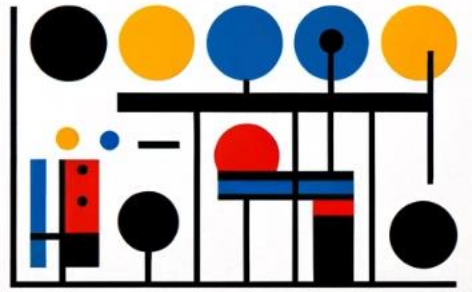
Object-Oriented Programming

Lecture 2: C++ Structure



Programming Paradigms

When to choose C++

> "If you're not at all interested in performance, shouldn't you be in the Python room down the hall?"
> — Scott Meyers (author of Effective Modern C++)

- Despite its many competitors C++ has remained popular for ~30 years and will continue to be so in the foreseeable future.
- Why?
  - Complex problems and programs can be effectively implemented
  - OOP works in the real world!
  - No other language quite matches C++'s combination of performance, expressiveness, and ability to handle complex programs.

- Choose C++ when:
  - Program performance matters
    - Dealing with large amounts of data, multiple CPUs, complex algorithms, etc.
  - Programmer productivity is less important
    - It is faster to produce working code in Python, R, Matlab or other scripting languages!
  - The programming language itself can help
  - organize your code
    - Not everything is a vector or matrix, right Matlab?
  - Access to libraries that will help with your
    - Ex. Nvidia's CUDA Thrust library for GPUs
  - Your group uses it already!

Programming Paradigms

Programming languages are classified in many ways based on their features:

- **Imperative**
  - **Structured/Procedural**
  - **Object-Oriented**
- **Declarative**
  - **Functional**
  - **Logic**

Procedural Programming

- In **procedural programming**, programs are mainly composed of three basic control structures:

  Sequence statements (execution in order)

  Selection statements (branching, if-else/switch)

  Iteration statements (looping, for/while)

After the source code get transformed to machine code, all program structures are reduced to simple and jumping/branching instructions.

Blocks and Subroutines

In addition to the basic control structures:

- **Blocks** are used to enable groups of statements to be treated as if they were one statement
- **Subroutines** (procedures, functions, methods, or subprograms) are used to allow a sequence to be referred to by a single statement

Object-Oriented Programming

- In object-oriented programming, programs are mainly composed of interrelated objects:
- An object is a computational entity which has both **state** and **behavior.**
- Objects are self-contained by bundling **data** and **operations** together, this notion is called **encapsulation**.
- By encapsulation, an object is mainly used by sending it a **message** via its **methods** (or member functions) Objects are often used for modeling things found in the real world

