement**.
 - When we do an assignment, the left hand side of the assignment operator must be an I-value.
 - Consequently, a statement like `5 = 6;` will cause a compile error, because 5 is not an I-value.
 - The value of 5 has no memory, and thus nothing can be assigned to it. 5 means 5, and its value can not be reassigned.
- When an I-value has a value assigned to it, the current value at that memory address is overwritten.

l-values and r-values

- The opposite of l-values are r-values (pronounced arr-values).
- An **r-value** refers to any value that can be assigned to an l-value.
- r-values are always evaluated to produce a single value.
- Examples of r-values are
 - single numbers (such as 5, which evaluates to 5),
 - variables (such as x, which evaluates to whatever value was last assigned to it),
 - expressions (such as $2 + x$, which evaluates to the value of x plus 2).

I-values and r-values

```
1  int y;           // define y as an integer variable
2  y = 4;           // 4 evaluates to 4, which is then assigned to y
3  y = 2 + 5;       // 2 + 5 evaluates to 7, which is then assigned to y
4
5  int x;           // define x as an integer variable
6  x = y;           // y evaluates to 7 (from before), which is then assigned to x.
7  x = x;           // x evaluates to 7, which is then assigned to x (useless!)
8  x = x + 1;       // x + 1 evaluates to 8, which is then assigned to x.
```

- In last statement, x is being used in two different contexts.
 - On the left side of the assignment operator, “x” is being used as an I-value (variable with an address).
 - On the right side of the assignment operator, x is being used as an r-value, and will be evaluated to produce a value (in this case, 7).
 - When C++ evaluates the above statement, it evaluates as:
 - $x = 7 + 1;$
- Notes
 - I-values must have address, r-values will be evaluated to produce a value
 - Expressions that refer to memory locations are called "I-value" expressions.

l-values and r-values

- Practical application (best practice)
 - `if (0==x)` //instead of `x==0` When placing the r-value on the left, mistyping the `==` operator as the `=` operator triggers a compilation error:
 - `if (0=x)` // error: "L-value required" Although this isn't the most intelligible error message, it certainly catches the bug.

Initialization vs assignment

- two related concepts: assignment and initialization (new programmers often get mixed up)
- After a variable is defined, a value may be **assigned** to it via the assignment operator (the = sign):

```
1 | int x; // this is a variable definition  
2 | x = 5; // assign the value 5 to variable x
```

- C++ will let you both define a variable AND give it an initial value in the same step: **initialization**.

```
1 | int x = 5; // initialize variable x with the value 5
```

- A variable can only be initialized when it is defined.
- we'll see cases in future lessons where some types of variables require an initialization value, or disallow assignment. For these reasons, it's useful to **make the distinction now**.

Uninitialized variables

- Unlike some programming languages, C/C++ does not initialize variables to a given value (such as zero) automatically (for performance reasons).
 - Thus when a variable is assigned to a memory location by the compiler, the default value of that variable is whatever **garbage** happens to already be in that memory location!
- A variable that has not been assigned a value is called an **uninitialized variable**.
 - Note: Some compilers, such as Visual Studio, *will* initialize the contents of memory when you're using a debug build configuration. This will not happen when using a release build configuration.

Uninitialized variables

- Uninitialized variables can lead to interesting (and by interesting, we mean unexpected) results. Consider the following short program:

```
1 // #include "stdafx.h" // Uncomment if Visual Studio user
2 #include <iostream>
3
4 int main()
5 {
6     // define an integer variable named x
7     int x;
8
9     // print the value of x to the screen (dangerous, because x is uninitialized)
10    std::cout << x;
11
12    return 0;
13 }
```

- In this case, the computer will assign some unused memory to x
- What value will it print? The answer is “who knows!”, and the answer may change every time you run the program.

Uninitialized variables

- Using uninitialized variables is one of the most common mistakes that novice programmers make,
 - Unfortunately, it can also be one of the most challenging to debug (because the program may run fine anyway if the uninitialized value happened to get assigned to a spot of memory that had a reasonable value in it, like 0).
 - Fortunately, most modern compilers will print warnings at compile-time.
 - `vc2005projectstesttesttest.cpp(11) : warning C4700: uninitialized local variable 'x' used`
- ***Rule: Initialize your variables.***
 - A good rule of thumb is to initialize your variables.
 - This ensures that your variable will always have a consistent value, making it easier to debug if something goes wrong somewhere else.

Quiz

```
1  int x = 5;
2  x = x - 2;
3  std::cout << x << std::endl; // #1
4
5  int y = x;
6  std::cout << y << std::endl; // #2
7
8  // x + y is an r-value in this context, so evaluate their values
9  std::cout << x + y << std::endl; // #3
10
11 std::cout << x << std::endl; // #4
12
13 int z;
14 std::cout << z << std::endl; // #5
```

cout & cin

std::cout

- the `std::cout` object (in the `iostream` library) can be used to output text to the console.
- To print more than one thing on the same line, the output operator (`<<`) can be used multiple times. For example:

```
1  #include <iostream>
2
3  int main()
4  {
5      int x = 4;
6      std::cout << "x is equal to: " << x;
7      return 0;
8  }
```

This program prints:

```
x is equal to: 4
```

std::cout

- What would you expect this program to print?

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Hi!";
6      std::cout << "My name is Alex.";
7      return 0;
8  }
```

- Hi!My name is Alex.

