Lecture 14

Introduction to C++ and Object-Oriented Programming

Fundamentals of Computer and Programming

Instructor: Morteza Zakeri, Ph.D. (m-zakeri@live.com)

Spring 2024

Modified Slides from Dr. *Brian Gregor*

Computer Engineering Department, Amirkabir University of Technology

Outline

- Very brief history of C++
- Definition of object-oriented programming (OOP)
- When C++ is a good choice
- First program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

Very brief history of C++

1962
Simula I was invented by Kristen Nygaard and Ole-Johan Dahl as a *simulation language*

1967
Simula 67 developed as the first *object-oriented* language

1969-1973
The C language was invented by Dennis Ritchie at Bell Labs

1972
D. Ritchie and Ken Thompson re-write the Unix OS in C

1979
Bjarne Stroustrop began developing "C with Classes"

1983
"C with Classes" renamed to C++

1985
1st commercial C++ compiler, Cfront, released by AT&T

1989
C++ 2.0 standard released.

2011
Major update: C++11 standard released

2014
Minor update: C++14 released.

C

C++

For details more check out [A History of C++: 1979−1991]

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

Object-oriented programming (OOP)

- Seeks to define a program in terms of the *things (objects)* in the problem
  - files, molecules, buildings, cars, people, etc.,
  - what they need, and what they can do.
- **Data** beside **operations**
- Modeling real-world phenomena's

class GasMolecule

- **Data:**
  - **molecular weight, structure, common names, etc.**
- **Methods:**
  - **IR(wavenumStart, wavenumEnd) : return IR emission spectrum in range**

Objects
(*instances* of a class)

```
GasMolecule ch4
GasMolecule co2

spectrum =
ch4.IR(1000,3500)
Name = co2.common_name
```

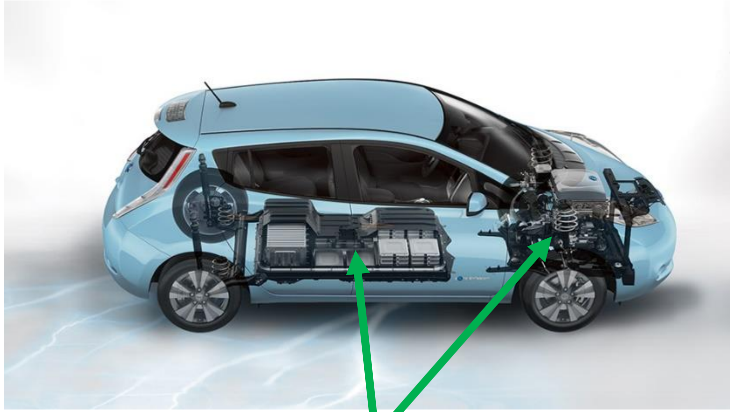
"pseudo-code"

Object-oriented programming

- OOP defines *classes* to represent these things.
- Classes can contain **data** and **methods** (internal/in-class functions).
- Classes control access to internal data and methods.
- A **public interface** is used by external code when using the class.
- This is a **highly effective** way of modeling **real-world problems** inside of a computer program.

"Class Car"

public interface



private data and methods

Characteristics of C++

- C++ is **object oriented**
  - With support for many programming styles (**procedural**, **functional**, *etc.*)
- C++ is **compiled**.
  - A separate program, the **compiler**, is used to turn C++ source code into a form directly executed by the CPU.
- C++ is **strongly typed** and **unsafe**
  - Conversions between variable types must be made by the programmer (strong typing) but can be circumvented when needed (unsafe)
- C++ is **C compatible**
  - call C libraries directly and C code is nearly 100% valid C++ code.
- C++ is capable of very **high performance**
  - The programmer has a very large amount of control over the program execution
- C++ has no **automatic memory management**
  - The programmer is in control of memory usage

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

Why 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.

When to choose C++

- 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
    - *Ex*. In C++ your objects can closely model elements of your problem.
  - Access to libraries
    - *Ex.* Nvidia's CUDA Thrust library for GPUs
  - Your group uses it already!

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First C++ program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

Hello, World! explained

main.cpp

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

The *main* routine – the start of every C++ program!

It returns an integer value to the operating system and (in this case) takes no arguments: `main()`

The return statement returns an integer value to the operating system after completion.

0 means “no error”.

C++ programs must return an integer value.

Hello, World! explained

main.cpp

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

loads a ***header*** file containing function and class definitions

- Loads a ***namespace*** called *std*.
- Namespaces are used to separate sections of code for programmer convenience.
- To save typing we'll always use this line in this tutorial.