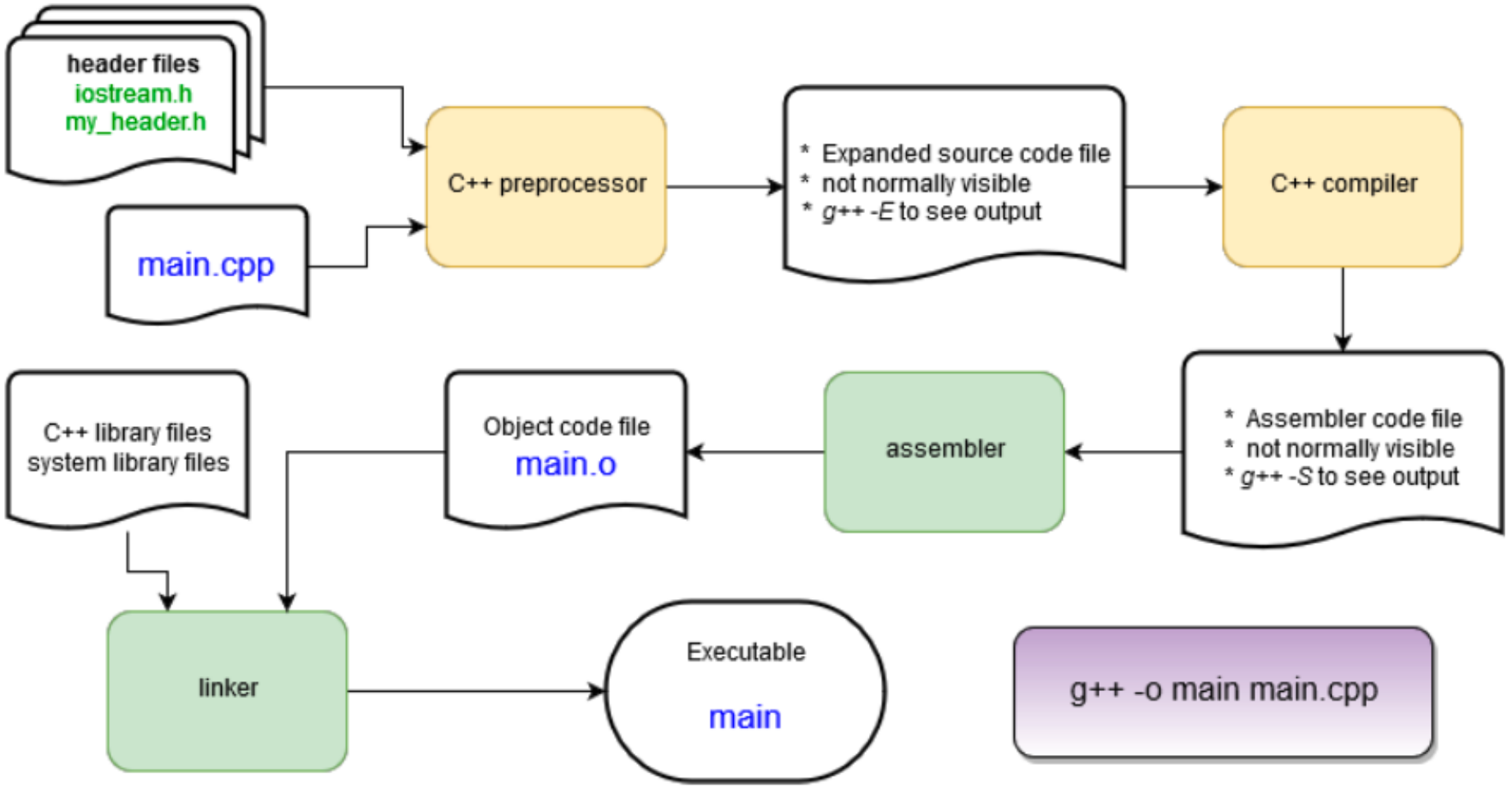
Objects

- An **object** can be created, stored and manipulated, **without** knowing its internal structure
- In C++, an object is created from a **class** and is an ***instance*** of a class
- A **class** is a user-defined type which is used as a blueprint for creating objects
- A **class** in C++ can be designed in a way that it behaves **"just like a built-in type"**

A **variable** (of a built-in type) can be considered an object. We sometimes used the term **"object"** to refer to a variable.

Extending Classes

- In OOP, a **class** can be defined and extended in many ways:
  - By **inheritance**, a class can be defined as an extension of existing classes, forming a class hierarchy
  - By overriding existing methods of existing classes, **polymorphism** can be achieved and objects from related types can be used in the same way with varying behavior
- **Encapsulation**, **inheritance**, and **polymorphism** are major concepts in OOP.


header files
iostream.h
my_header.h
main.cpp
C++ preprocessor
* Expanded source code file
* not normally visible
* g++ -E to see output
C++ compiler
* Assembler code file
* not normally visible
* g++ -S to see output
assembler
Object code file
main.o
C++ library files
system library files
linker
Executable
main
g++ -o main main.cpp

Working with Objects

Example: Interacting withStrings

```
  // 1. What are you expecting the program to do?
  // 2. What's wrong with the code?
  // 3. Does it compile?
  // 4. Does it work as expected?
  // 5. If not, what would you do to correct the program?

#include <iostream>

int main()
{
    std::cout << "Please enter P1 name: ";
    std::string p1_name;
    std::cin >> p1_name;
    std::cout << "Player 1: " << p1_name << std::endl;

    std::cout << "Please enter P2 name: ";
    std::string p2_name;
    std::cin >> p2_name;
    std::cout << "Player 2: " << p2_name << std::endl;
    return 0;
}
```

Slight change

- Let's put the message into some variables of type *string* and print some numbers.
- Things to note:
  - Strings can be concatenated with a + operator.
  - No messing with null terminators as in C
- Some string notes:
  - Access a string character by brackets or function:
    - msg[0] -> "H" or msg.at(0) -> "H"
    - C++ strings are *mutable* – they can be changed in place.

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world ;
    cout <<  msg << endl;
    msg[0] = 'h';
    cout <<  msg << endl;
    return 0;
}
```

Blocks and Scope of Variables

A block defines an extent to which an inner object/variable exists:

```cpp
#include <iostream>

int main()
{
    {
        std::cout << "Please enter P1 name: ";
        std::string p1_name;
        std::cin >> p1_name;

        std::cout << "Please enter P2 name: ";
        std::string p2_name;
        std::cin >> p2_name;
    }

    // `p1_name` and `p2_name` doesn't exist here!
    std::cout << "Player 1: " << p1_name << std::endl;
    std::cout << "Player 2: " << p2_name << std::endl;
    return 0;
}
```

Objects vs Variables

In some language (e.g. Rust), a primitive variable can be used like an object.

```rust
println!("{}", (-10_i32).abs());
```

In C++, some object can be used like a variable.