```
std::cout << "Hi!" << std::endl;
std::cout << "My name is Alex." << std::endl;
```

std::cin

- std::cin reads input from the user at the console using the input operator (>>).
- we can store it in a variable.

```
1 // #include "stdafx.h" // Uncomment this line if using Visual Studio
2 #include <iostream>
3
4 int main()
5 {
6     std::cout << "Enter a number: "; // ask user for a number
7     int x = 0;
8     std::cin >> x; // read number from console and store it in x
9     std::cout << "You entered " << x << std::endl;
10    return 0;
11 }
```

Enter a number: 4

You entered 4

If your screen closes immediately after entering a number, add a statement: `std::cin.get();` or `std::cin >>x;`

After pressing “Enter” key, the console will be closed.

The std namespace

- Everything in the standard library is defined inside a special area (called a **namespace**) that is named *std* (short for standard).
 - It turns out that `std::cout`'s name isn't really "`std::cout`". It's actually just "`cout`", and **"std" is the name of the namespace it lives inside**.
 - We'll talk more about namespaces in future and also teach you how to create your own.
- whenever we use something (like `cout`) that is part of the standard library, we need to **tell the compiler that it is inside** the `std` namespace.
 - **Explicit namespace qualifier** `std::`, `std::cout << "Hello world!";`
 - One way to simplify things is to utilize a **using declaration** statement.

```
using std::cout; // this using declaration tells the compiler that cout should resolve to std::cout
cout << "Hello world!"; // so no std:: prefix is needed here!
```


namespace

- whenever we use something (like cout) that is part of the standard library, we need to **tell the compiler that it is inside** the std namespace.

- **Explicit namespace qualifier** std::, std::cout << "Hello world!";
- One way to simplify things is to utilize a **using declaration** statement.

```
using std::cout; // this using declaration tells the compiler that cout should resolve to std::cout
cout << "Hello world!"; // so no std:: prefix is needed here!
```

- The **using directive**: tells the compiler that we want to use *everything* in the std namespace, so if the compiler finds a name it doesn't recognize, it will check the std namespace.

```
using namespace std; // this using directive tells the compiler that we're using everything in the std namespace!
cout << "Hello world!"; // so no std:: prefix is needed here!
```

- a naming conflict => compile error

Using declarations and directives inside or outside of a function

- If a using declaration or directive is used within a function, the names in the namespace are only directly accessible within that function.
 - That limits the chance for naming collisions to occur just within that function.
- if a using declaration or directive is used outside of a function, the names in the namespace are directly accessible anywhere in the entire file!
 - That can greatly increase the chance for collisions.

```
1  #include <iostream>
2
3  int cout() // declares our own "cout" function
4  {
5      return 5;
6  }
7
8  int main()
9  {
10     using namespace std; // makes std::cout accessible as "cout"
11     cout << "Hello, world!"; // uh oh! Which cout do we want here?
12
13     return 0;
14 }
```

A note on using statements

- *Rule: Avoid "using" statement outside of a function body.*
 - In either case, you should avoid "using" statements outside of a function body.
 - It is acceptable so long as they are used *only* within individual functions (which limits the places where naming collisions can occur to just those functions).

functions and return values

Function

- A **function** is a **reusable** sequence of statements designed to do a particular job.
 - You already know that every program must have a function named `main()` (which is the where the program starts execution).
 - However, most programs use many functions.
- A **function call** is an expression that tells the CPU to interrupt the current function and execute another function.
 - A program will be executing statements sequentially inside one function when it encounters a function call.
 - The CPU “puts a bookmark” at the current point of execution, and then **calls** (executes) the function named in the function call.
 - When the called function terminates, the CPU goes back to the point it bookmarked, and resumes execution.

Function call

- The function initiating the function call is called the **caller**, and the function being called is the **callee** or **called** function.

```
1  // #include <stdafx.h> // Visual Studio users need to uncomment this line
2  #include <iostream> // for std::cout and std::endl
3
4  // Definition of function doPrint()
5  void doPrint() // doPrint() is the called function in this example
6  {
7      std::cout << "In doPrint()" << std::endl;
8  }
9
10 // Definition of function main()
11 int main()
12 {
13     std::cout << "Starting main()" << std::endl;
14     doPrint(); // Interrupt main() by making a function call to doPrint().
15     // main() is the caller.
16     std::cout << "Ending main()" << std::endl;
17     return 0;
18 }
```

This program produces the following output:

Starting main()
In doPrint()
Ending main()

Return values

- A return type of **void** means the function does not return a value. A return type of **int** means the function returns an integer value to the caller.