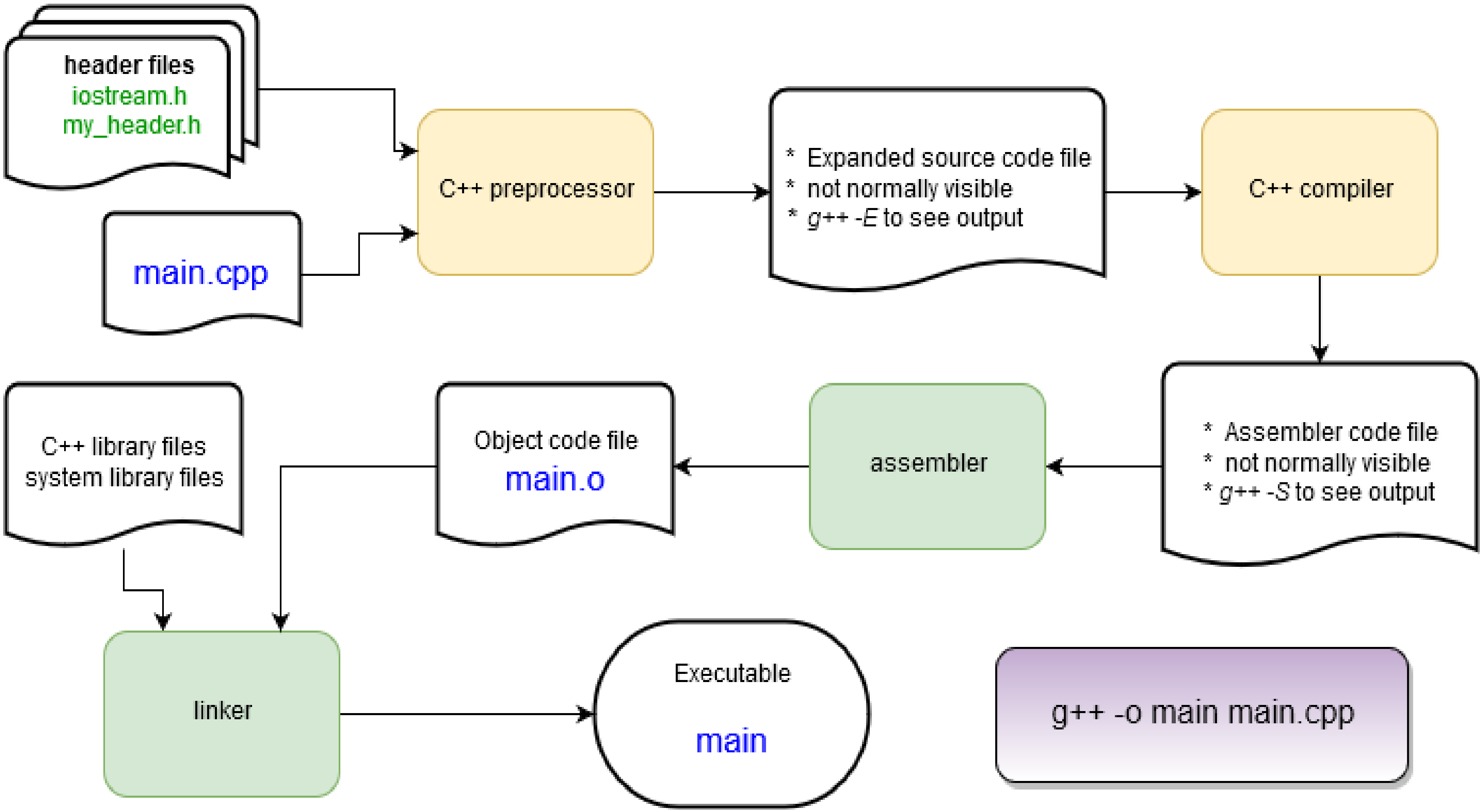
- *cout* is the *object* that writes to the stdout device, *i.e.,* the console window.
- It is part of the C++ standard library.
- Without the "using namespace std;" line this would have been called as *std::cout*. It is defined in the *iostream* header file.
- << is the C++ *insertion operator*. It is used to pass characters from the right to the object on the left.
- *endl* is the C++ newline character.

C++ Reserved Keywords

```
alignas        constexpr      mutable            switch template
alignof        constinit      namespace          this
and            const_cast     new                thread_local
and_eq         continue       noexcept           throw
asm auto       decltype       not not_eq         true try
bitand         default        nullptr            typedef
bitor          delete         operator or        typeid
bool           do             or_eq              typename
break          double         private            union
case           dynamic_cast   protected          unsigned
catch          else           public             using
char           enum           register           virtual
char8_t        explicit       reinterpret_cast   void
char16_t       export         requires           volatile
char32_t       extern         return             wchar_t
class          false          short              while  xor
co_await       float for      signed             xor_eq
co_return      friend         sizeof             final*
co_yield       goto           static             import*
compl          if             static_assert      module*
concept        inline         static_cast        override*
const          int long       struct
consteval
```

* Note: context sensitive

Behind the Scenes: The Compilation Process



Header Files

- C++ (along with C) uses *header files* as to hold definitions for the compiler to use while compiling.
- A source file (file.cpp) contains the code that is compiled into an object file (file.o).
- The header (file.h) is used to tell the compiler what to expect when it assembles the program in the linking stage from the object files.
- Source files and header files can refer to any number of other header files.