```cpp
std::complex<double> a = 2;
std::complex<double> b = 3i;
a += b; // `a` becomes `2 + 3i`
```

“auto” Keyword

- allows the compiler to automatically deduce the type of a variable from its initializer.
- When used in the context of object creation, auto can simplify the syntax and make the code more readable, especially when dealing with complex types.

```
#include <iostream>
#include <vector>

int main() {
    auto x = 5; // x is deduced to be of type int
    auto y = 3.14; // y is deduced to be of type double

    std::cout << "x: " << x << ", y: " << y << std::endl;

    std::vector<int> vec = {1, 2, 3, 4, 5};
    auto it = vec.begin(); // it is deduced to be std::vector<int>::iterator

    for (auto n : vec) { // n is deduced to be int
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

```cpp
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<std::string, std::vector<int>> mapOfVectors;
    mapOfVectors["one"] = {1, 2, 3};
    mapOfVectors["two"] = {4, 5, 6};

    // Without auto, declaring an iterator for this map would be verbose:
    // std::map<std::string, std::vector<int>>::iterator it = mapOfVectors.begin();

    // With auto, it's much simpler and just as effective:
    for (auto it = mapOfVectors.begin(); it != mapOfVectors.end(); ++it) {
        std::cout << it->first << ": ";
        for (auto val : it->second) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

C++ Pragramming Environments

Basic Syntax

- C++ syntax is very similar to C, Java, or C#. Here's a few things up front and we'll cover more as we go along.
- Curly braces are used to denote a code block:

```
{ … some code … }
```

- Statements end with a semicolon:

```
int a ;
a = 1 + 3 ;
```

- Comments are marked for a single line with a ***//*** or for multilines with a pair of ***/**** and ****/*** :

```
// this is a comment.
/* everything in here
        is a comment */
```

- Variables can be declared at any time in a code block.

```
void my_function() {
   int a ;
   a=1 ;
   int b;
}
```

- Functions are sections of code that are called from other code. Functions always have a return argument type, a function name, and then a list of arguments:

```
int my_function(int x) {
    return x ;
}
```

```
// No arguments? Still need ()
void my_function() {
        /* do something...
           but a void value means the
           return statement can be skipped.*/
}
```

- Variables are declared with a type and a name:

```
// Usually enter the type
int x = 100;
float y;
vector<string> vec ;
// Sometimes it can be inferred
auto z = x;
```

- A sampling of Operators:
  - Arithmetic: + - * / % ++ --
  - Logical: && (AND) ||(OR) !(NOT)
  - Comparison: == > < >= <= !=

Built-in (aka primitive or intrinsic) Types

- “primitive” or “intrinsic” means these types are not objects
- Here are the most commonly used types.
- Note: The exact bit ranges here are **platform and compiler dependent**!
  - Typical usage with PCs, Macs, Linux, etc. use these values
  - Variations from this table are found in specialized applications like embedded system processors.

Name	Name	Value
char	unsigned char	8-bit integer
short	unsigned short	16-bit integer
int	unsigned int	32-bit integer
long	unsigned long	64-bit integer
bool		true or false

Name	Value
float	32-bit floating point
double	64-bit floating point
long long	128-bit integer
long double	128-bit floating point

http://www.cplusplus.com/doc/tutorial/variables/

Need to be sure of integer sizes?

- In the same spirit as using *integer(kind=8)* type notation in Fortran, there are type definitions that exactly specify exactly the bits used. These were added in C++11.
- These can be useful if you are planning to port code across CPU architectures (ex. Intel 64-bit CPUs to a 32-bit ARM on an embedded board) or when doing particular types of integer math.
- For a full list and description see: http://www.cplusplus.com/reference/cstdint/

#include <cstdint>

Name	Name	Value
int8_t	uint8_t	8-bit integer
int16_t	uint16_t	16-bit integer
int32_t	uint32_t	32-bit integer
int64_t	uint64_t	64-bit integer

Type Casting

- C++ is strongly typed. It will auto-convert a variable of one type to another in a limited fashion: if it will not change the value.

```
short x = 1 ;
int y = x ;    // OK
short z = y ; // NO!
```

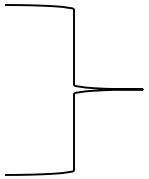
- Conversions that don't change value: increasing precision (float → double) or integer → floating point of at least the same precision.
- C++ allows for C-style type casting with the syntax: (new type) expression

```
double x = 1.0 ;
int y = (int) x ;
float z = (float) (x / y) ;
```

- In addition to this C++ offers 4 different variations in a C++ style.

Type Casting

- static_cast<new type>(expression)
- dynamic_cast<new type>(expression)
- const_cast<new type>(expression)
- reinterpret_cast<new type>(expression)



"unsafe": the compiler will not protect you here.