```
1 // void means the function does not return a value to the caller
2 void returnNothing()
3 {
4     // This function does not return a value so no return statement is
   needed
5 }
6
7 // int means the function returns an integer value to the caller
8 int return5()
9 {
10     return 5; // this function returns an integer, so a return statemen
   t is needed
11 }
```

```

15 int main()
16 {
17     std::cout << return5() << std::endl; // prints 5
18     std::cout << return5() + 2 << std::endl; // prints 7
19
20     returnNothing(); // okay: function returnNothing() is called, no va
lue is returned
21     return5(); // okay: function return5() is called, return value is d
iscarded
22
23     std::cout << returnNothing(); // This line will not compile. You'l
l need to comment it out to continue.
24
25     return 0;
26 }

```

- “Can my function return multiple values using a return statement?”.
- No.
- Functions can only return a single value using a return statement.
- However, there are ways to work around the issue,

Returning to main

- When the program is executed, the operating system makes a function call to `main()`.
- Execution then jumps to the top of `main`.
- The statements in `main` are executed sequentially.
- Finally, `main` returns an integer value (usually 0) back to the operating system.

Returning to main

- Why return a value back to the operating system?
 - This value is called a **status code**, and it tells the operating system (and any other programs that called yours) whether your program executed successfully or not.
 - By consensus, a return value of 0 means success, and a positive return value means failure.

Reusing functions

- The same function can be called multiple times, which is useful if you need to do
- `main()` isn't the only function that can call other functions.
- Any function can call another function!

```
1  // #include <stdafx.h> // Visual Studio users need to uncomment this line
2  #include <iostream>
3
4  void printA()
5  {
6      std::cout << "A" << std::endl;
7  }
8
9  void printB()
10 {
11     std::cout << "B" << std::endl;
12 }
13
14 // function printAB() calls both printA() and printB()
15 void printAB()
16 {
17     printA();
18     printB();
19 }
20
21 // Definition of main()
22 int main()
23 {
24     std::cout << "Starting main()" << std::endl;
25     printAB();
26     std::cout << "Ending main()" << std::endl;
27     return 0;
28 }
```

Nested functions

- Functions can not be defined inside other functions (called nesting) in C++. The following program is not legal:

```
1  #include <iostream>
2
3  int main()
4  {
5      int foo() // this function is nested inside main(), which is illegal
6      {
7          std::cout << "foo!";
8          return 0;
9      }
10
11     foo();
12     return 0;
13 }
```

```
1  #include <iostream>
2
3  int foo() // no longer inside of main()
4  {
5      std::cout << "foo!";
6      return 0;
7  }
8
9  int main()
10 {
11     foo();
12     return 0;
13 }
```

Functions, parameters & arguments

Function parameters and arguments

- a function can return a value back to the caller via the function's return value.
- it is useful to be able to pass information *to* a function being called, so that the function has data to work with.
- For example, if we wanted to write a function to add two numbers, we need a way to tell the function which two numbers to add when we call it. Otherwise, how would the function know what to add?
 - We do that via define many functions => stupid
 - Or via function parameters and arguments => smart

Function parameters

- A parameter is the variable which is part of the function's signature (function declaration/definition).

```
1 // This function takes no parameters
2 // It does not rely on the caller for anything
3 void doPrint()
4 {
5     std::cout << "In doPrint()" << std::endl;
6 }
7
8 // This function takes one integer parameter named x
9 // The caller will supply the value of x
10 void printValue(int x)
11 {
12     std::cout << x << std::endl;
13 }
14
15 // This function has two integer parameters, one named x, and one named y
16 // The caller will supply the value of both x and y
17 int add(int x, int y)
18 {
19     return x + y;
20 }
```

Function argument

- An **argument** is a value that is passed *from* the caller *to* the function when a function call is made:

```
1 | printValue(6); // 6 is the argument passed to function printValue()  
2 | add(2, 3); // 2 and 3 are the arguments passed to function add()
```


How parameters, arguments & return values work together

- When a function is called, all of the **parameters** of the function are created as **variables**
- the value of each of the **arguments** is *copied* into the matching parameter. This process is called **pass by value**.
- By using both parameters and a return value, we can create functions that take data as input, do some calculation with it, and return the value to the caller.

```
1  // #include "stdafx.h" // Visual Studio users need to uncomment this line
2  #include <iostream>
3
4
5  int add(int x, int y)
6  {
7      return x + y;
8  }
9
10
11 int main()
12 {
13     std::cout << add(4, 5) << std::endl;
14     return 0;
15 }
```

The diagram illustrates the execution of the `add` function call within `main`. It shows the argument `4` being passed to the parameter `x` and the argument `5` being passed to the parameter `y`. The return value of the function, `9`, is then passed back to the caller.

How parameters and return values work together

```
2  #include <iostream>
3
4  int add(int x, int y)
5  {
6      return x + y;
7  }
8
9  int multiply(int z, int w)
10 {
11     return z * w;
12 }
13
14 int main()
15 {
16     using namespace std;
17     cout << add(4, 5) << endl; // within add(), x=4, y=5, so x+y=9
18     cout << multiply(2, 3) << endl; // within multiply(), z=2, w=3, so
    z*w=6
19
20     // We can pass the value of expressions
21     cout << add(1 + 2, 3 * 4) << endl; // within add(), x=3, y=12, so
    x+y=15
22
23     // We can pass the value of variables
24     int a = 5;
25     cout << add(a, a) << endl; // evaluates (5 + 5)
26
27     cout << add(1, multiply(2, 3)) << endl; // evaluates 1 + (2 * 3)
28     cout << add(1, add(2, 3)) << endl; // evaluates 1 + (2 + 3)
29
30     return 0;
31 }
```

Conclusion

- **Parameters** are the key mechanism by which functions can be written in a **reusable** way,
 - as it allows them to perform tasks without knowing the specific input values ahead of time.
 - Those input values are passed in as **arguments** by the caller.
- Return values allow a function to return a value back to the caller.

Quiz

1) What's wrong with this program fragment?