C++ language headers aren't referred to with the **.h** suffix. <iostream> provides definitions for I/O functions, including the *cout* function.

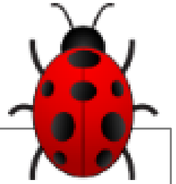
```
#include <iostream>
using namespace std;
int main(){
  string hello = "Hello";
  string world = "world!";
  string msg = hello + " " + world;
  cout <<  msg << endl;
  msg[0] = 'h';
  cout <<  msg << endl;
  return 0;
}
```

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 or *strcat()* 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;
}
```

A first C++ class: *string*

- *string* is not a basic type (more on those later), it is a **class**.
- `string hello` creates an *instance* of a string called "hello".
- `hello` is an object.
- Remember that a class defines some data and a set of functions (methods) that operate on that data.

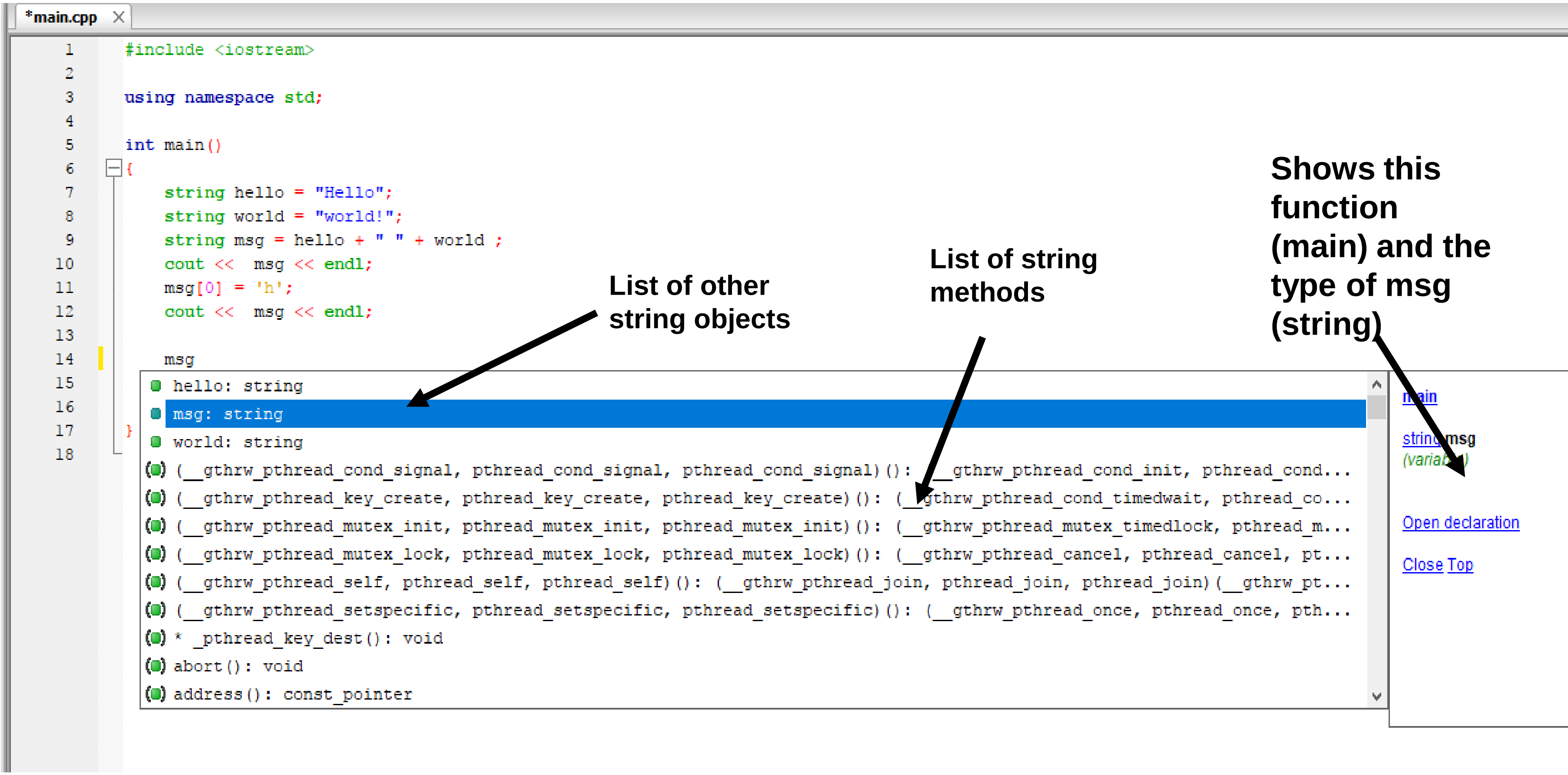
```
#include <iostream>

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;
}
```

A first C++ class: *string*

- Update the code as you see here.
- After the last character is entered, **IDE** will display some info about the string class.
- If you click or type something else just delete and re-type the last character.
- Ctrl-space will force the list to appear.