```
int i = 10;
float f1 = static_cast<float>(i);  // C++ style cast
float f2 = (float)i;               // C-style cast
```

static_cast

- The most commonly used type of cast in C++.
- It performs implicit conversions between types, such as non-const to const modifications, upcasts (from a derived class to its base class), and conversions between numeric types (like int to float, or vice versa).

```
#include <iostream>

int main() {
    int i = 10;
    float f = static_cast<float>(i); // Converts int to float
    std::cout << "Integer: " << i << ", Float: " << f << std::endl;
    return 0;
}
```

dynamic_cast

- Used for handling polymorphism.
- Convert a pointer or reference from one class type to another within an inheritance hierarchy.

```
#include <iostream>

class Base {
    virtual void print() {} // Making Base polymorphic
};

class Derived : public Base {
    void print() override { std::cout << "Derived class" << std::endl; }
};

int main() {
    Base *b = new Derived();
    Derived *d = dynamic_cast<Derived*> (b); // Safe downcasting
    if (d) {
        d->print();  // Correctly calls Derived::print()
    }
    delete b;
    return 0;
}
```

const_cast

- Add or remove the const qualifier from a variable.

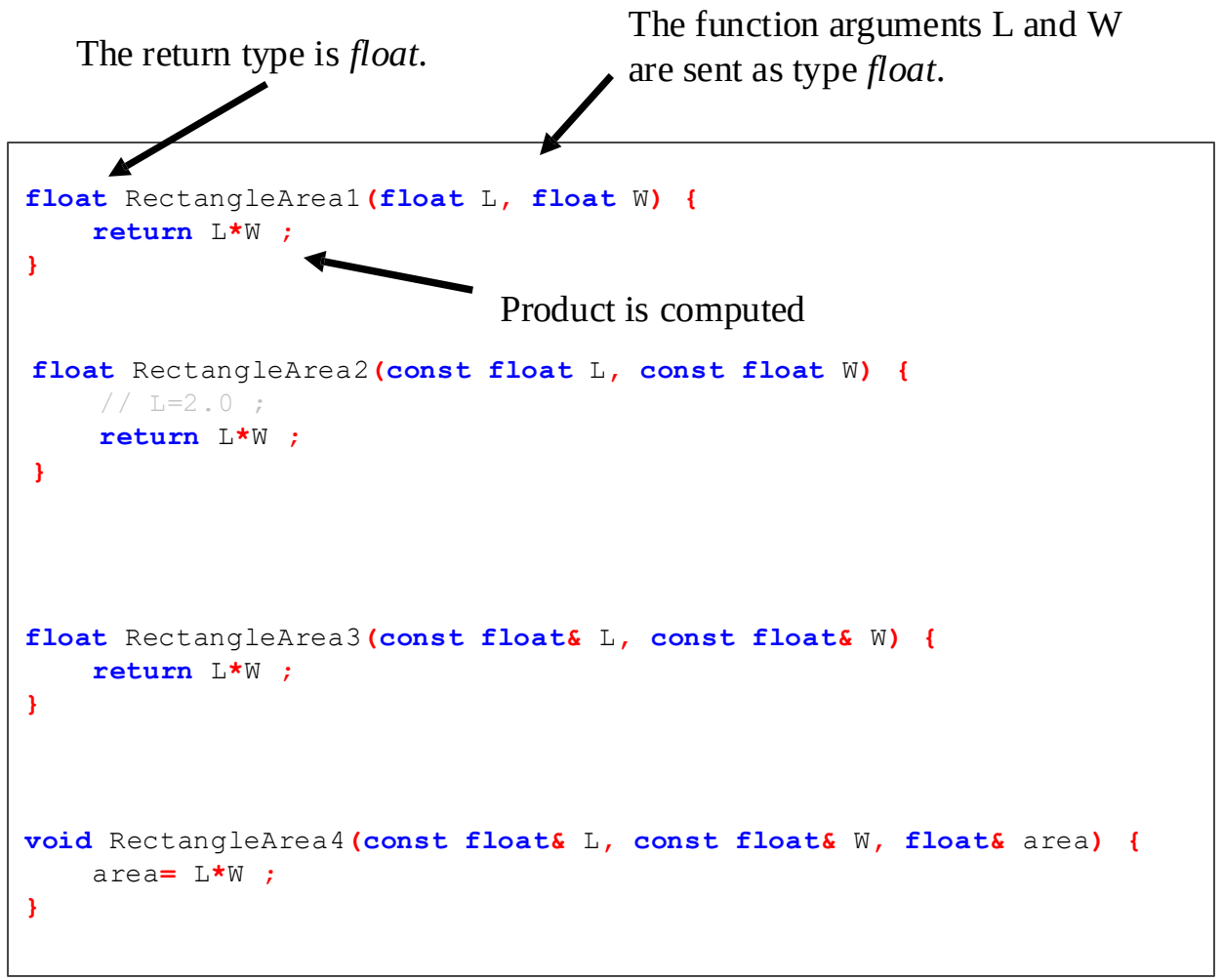
```
#include <iostream>