```
1 void multiply(int x, int y)
2 {
3     return x * y;
4 }
5
6 int main()
7 {
8     std::cout << multiply(4, 5) << std::endl;
9     return 0;
10 }
```

Quiz

2) What two things are wrong with this program fragment?

```
1  int multiply(int x, int y)
2  {
3      int product = x * y;
4  }
5
6  int main()
7  {
8      std::cout << multiply(4) << std::endl;
9      return 0;
10 }
```

Quiz

3) What value does the following program print?

```
1  #include <iostream>
2
3  int add(int x, int y, int z)
4  {
5      return x + y + z;
6  }
7
8  int multiply(int x, int y)
9  {
10     return x * y;
11 }
12
13 int main()
14 {
15     std::cout << multiply(add(1, 2, 3), 4) << std::endl;
16     return 0;
17 }
```

Why functions are useful, and how to use them effectively

Why functions are useful, and how to use them effectively

- Organization: all code inside main() => complicated => divide & conquer
- Abstraction
 - In order to use a function, you only need to know its name, inputs, outputs. You don't need to know how it works. This is super-useful for making other people's code accessible (such as everything in the standard library).
- Testing
 - functions are self-contained, once be tested to ensure it works, don't need to test it again unless it is changed
- Reusability:
 - called multiple times from within the program and other program.
 - This avoids duplicated code
 - minimizes the probability of copy/paste errors.
- Extensibility
 - When we need to extend our program to handle a case it didn't handle before, functions allow us to make the change in one place and have that change take effect every time the function is called.

When to write functions

- Codes that appear more than once.
- A function should generally perform one (and only one) task.
- Codes for accomplish a task are too long
- When a function becomes too long, too complicated, or hard to understand, it should be split into multiple sub-functions.
- This is called **refactoring**.

function – no more than 1 page

- Typically, when learning C++, you will write a lot of programs that involve 3 subtasks:
 1. Reading inputs from the user
 2. Calculating a value from the inputs
 3. Printing the calculated value
- For trivial programs (e.g. less than 20 lines of code), some or all of these can be done in main().
- However, for longer programs (or just for practice) each of these is a good candidate for an individual function

function – no more than 1 page

- New programmers often **combine calculating a value and printing the calculated value** into a single function.
- **this violates the “one task” rule of thumb for functions.**
- A function that calculates a value should return the value to the caller and let the caller decide what to do with the calculated value (such as call another function to print the value).

A first look at local scope

When is an instantiated variable destroyed?

- A variable's **scope** determines who can see and use the variable
- function parameters & variables declared inside function body have **local scope**.
- That is, those variables can only be seen and used within the function that declares them.
- Local variables are created at the point of definition, and **destroyed** when they go out of scope.

When is an instantiated variable destroyed?

```
1  #include <iostream>
2
3  int add(int x, int y) // x and y are created here
4  {
5      // x and y are visible/usable within this function only
6      return x + y;
7  } // x and y go out of scope and are destroyed here
8
9  int main()
10 {
11     int a = 5; // a is created and initialized here
12     int b = 6; // b is created and initialized here
13     // a and b are usable within this function only
14     std::cout << add(a, b) << std::endl; // calls function add() with
15     // x=a and y=b
16     return 0;
17 } // a and b go out of scope and are destroyed here
```

- Note that if function add() were to be called twice, parameters x and y would be created and destroyed twice -- once for each call. In a program with lots of functions, variables are created and destroyed often.

Local scope prevents naming collisions

- each function doesn't need to care what other functions, including caller, name their variables.

```
1  #include <iostream>
2
3  int add(int x, int y) // add's x is created here
4  {
5      return x + y;
6  } // add's x goes out of scope and is destroyed here
7
8  int main()
9  {
10     int x = 5; // main's x is created here
11     int y = 6;
12     std::cout << add(x, y) << std::endl; // the value from main's x is
        copied into add's x
13     return 0;
14 } // main's x goes out of scope and is destroyed here
```

What does the following program print?

```
1  #include <iostream>
2
3  void doIt(int x)
4  {
5      x = 3;
6      int y = 4;
7      std::cout << "doIt: x = " << x << " y = " << y << std::endl;
8  }
9
10 int main()
11 {
12     int x = 1;
13     int y = 2;
14     std::cout << "main: x = " << x << " y = " << y << std::endl;
15     doIt(x);
16     std::cout << "main: x = " << x << " y = " << y << std::endl;
17     return 0;
18 }
```


Keywords and naming identifiers

Keywords

- C++ reserves a set of 73 words for its own use. These words are called **keywords**
- each of these keywords has a special meaning within the C++ language.

alignas **	class	else	inline	public	switch	unsigned
alignof **	const	enum	int	register	template	using *
asm	constexpr **	explicit	long	reinterpret_cast	this	virtual
auto	const_cast *	*	mutable *	*	thread_local	void
bool *	continue	export	namespace	return	**	volatile
break	decltype	*	*	short	throw	wchar_t
case	default	extern	new	signed	true *	*
catch	delete	false *	noexcept	sizeof	try	while
char	do	float	**	static	typedef	
char16_t	double	for	nullptr **	static_assert **	typeid *	
**	dynamic_cast	friend	operator	static_cast *	typename *	
char32_t	*	goto	private	struct	union	
**		if	protected			

* These 15 keywords were added in C++98. Some older reference books or material may omit these.

** These 9 keywords were added in C++11. If your compiler is not C++11 compliant, these keywords may not be functional.

Identifiers, and naming them

- The name of a variable, function, class, or other object in C++ is called an **identifier**.
 - identifier can not be a keyword.
 - can only be composed of **letters** (lower or upper case), **numbers**, and the **underscore** character.
 - That means the name can not contain symbols (except the underscore) nor whitespace.
 - The identifier must begin with a letter (lower or upper case) or an underscore. It can not start with a number.
 - C++ distinguishes between lower and upper case letters.
 - nvalue is different than nValue is different than NVALUE.

Convention of C++

- function/variable names should begin with a lowercase letter