```
#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;

  msg

  return 0;
}
```

A first C++ class: *string*



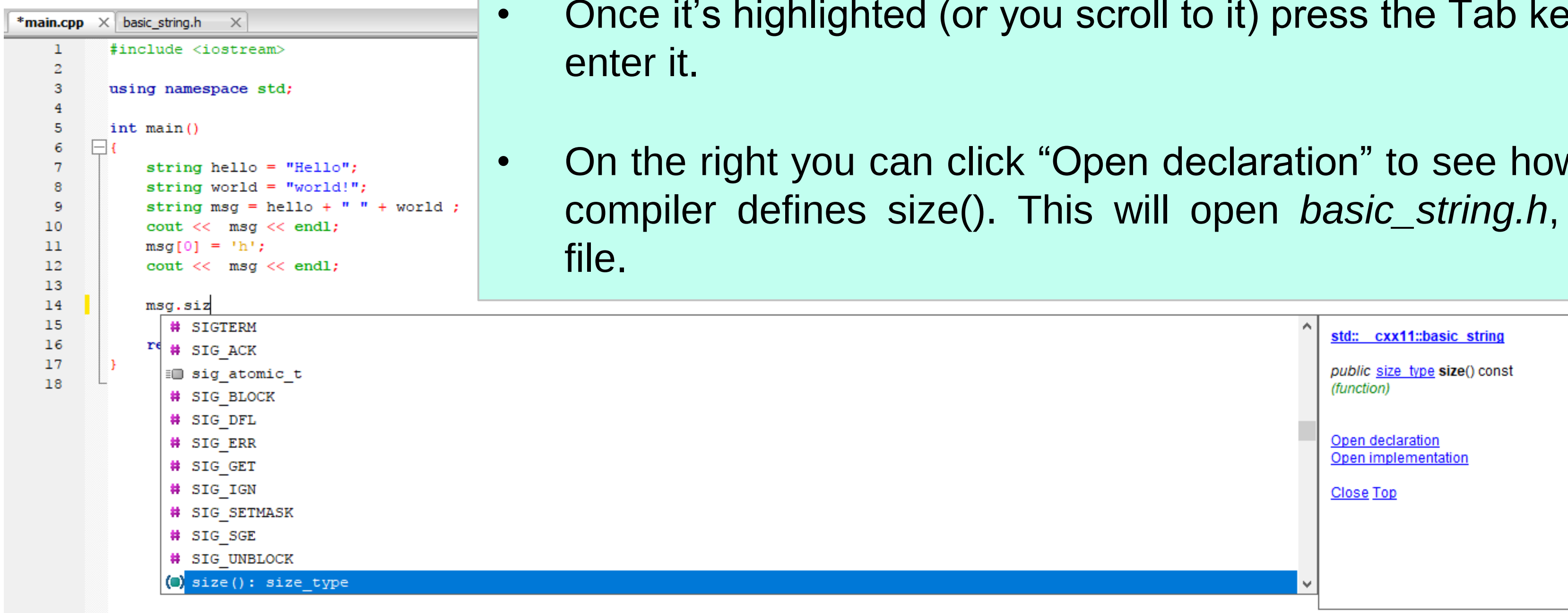
Next: let's find the size() method without scrolling for it.

A first C++ class: *string*

- Start typing “msg.size()” until it appears in the list.

- Once it’s highlighted (or you scroll to it) press the Tab key to auto-enter it.

- On the right you can click “Open declaration” to see how the C++ compiler defines size(). This will open *basic_string.h*, a built-in file.



A first C++ class: *string*

- Tweak the code to print the number of characters in the string, build, and run it.

- From the point of view of main(), the *msg* object has hidden away its means of tracking and retrieving the number of characters stored.

- Note: while the string class has a **huge** number of methods your typical C++ class has far fewer!

```cpp
#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 ;

    cout << msg.size() << endl ;

    return 0;
}
```

- Note that *cout* prints integers without any modification!

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First C++ program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

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** (like the main() function):

```
{ … 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;
}
```

Basic Syntax

- 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 separated by commas:
- A *void* type means the function does not return a value.

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

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

- Variables are declared with a type and a name:

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

Basic Syntax

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

- Sometimes these can have special meanings beyond arithmetic, for example the "+" is used to **concatenate strings**.

- What happens when a **syntax error** is made?
  - The compiler will complain and refuse to compile the file.
  - The error message *usually* directs you to the error but sometimes the error occurs before the compiler discovers syntax errors so you hunt a little bit.

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

Reference and Pointer Variables

The object *hello* occupies some computer memory.