void print(char *str) {
    std::cout << str << std::endl;
}

int main() {
    const char *cstr = "Hello World";
    char *modStr = const_cast<char*>(cstr);
    modStr[0] = 'J'; // Dangerous: modifying a string literal is undefined
behavior
    print(modStr);
    return 0;
}
```

reinterpret_cast

- Powerful casting operator that should be used carefully. It converts any pointer type to any other pointer type, even if the classes are not related to each other.

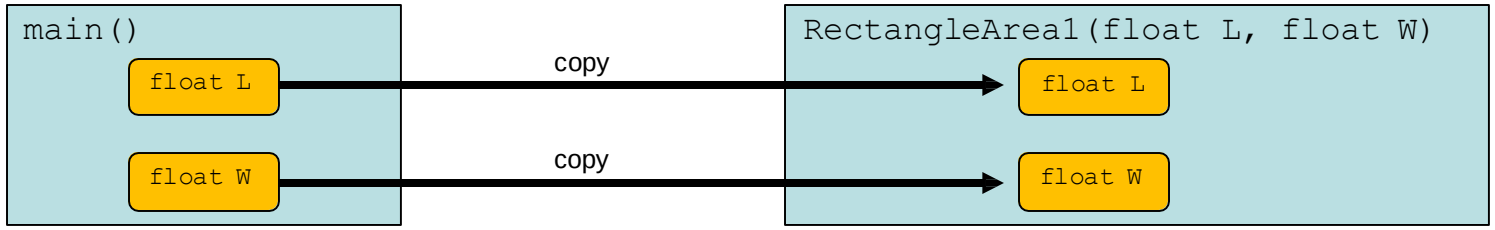
```
#include <iostream>

int main() {
    int *p = new int(10);
    std::size_t addr = reinterpret_cast<std::size_t>(p);  // Storing pointer value in integer
    std::cout << "The address in integer form: " << addr << std::endl;

    int *q = reinterpret_cast<int*>(addr);  // Converting back to pointer
    std::cout << "Value at pointer: " << *q << std::endl;
    delete p;
    return 0;
}
```

Functions



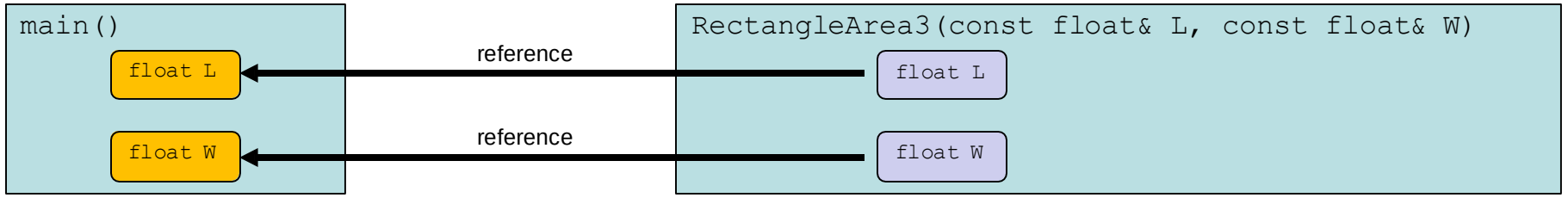

```
float RectangleArea1(float L, float W) {
    return L*W ;
}

float RectangleArea2(const float L, const float W) {
    // L=2.0 ;
    return L*W ;
}



float RectangleArea3(const float& L, const float& W) {
    return L*W ;
}



void RectangleArea4(const float& L, const float& W, float& area) {
    area= L*W ;
}
```

Pass by Value



- C++ defaults to *pass by value* behavior when calling a function.
- The function arguments are **copied** when used in the function.
- Changing the value of L or W in the RectangleArea1 function does **not** effect their original values in the main() function
- When passing objects as function arguments it is important to be aware that potentially large data structures are automatically copied!

Pass by Reference



- *Pass by reference* behavior is triggered when the & character is used to modify the type of the argument.
- Pass by reference function arguments are **NOT** copied. Instead the compiler sends a *pointer* to the function that references the memory location of the original variable. The syntax of using the argument in the function does not change.
- Pass by reference arguments almost always act just like a pass by value argument when writing code **EXCEPT** that changing their value changes the value of the original variable!!
- The *const* modifier can be used to prevent changes to the original variable in main().

void does not return a value.