```
1  int value; // correct
2
3  int Value; // incorrect (should start with lower case letter)
4  int VALUE; // incorrect (should start with lower case letter)
5  int VaLuE; // incorrect (see your psychiatrist) ;)
```

- Identifier names that start with a capital letter are typically used for structs, classes, and enumerations
- For multi-word: there are two conventions:
 - separated by underscores,
 - intercased (CamelCase)

```
1  int my_variable_name; // correct (separated by underscores)
2  int myVariableName; // correct (intercased/CamelCase)
3
4  int my variable name; // incorrect (spaces not allowed)
5  int MyVariableName; // incorrect (should start with lower case letter)
```

give your identifiers names that actually describe what they are

- Not too short
- The more complex the code the variable is being used in, the better name it should have.

int ccount	Bad	Nobody knows what a ccount is
int customerCount	Good	Clear what we're counting
int i	Bad*	Generally bad unless use is trivial, such as loop variables
int index	Either	Okay if obvious what we're indexing
int totalScore	Good	Descriptive
int _count	Bad	Do not start variable names with underscore
int count	Either	Okay if obvious what we're counting
int data	Bad	What kind of data?
int value1, value2	Either	Can be hard to differentiate between the two
int numberOfApples	Good	Descriptive
int monstersKilled	Good	Descriptive
int x, y	Bad*	Generally bad unless use is trivial, such as in trivial mathematical functions

Comment

- *numberOfChars* that is supposed to store the number of characters in a piece of text.
- Does the text “Hello World!” have 10, 11, or 12 characters?
- It depends on whether we’re including whitespace or punctuation.
- Rather than naming the ***variable**numberOfCharsIncludingWhitespaceAndPunctuation***,
- a comment on the declaration line should help the user figure it out:

// holds number of chars in a piece of text -- including whitespace and punctuation!

int numberOfChars;

A first look at operators

Operators & expression

- What is an expression?
- “A mathematical entity that evaluates to a value”.
- **literals, variables, functions, & operators => a value.**
- What is a literal?
 - Fixed value, hardcoded in the source code

```
1  #include <iostream>

   int main()
   {
       int x = 2; // x is a variable, 2 is a literal
       std::cout << 3 + 4; // 3 + 4 is an expression, 3 and 4 are literals
       std::cout << "Hello, world!"; // "Hello, world" is a literal too
2  }
```


Operands

- **Literals, variables, & functions** are all known as operands.
- **Operands** supply the data that the expression works with.
 - We just introduced literals, which evaluate to themselves.
 - Variables evaluate to the values they hold.
 - Functions evaluate to produce a value of the function's return type (unless the return type is void).

Operators

- **Operators** tell the expression how to combine one or more operands to produce a new result
- “3 + 4”
 - plus operator
 - tells how to combine the operands 3 and 4 to produce a new value (7).
- arithmetic operators: +, -, *, /, = (Assignment)
- Relational & Logical operators: >, >=, <, <=, !=, == (equality)
- ...
- The most common mistake: = vs ==

Operators come in three types:

- **Unary.**

- In the expression -5 , $-$ operator is only being applied to one operand (5) to produce a new value (-5).

- **Binary**

- In the expression $3 + 4$, the $+$ operator is working with a left operand (3) and a right operand (4) to produce a new value (7).

- **Ternary**

- There is only one of these in C++, which we'll cover later.

- **Note:** some operators have more than one meaning.

- $-$ operator
 - Unary: invert a number's sign
 - Binary: do arithmetic subtraction

This is just the tip of the iceberg in terms of operators. We will take an in-depth look at operators in more detail in a future section.

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, " <<
4          "really long line" << endl; // one extra indentation for continuation line
5
6      cout << "This is another really, really, really, really, really, really, really, really, " <<
7          "really long line" << endl; // text aligned with the previous line for continuation line
8
9      cout << "This one is short" << endl;
10 }
```


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, " <<  
2      "really long line" << endl;
```

Not

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

Use whitespace to make your code easier to read.

Harder to read:

```
1  nCost = 57;  
2  nPricePerItem = 24;  
3  nValue = 5;  
4  nNumberOfItems = 17;
```

Easier to read:

```
1  nCost          = 57;  
2  nPricePerItem  = 24;  
3  nValue         = 5;  
4  nNumberOfItems = 17;
```

Harder to read:

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

Easier to read:

```
1  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  
2  cout << "Yeah!" << endl; // especially when all lined up
```

Harder to read:

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

Easier to read:

```
1 // cout and endl live in the iostream library
2 cout << "Hello world!" << endl;
3
4 // these comments are easier to read
5 cout << "It is very nice to meet you!" << endl;
6
7 // when separated by whitespace
8 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()

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

- 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**.