```
string hello = "Hello";

string *hello_ptr = &hello;

string &hello_ref = hello;
```

The asterisk indicates that *hello_ptr* is a pointer to a string. *hello_ptr* variable is assigned the memory address of object *hello* which is accessed with the "&" syntax.

The & here indicates that *hello_ref* is a reference to a string. The *hello_ref* variable is assigned the memory address of object *hello* automatically.

- Variable and object values are stored in particular locations in the computer's memory.
- Reference and pointer variables **store the memory location of other variables.**
- Pointers are found in C. References are a C++ variation that makes pointers easier and safer to use.

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) ;
```

Type Casting

```
static_cast<new type>( expression )
```

- This is exactly equivalent to the C style cast.
- This identifies a cast **at compile time**.
- This will allow casts that reduce precision (ex. double → float)
- ~99% of all your casts in C++ will be of this type.

```
double d = 1234.56 ;
float f = static_cast<float>(d) ;
// same as
float g = (float) d ;
```

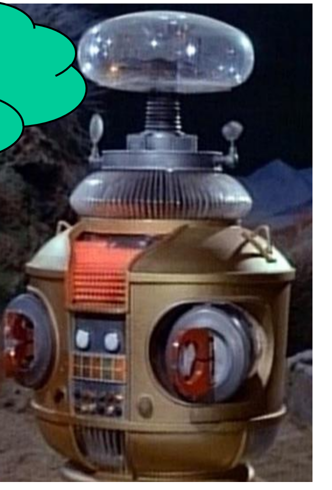
Type Casting

`dynamic_cast<new type>( expression)`

- Special version where type casting is performed at runtime, only works on reference or pointer type variables.
- Usually handled automatically by the compiler where needed, rarely done by the programmer.

Type Casting cont’d

Danger!



```
const_cast<new type>( expression )
```

- Variables labeled as *const* can’t have their value changed.
- const_cast lets the programmer remove or add *const* to reference or pointer type variables.
- If you need to do this, you probably want to re-think your code.

“unsafe”: the compiler will not protect you here!

The programmer must make sure everything is correct!

Type Casting cont'd

```
reinterpret_cast<new type>( expression )
```

- Takes the bits in the expression and re-uses them **unconverted** as a new type.
- Also only works on reference or pointer type variables.
- Sometimes useful when reading in binary files and extracting parameters.

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First C++ program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

Functions

- 4 function calls are listed.
- The 1st and 2nd functions are identical in their behavior.
  - The values of L and W are sent to the function, multiplied, and the product is returned.
- RectangleArea2 uses *const* arguments
  - The compiler **will not** let you modify their values in the function.
  - Try it! Uncomment the line and see what happens when you recompile.
- The 3rd and 4th versions pass the arguments by *reference* with an added &

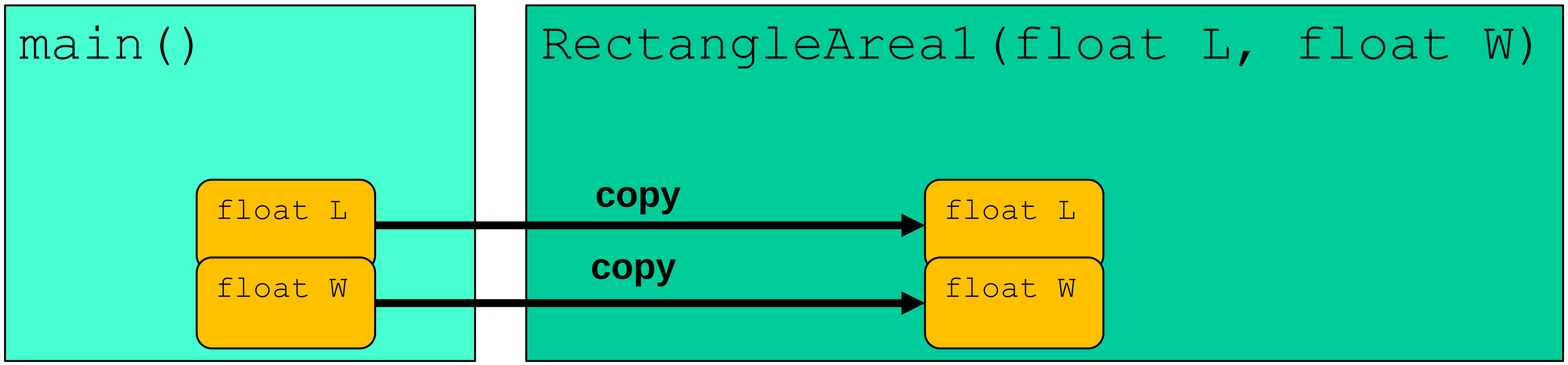
The return type is *float*.

The function arguments L and W are sent as type *float*.