```
void RectangleArea4(const float& L, const float& W, float& area) {
    area= L*W ;
}
```

- In RectangleArea4 the pass by reference behavior is used as a way to return the result without the function returning a value.
- The value of the *area* argument is modified in the main() routine by the function.
- This can be a useful way for a function to return multiple values in the calling routine.

- In C++ arguments to functions can be objects…which can contain any quantity of data you've defined!
  - Example: Consider a string variable containing 1 million characters (approx. 1 MB of RAM).
    - Pass by value requires a copy – 1 MB.
    - Pass by reference requires 8 bytes!
- Pass by value could potentially mean the accidental copying of large amounts of memory which can greatly impact program memory usage and performance.
- When passing by reference, use the *const* modifier whenever appropriate to protect yourself from coding errors.
  - Generally speaking – use *const* anytime you don't want to modify function arguments in a function.

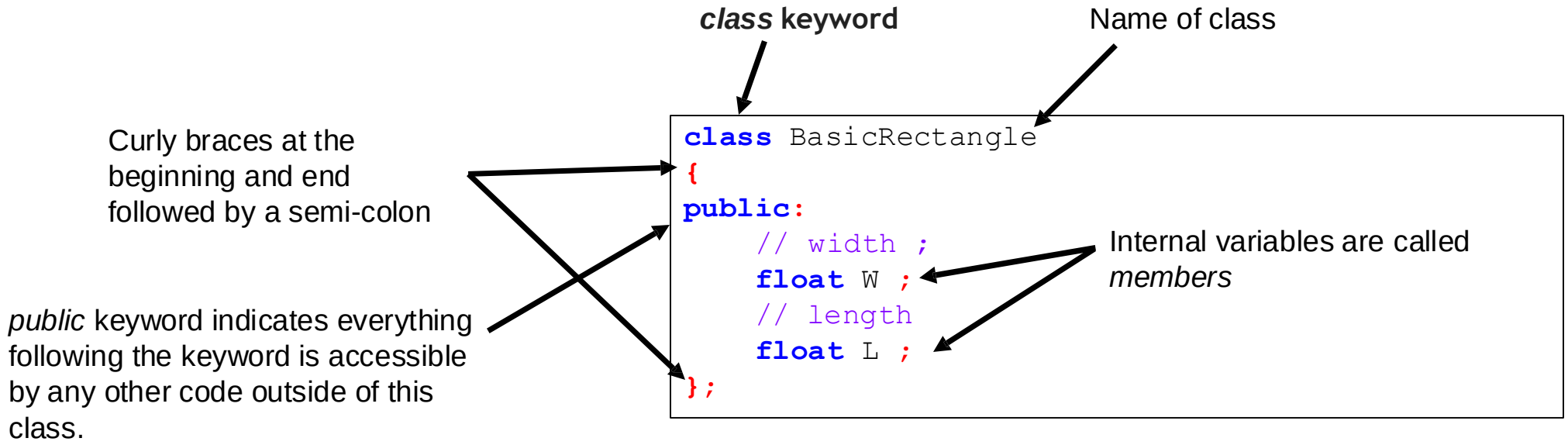
> "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off." – Bjarne Stroustrop

A first C++ class

- In the main.cpp, we'll define a class called BasicRectangle
- First, just the basics: length and width
- Enter the code on the right before the main() function in the main.cpp file (copy & paste is fine) and create a BasicRectangle object in main.cpp:

```
#include <iostream>

using namespace std;

class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
};

int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```

Basic C++ Class Syntax



The class can now be used to declare an object named *rectangle*. The width and length of the rectangle can be set.

```
BasicRectangle rectangle ;
rectangle.W = 1.0 ;
rectangle.L = 2.0 ;
```

Accessing data in the class

- Public members in an object can be accessed (for reading or writing) with the syntax:
  - object.member

```
int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```

- Next let's add a function inside the object (called a *method*) to calculate the area.

method Area does not take any arguments, it just returns the calculation based on the object members.

Methods are accessed just like members:

object.method(arguments)

```
class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
    float Area() {
        return W * L ;
    }

};

int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 21.0 ;
    rectangle.L = 2.0 ;

    cout << rectangle.Area() << endl ;

    return 0;
}
```

Basic C++ Class Summary

- C++ classes are defined with the keyword *class* and must be enclosed in a pair of curly braces **plus a semi-colon**:

  class ClassName { …. } **;**

- The *public* keyword is used to mark members (variables) and methods (functions) as accessible to code outside the class.

- The combination of data and the functions that operate on it is the OOP concept of *encapsulation.*

Encapsulation in Action

- In C – calculate the area of a few shapes…

```
/* assume radius and width_square are assigned
   already ; */
float a1 = AreaOfCircle(radius) ;   // ok
float a2 = AreaOfSquare(width_square) ; // ok
float a3 = AreaOfCircle(width_square) ; // !! OOPS
```

- In C++ with Circle and Rectangle classes…not possible to miscalculate.
  - Well, provided the respective Area() methods are implemented correctly!