- return type
- Name
- Parameters
- but no function body

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

```

1  #include <iostream>
2
3  int add(int x, int y); // forward declaration of add() (using a function prototype)
4
5  int main()
6  {
7      using namespace std;
8      cout << "The sum of 3 and 4 is: " << add(3, 4) << endl; // this works because we for
ward declared add() above
9      return 0;
10 }
11
12 int add(int x, int y) // even though the body of add() isn't defined until here
13 {
14     return x + y;
15 }

```

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

```

1  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 allocated 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;
5  }
```

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;
15 }
```

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:

```
1  //#include "stdafx.h" // uncomment if using Visual Studio
2
3  int add(int x, int y)
4  {
5      return x + y;
6  }
```

- main.cpp:

```
1  //#include "stdafx.h" // uncomment if using Visual Studio
2  #include <iostream>
3
4  int main()
5  {
6      using namespace std;
7      cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
8      return 0;
9  }
```

```
1 // #include "stdafx.h" // uncomment if using Visual Studio
  #include <iostream>
2
3 int add(int x, int y); // needed so main.cpp knows that add() is a function declared elsewhere
4
5 int main()
6 {
7     using namespace std;
8     cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
9     return 0;
10 }
```

- Does forward declaration work?

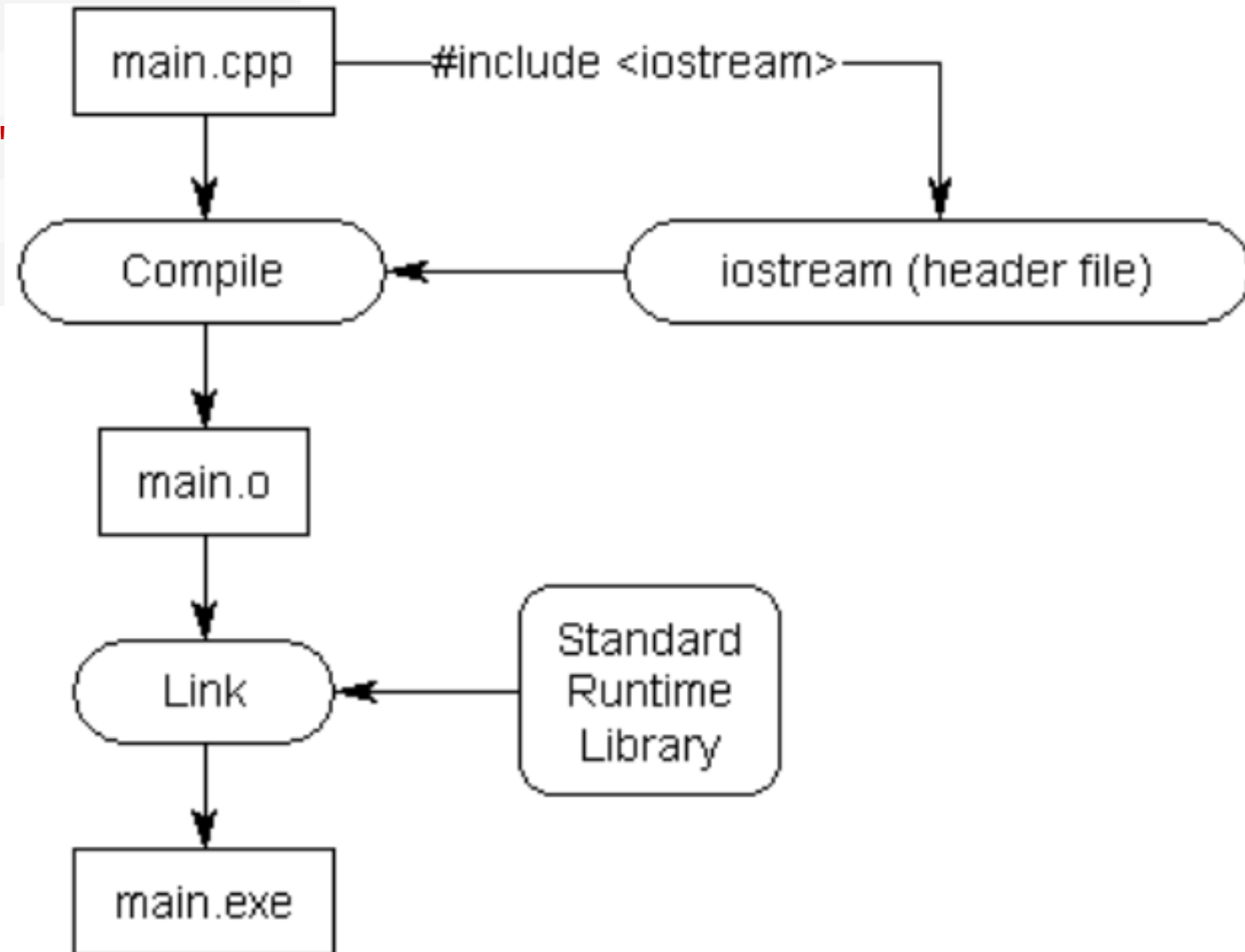
Header files

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
10 // This is the end of the header guard
11 #endif
```