Product is computed

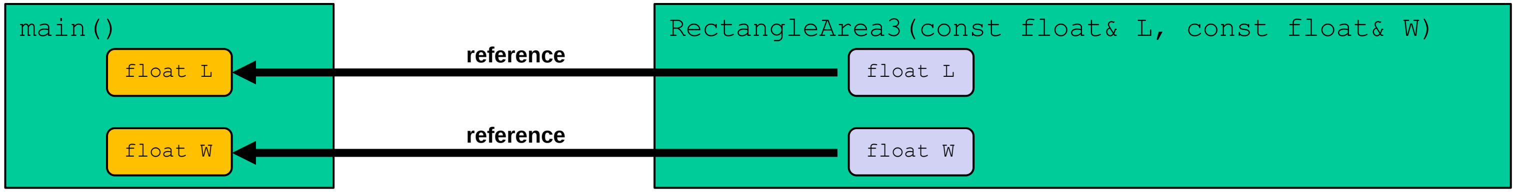
```
float RectangleArea1(float L, float W) {
    return L*W ;
}

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.
- This is the type of behavior you see in Fortran, Matlab, Python, and others.
- 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().

Pass by Reference

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.

Passing objects to Functions

- 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**!

Passing objects to Functions

- 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.

Function overloading

- Briefly: The same function can be implemented **multiple times** with different arguments.
- This allows for special cases to be handled, or specialized behavior for different types.
- Multiple constructors in a class are an example of function overloading.

```
float sum(float a, float b) {
    return a + b;
}

int sum(int a, int b) {
    return a + b;
}
```

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First C++ program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

A first C++ class

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

```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

class keyword

Name of class

Curly braces at the beginning and end followed by a semi-colon

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

Internal variables are called *members*

public keyword indicates everything following the keyword is accessible by any other code outside of this class.

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

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

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

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

    return 0;
}
```

Accessing methods in the class

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.

```
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.

Now for a “real” class

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
}
```

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.
- The *protected* keyword will be discussed later.
- The *private* keyword declares anything following it (members, methods) to be visible only to code **in this class**.

```
#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
```

rectangle.cpp

- 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()
{

}
```

Last Step

1. Go to the main.cpp file
2. Add an include statement for “rectangle.h”
3. Create a Rectangle object in `main()`
4. Add a length and width
5. Print out the area using *cout*.
   - Hint: just like the `BasicRectangle` example…

Solution

- You should have come up with something like this:

```
#include <iostream>

using namespace std;

#include "rectangle.h"

int main()
{
    Rectangle rT ;
    rT.m_width = 1.0 ;
    rT.m_length = 2.0 ;

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

    return 0;
}
```

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First C++ program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

References and Pointers

- Part 1 introduced the concept of passing by reference when calling functions.
  - Selected by using the & character in function argument types:

```
int add (int &a, int b)
```

- References hold a memory address of a value.

  `int add (int &a, int b)` → **`a`** has the value of a memory address, **`b`** has an integer value.

- Used like regular variables and C++ automatically fills in the value of the reference when needed:

  `int c = a + b ;` → “retrieve the value of **`a`** and add it to the value of **`b`**”

References and Pointers

- **From C** there is another way to deal with the memory address of a variable: via *pointer* types.

- Similar syntax in functions except that the & is replaced with a *:

```
int add (int *a, int b)
```

- To get a value from a pointer requires a manual *dereferencing* by the programmer:

`int c = *a + b ;` → "retrieve the value of **a** and add it to the value of **b**"

References and Pointers

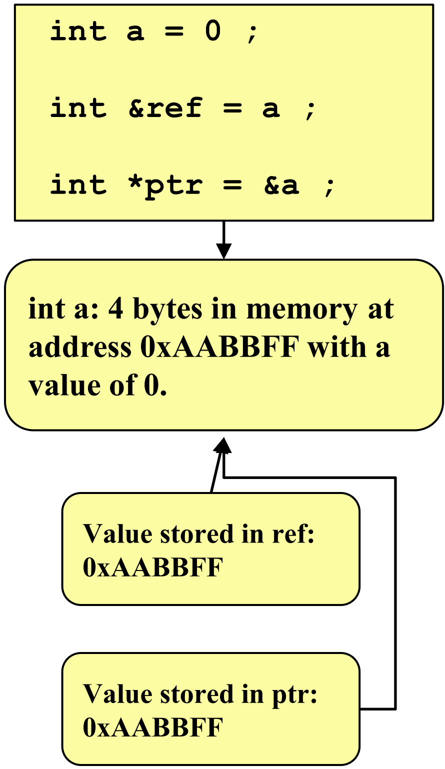
Item	Reference	Pointer
Declaration	int &ref ;	int *ptr ;
Set memory address to something in memory	int a = 0 ; int &ref = a ;	int a = 0 ; int *ptr = &a ;
Fetch value of thing in memory	cout << ref ;	cout << *ptr ;
Can refer/point to nothing (null value)?	No	Yes
Can change address that it refers to/points at?	No. int a = 0 ; int b = 1 ; int &ref = a ; ref = b ; // value of a is now 1!	Yes int a = 0 ; int b = 1 ; int *ptr = &a ; ptr = &b ; // ptr now points at b

Object member/method syntax	MyClass obj ;
	MyClass &ref = obj ;
	
	ref.member ;
	ref.method();
		MyClass obj ;
		MyClass *ptr = obj ;
		ptr->member ;
		ptr->method();
		// OR
		(*ptr).member ;
		(*ptr).method() ;