```
Circle c1 ;
Rectangle r1 ;
// ... assign radius and width ...
float a1 = c1.Area() ;
float a2 = r1.Area() ;
```

Now for a “real” class

- Defining a class in the main.cpp file is not typical.
- Two parts to a C++ class:
  - Header file (my_class.h)
    - Contains the interface (definition) of the class – members, methods, etc.
    - The interface is used by the compiler for type checking, enforcing access to private or protected data, and so on.
    - Also useful for programmers when *using* a class – no need to read the source code, just rely on the interface.
  - Source file (my_class.cc)
    - Compiled by the compiler.
    - Contains implementation of methods, initialization of members.
  - In some circumstances there is no source file to go with a header file.

rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H


class Rectangle
{
    public:
        Rectangle();
        virtual ~Rectangle();

    protected:

    private:
};

#endif // RECTANGLE_H
```

rectangle.cpp

```
#include "rectangle.h"

Rectangle::Rectangle()
{
    //ctor
}

Rectangle::~Rectangle()
{
    //dtor
}
```

- 2 files are automatically generated: rectangle.h and rectangle.h.cpp

Modify rectangle.h

- As in the sample *BasicRectangle*, add storage for the length and width to the header file. Add a *declaration* for the Area method.

```
#ifndef RECTANGLE_H
#define RECTANGLE_H


class Rectangle
{
    public:
        Rectangle();
        virtual ~Rectangle();

        float m_length ;
        float m_width ;

        float Area() ;
    protected:

    private:
};
#endif // RECTANGLE_H
```

Private

- Can only be accessed within the same class.
- Cannot be accessed from outside the class, including derived classes
- Help enforce encapsulation by hiding implementation details

Protect

- Can be accessed within the same class and by derived classes.
- Cannot be accessed from outside the class hierarchy.
- Useful when you want to share certain members with child classes while keeping them hidden from the public

rectangle.cpp

- The Area() method now has a basic definition added.
- The syntax:

class::method

tells the compiler that this is the code for the Area() method declared in rectangle.h

- Now take a few minutes to fill in the code for Area().
  - Hint – look at the code used in BasicRectangle...

```
#include "rectangle.h"

Rectangle::Rectangle()
{
    //ctor
}

Rectangle::~Rectangle()
{
    //dtor
}

float Rectangle::Area()
{

}
```

Program Exit Code

```cpp
// success.cpp
int main()
{
    return 0;
}
```

```cpp
// failed.cpp
int main()
{
    return 1;
}
```

Example Session

```
$ g++ -o success -Wall -Wextra success.cpp
$ ./success
$ echo $?
0
$ g++ -o failed -Wall -Wextra failed.cpp
$ ./failed
$ echo $?
1
```

Tips: For Windows, check for **ERRORLEVEL**.

Improving the FrameProgram

Overall structure (starting from previous version):

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

int main()
{
    // ask for a person's name
    std::cout << "Please enter your first name: ";

    // read the name
    std::string name;
    std::cin >> name;

    // build the message that we intend to write
    const std::string greeting = "Hello, " + name + "!";

    // we have to rewrite this part ...

    return 0;
}
```

Compute the Number of Rows

```
  // the number of blanks
  // surrounding the greeting
const int pad = 1;

  // the number of rows and columns to write
const int rows = pad * 2 + 3;
```

```
  // constexpr is also `const`
constexpr int pad = 2;
constexpr int rows = pad * 2 + 3;
```

```
*****   | top border
*    *->| top pad
* x *
*    *--->| bottom pad
*****     | bottom border

Rows = (1 * 2) + 3 = 5
```

```
*******
*     *->| top pad
*     *  |
* xxx *
*     *--->|
*     *     | bottom pad
*******

Rows = (2 * 2) + 3 = 7
```

The `while` Statement

Print rows of output

```
constexpr int pad = 1;
constexpr int rows = pad * 2 + 3;

// separate the output from the input
cout << endl;

// write `rows' rows of output
int r = 0;

// invariant: we have written `r' rows so far
while (r != rows) {
    // write a row of output
    std::cout << std::endl;
    ++r;
}
```

```
while (condition)
    statement
```

The `if` Statement

Print a row

```
    // invariant: we have written `c' characters
    //            so far in the current row
    while (c != cols) {
        // is it time to write the greeting?
        if (r == pad + 1 && c == pad + 1) {
            cout << greeting;
            c += greeting.size();
        }
        else {
            // are we on the border?
            if (r == 0 || r == rows - 1 ||
                c == 0 || c == cols - 1)
                cout << "*";
            else
                cout << " ";
            ++c;
        }
    }
```

```
if (condition)
    statement
```

```
if (condition)
    statement1
else
    statement2
```

The `for` Statement

```
for (init-statement condition; expression)
    statement
```

```
{
    init-statement
    while (condition) {
        statement
        expression;
    }
}
```