- **header guard**

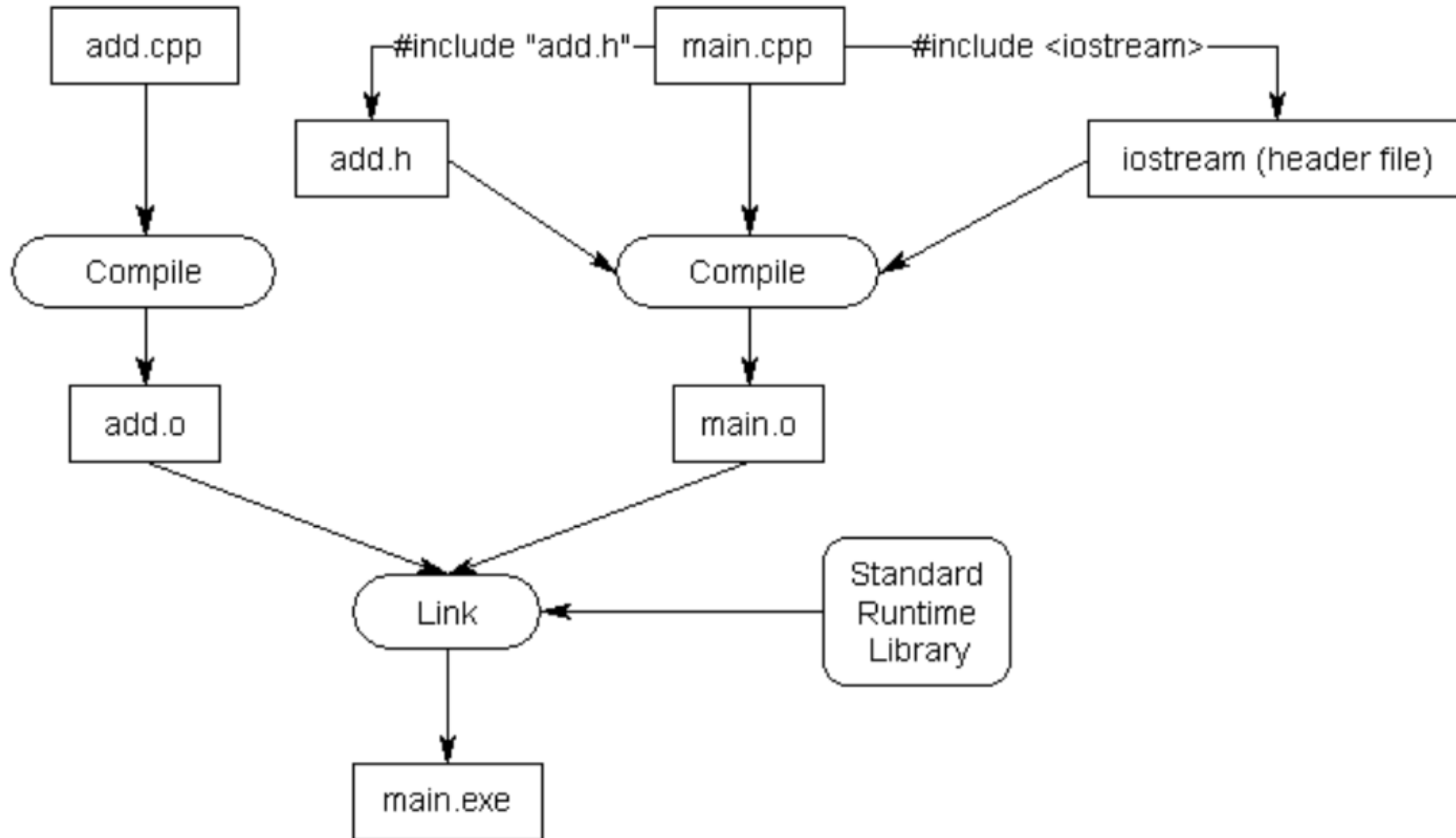
- prevent a given header file from being #included more than once from the same file.
- Discuss it later

Use your own header files

```
1  #include <iostream>
2  #include "add.h"
3
4  int main()
5  {
6      using namespace std;
7      cout << "The sum of 3 and 4 is " << add(3, 4) << endl;
8      return 0;
9  }
```

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.