References and Pointers

Item	Reference	Pointer
Declaration	int &ref ;	int *ptr ;
Set memory address to something in memory	int a = 0 ; int &ref = a ;	int a = 0 ; int *ptr = &a ;
Fetch value of thing in memory	cout << ref ;	cout << *ptr ;
Can refer/point to nothing (null value)?	No	Yes
Can change address that it refers to/points at?	No. int a = 0 ; int b = 1 ; int &ref = a ; ref = b ; // value of a is now 1!	Yes int a = 0 ; int b = 1 ; int *ptr = &a ; ptr = &b ; // ptr now points at b

Object member/method syntax	MyClass obj ;
	MyClass &ref = obj ;
	
	ref.member ;
	ref.method();
		MyClass obj ;
		MyClass *ptr = obj ;
		ptr->member ;
		ptr->method();
		// OR
		(*ptr).member ;
		(*ptr).method() ;



When to use a reference or a pointer

- Both **references** and **pointers** can be used to refer to objects in memory in methods, functions, loops, *etc*.
- Avoids copying due to default **call-by-value C++** behavior
  - Could lead to memory/performance problems.
  - Or cause issues with open files, databases, etc.
- If you need to:
  - Hold a null value (*i.e.,* point at nothing), use a **pointer.**
  - Re-assign the memory address stored, use a **pointer.**
- Otherwise, use a reference.
  - **References are much easier to use!**
  - No need to check if a reference has a null value … since they can't hold one.

When to use a reference or a pointer

- Both references and pointers can be used to refer to objects in memory in methods, functions, loops, etc.

- Avoids copying due to default call-by-value C++ behavior
  - Could lead to memory/performance problems.
  - Or cause issues with open files, databases, etc.

```
//  Pointer to a null value
int *a = NULL ; // C-style
int *b = nullptr ; // C++11 style.


// Reference to a null value
// won't compile.
int &c = nullptr ;
```

Read more: https://www.geeksforgeeks.org/pointers-vs-references-cpp/

Null Value Checking

```
//  Pointer version
void add(const int *a, const int b, int *c)
{
   if (a && c) { // check for null pointer
     *c = *a + b ;
   }
}

// a && c  → this means if a AND c are not
//            null
```

```
//  Reference version
void add(const int &a, const
int b, int &c)
{
   c = a + b ;
}
```

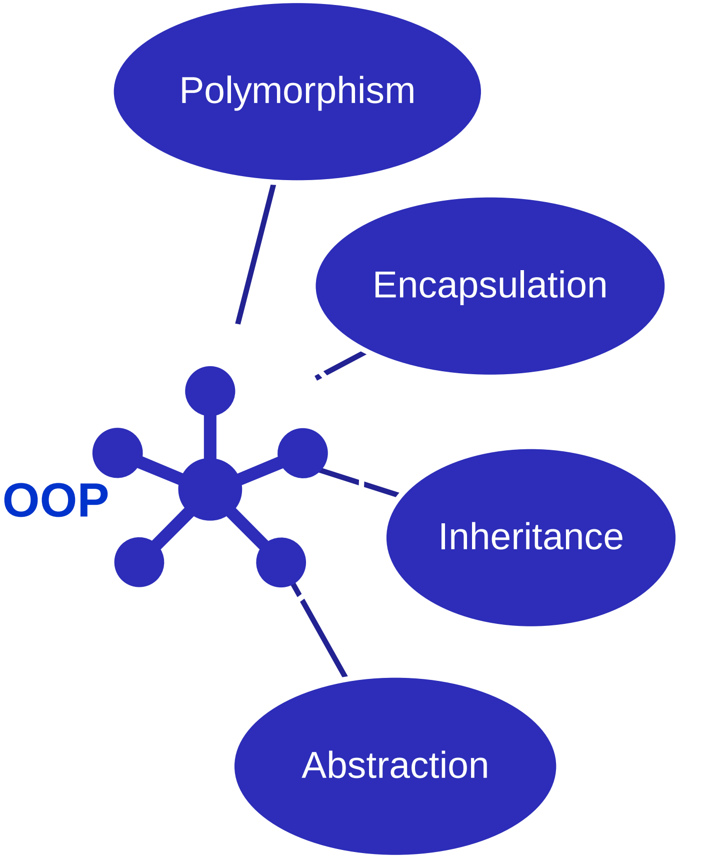
- A **null** value means the pointer is not currently pointing at anything.
  - It's a good idea to check before accessing the value they point at.
- References cannot be null, so the code on the right does not need checking.

Outline

- Very brief history of C++
- Definition of object-oriented programming
- When C++ is a good choice
- First C++ program!
- Some C++ syntax
- Function calls
- Create a C++ class
- References and Pointers
- More on object-oriented programming

The formal concepts in OOP

- Object-oriented programming (OOP):
  - Defines *classes* to represent data and logic in a program. Classes can contain members (data) and methods (internal functions).
  - Creates *instances* of classes, aka *objects*, and builds the programs out of their interactions.
- The core concepts in addition to classes and objects are:
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction



OOP Core Concepts

Encapsulation

- As mentioned while building the C++ class in the last session.
- Bundles related data and functions into a class

Inheritance

- Builds a relationship between classes to share class members and methods

Abstraction

- The hiding of members, methods, and implementation details inside of a class.