```
  // `r' takes on the values in [0, rows)
for (int r = 0; r != rows; ++r) {
    // stuff that doesn't change
    // the value of `r'
}
```

```
{
    int r = 0;
    while (r != rows) {
        // ...
        ++r;
    }
}
```

for statements is used as a shorthand way of writing a loop

The Complete Framing Program (1)

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

  // say what standard-library names we use
using std::cin;          using std::endl;
using std::cout;         using std::string;

int main()
{
    // ask for the person's name
    cout << "Please enter your first name: ";

    // read the name
    string name;
    cin >> name;

    // build the message that we intend to write
    const string greeting = "Hello, " + name + "!";

    // the number of blanks surrounding the greeting
    constexpr int pad = 1;

    // the number of rows and columns to write
    constexpr int rows = pad * 2 + 3;
    const string::size_type cols = greeting.size() + pad * 2 + 2;

    // write a blank line to separate the output from the input
    cout << endl;

    // ...
```

The Complete Framing Program (2)

```
    // ...

    // write `rows' rows of output
    // invariant: we have written `r' rows so far
    for (int r = 0; r != rows; ++r) {
        string::size_type c = 0;

        // invariant: we have written `c' characters
        //            so far in the current row
        while (c != cols) {
            // is it time to write the greeting?
            if (r == pad + 1 && c == pad + 1) {
                cout << greeting;
                c += greeting.size();
            }
            else {
                // are we on the border?
                if (r == 0 || r == rows - 1 ||
                    c == 0 || c == cols - 1)
                    cout << "*";
                else
                    cout << " ";
                ++c;
            }
        }

        cout << endl;
    }
    return 0;
}
```

Loop Counter

r takes on the values in **[0, rows)**

```
for (int r = 0; r != rows; ++r) {
    // write a row
}
```

r takes on the values in **[1, rows]**

```
for (int r = 1; r <= rows; ++r) {
    // write a row
}
```

The number of iterations is the same in both cases.

Tips: In C++, we often **prefer** to count from **0** and use the half-open range **[0, n)** to control the loop. We also **prefer ++r** over **r++** whenever we have a choice.

Computing StudentGrades (1)

```
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>

using std::cin;                    using std::setprecision;
using std::cout;                   using std::string;
using std::endl;                   using std::streamsize;

int main()
{
    // ask for and read the student's name
    cout << "Please enter your first name: ";
    string name;
    cin >> name;
    cout << "Hello, " << name << "!" << endl;

    // ask for and read the midterm and final grades
    cout << "Please enter your midterm and final exam grades: ";
    double midterm, final;
    cin >> midterm >> final;

    // ask for the homework grades
    cout << "Enter all your homework grades, "
            "followed by end-of-file: ";

    // ...
```

Computing StudentGrades (2)

```
    // ...

    // the number and sum of grades read so far
    int count = 0;
    double sum = 0;

    // a variable into which to read
    double x;

    // invariant:
    //     we have read `count' grades so far, and
    //     `sum' is the sum of the first `count' grades
    while (cin >> x) {
        ++count;
        sum += x;
    }

    // write the result
    streamsize prec = cout.precision();
    cout << "Your final grade is " << setprecision(3)
         << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
         << setprecision(prec) << endl;
    return 0;
}
```

Using Medians to Compute Grades (1)

```
#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>
#include <vector>

using std::cin;              using std::sort;
using std::cout;             using std::streamsize;
using std::endl;             using std::string;
using std::setprecision;     using std::vector;

int main()
{
    // ask for and read the student's name
    cout << "Please enter your first name: ";
    string name;
    cin >> name;
    cout << "Hello, " << name << "!" << endl;

    // ask for and read the midterm and final grades
    cout << "Please enter your midterm and final exam grades: ";
    double midterm, final;
    cin >> midterm >> final;

    // ask for and read the homework grades
    cout << "Enter all your homework grades, "
            "followed by end-of-file: ";

    // ...
```

Using Medians to Compute Grades (2)

```
    // ...

    vector<double> homework;
    double x;
    // invariant: `homework' contains all the homework grades read so far
    while (cin >> x)
        homework.push_back(x);

    // check that the student entered some homework grades
    typedef vector<double>::size_type vec_sz;
    vec_sz size = homework.size();
    if (size == 0) {
        cout << endl << "You must enter your grades.  "
                        "Please try again." << endl;
        return 1;
    }

    // sort the grades
    sort(homework.begin(), homework.end());

    // compute the median homework grade
    auto mid = size / 2;
    double median;
    median = size % 2 == 0 ? (homework[mid] + homework[mid-1]) / 2
                           : homework[mid];

    // compute and write the final grade
    auto prec = cout.precision();
    cout << "Your final grade is " << setprecision(3)
         << 0.2 * midterm + 0.4 * final + 0.4 * median
         << setprecision(prec) << endl;
    return 0;
}
```


Q&A