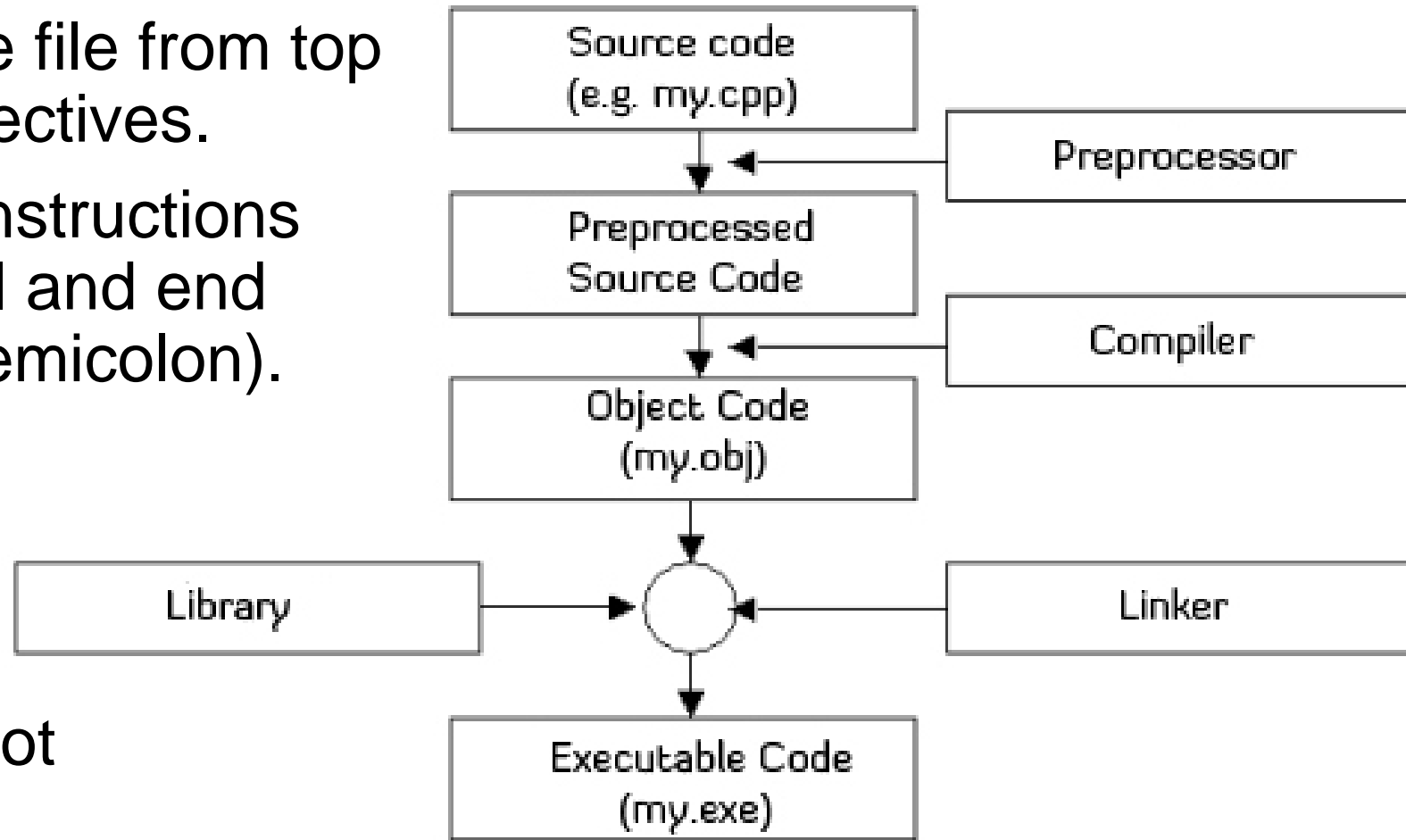
Header file best practices

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 }
```

- Directives are resolved before compilation
- file-by-file
- Once the preprocessor has finished, all directives from that file are replaced with ...
- This means?

Not printing!

main.cpp:

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

- **valid from the point of definition to the end of the file**
- How to define it once and use it in multiple files?

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?

math.h:

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

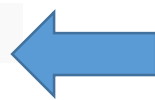
geometry.h:

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

main.cpp:

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

```
1 | int getSquareSides() // from math.h  
2 | {  
3 |     return 4;  
4 | }  
5 |  
6 | int getSquareSides() // from geometry.h  
7 | {  
8 |     return 4;  
9 | }  
10 |  
11 | int main()  
12 | {  
13 |     return 0;  
14 | }
```



Header guards

```
1  #ifndef SOME_UNIQUE_NAME_HERE
2  #define SOME_UNIQUE_NAME_HERE
3
4  // your declarations and definitions here
5
6  #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:

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

Updating our previous example with header guards

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:

```
1  #ifndef SQUARE_H
2  #define SQUARE_H
3
4  int getSquareSides()
5  {
6      return 4;
7  }
8
9  int getSquarePerimeter(int sideLength);
10
11 #endif
```

square.cpp:

```
1  #include "square.h" // square.h is included
2
3  int getSquarePerimeter(int sideLength)
4  {
5      return sideLength * getSquareSides();
6  }
```

main.cpp:

```
1  #include "square.h" // square.h is also included once here
2
3  int main()
4  {
5      std::cout << "a square has " << getSquareSides() << "sides" << std::endl;
6      std::cout << "a square of length 5 has perimeter length " << getSquarePerimeter(5) << std::endl;
7
8      return 0;
9  }
```

square.h:

```
1  #ifndef SQUARE_H
2  #define SQUARE_H
3
4  int getSquareSides()
5  {
6      return 4;
7  }
8
9  int getSquarePerimeter(int sideLength);
10
11 #endif
```

main.cpp:

```
1  #include "square.h" // square.h is also included once here
2
3  int main()
4  {
5      std::cout << "a square has " << getSquareSides() << "sides" << std::endl;
6      std::cout << "a square of length 5 has perimeter length " << getSquarePerimeter(5) << std::endl;
7
8      return 0;
9  }
```

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

square.h:

```
1  #ifndef SQUARE_H
2  #define SQUARE_H
3
4  int getSquareSides(); // forward declare
5  int getSquarePerimeter(int sideLength);
6
7  #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 project files.
- This is a good thing, because we often need to reference the contents of a given header from different project files.