Polymorphism

- The application of the same code to multiple data types
- There are 3 kinds, all of which are supported in C++.
- However only 1 is actually called polymorphism in C++ jargon (!)

C++ Classes

- In the Rectangle class, IDE generated two methods automatically.

- *Rectangle()* is a *constructor.* This is a method that is called when an object is instantiated for this class.
  - Multiple constructors per class are allowed

- ~Rectangle() is a *destructor*. This is called when an object is removed from memory.
  - Only **one** destructor per class is allowed!
  - (ignore the *virtual* keyword for now)

```
#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
```

Encapsulation

- Bundling the data and area calculation for a rectangle into a single class is and example of the concept of *encapsulation.*

```
#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
```

Construction and Destruction

- The ***constructor*** is called when an object is created.

- This is used to initialize an object:
  - Load values into member variables
  - Open files
  - Connect to hardware, databases, networks, etc.

Construction and Destruction

- The ***constructor*** is called when an object is created.
- This is used to initialize an object:
  - Load values into member variables
  - Open files
  - Connect to hardware, databases, networks, etc.

- The ***destructor*** is called when an object goes *out of scope*.
- Example:
  - Object c1 is created when the program reaches the first line of the function, and destroyed when the program leaves the function.

```
void function() {
        ClassOne c1 ;
}
```

When an object is instantiated…

- The **rT** object is created in memory.
- When it is created its *constructor* is called to do any necessary initialization.
  - Here the constructor is empty so nothing is done.
- The constructor can take any number of arguments like any other function but it *cannot* return any values.
  - Essentially the return value is the object itself!
- What if there are multiple constructors?
  - The compiler chooses the correct one based on the arguments given.

```
#include "rectangle.h"

int main()
{
    Rectangle rT ;
    rT.m_width = 1.0 ;
}
```

```
#include "rectangle.h"

Rectangle::Rectangle()
{
    //ctor
}
```

Note the constructor has no return type!

A second constructor

- Two styles of constructor. The right code is the **C++11** *member initialization list* style. At the left is the old way. C++11 is preferred.
- With the old way *the empty constructor is called automatically* even though it does nothing – it still adds a function call.
- Same `rectangle.h` for both styles.

rectangle.cpp

```
#include "rectangle.h"
/* OK to do this */

Rectangle::Rectangle(float width, float length){
    m_width = width ;
    m_length = length ;
}
```

OR

rectangle.cpp

```
#include "rectangle.h"

/* Better to do this */
Rectangle::Rectangle(float width, float length):
    m_width(width),m_length(length) {

}
```

rectangle.h

```
class Rectangle
{
    public:
        Rectangle();
        Rectangle(float width, float length) ;
    /* etc */
};
```

Member Initialization Lists

Syntax:

Colon goes here

Members assigned and separated with commas. Note: order doesn't matter.

```
MyClass(int A, OtherClass &B, float C):
          m_A(A),
          m_B(B),
          m_C(C) {
                    /* other code can go
here */
          }
```

Additional code can be added in the code block.

and now use both constructors

- Both constructors are now used.
- The new constructor initializes the values when the object is created.
- Constructors are used to:
  - Initialize members
  - Open files
  - Connect to databases
  - Etc.

```
#include <iostream>

using namespace std;

#include "rectangle.h"

int main(){

    Rectangle rT ;
    rT.m_width = 1.0 ;
    rT.m_length = 2.0 ;

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

    Rectangle rT_2(2.0,2.0) ;
    cout << rT_2.Area() << endl ;

    return 0;
}
```

Default values

- C++11 added the ability to define default values in headers in an intuitive way.
- Pre-C++11 default values would have been coded into constructors.
- If members with default values get their value set in constructor than the default value is ignored.
  - *i.e.*, no “double setting” of the value.

```
#ifndef RECTANGLE_H
#define RECTANGLE_H


class Rectangle
{
    public:
        Rectangle();
        virtual ~Rectangle();
        // could do:
        float m_length = 0.0 ;
        float m_width = 0.0 ;

        float Area() ;
    protected:

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

Default constructors and destructors

- The two methods created by IDE automatically are explicit versions of the default C++ constructors and destructors.
- Every class has them – if you don't define them then empty ones that do nothing will be created for you by the compiler.
    - If you really don't want the default constructor you can delete it with the *delete* keyword.
    - Also in the header file you can use the *default* keyword if you like to be clear that you are using the default.

```
class Foo {
    public:
        Foo() = delete ;
        // Another constructor
        // must be defined!
        Foo(int x) ;
};

class Bar {
    public:
        Bar() = default ;
};
```

Custom constructors and destructors

- You must define **your own constructor** when you want to initialize an object with arguments.
- A custom destructor is **always** needed when internal members in the class need special handling.
  - Examples: manually allocated memory, open files, hardware drivers, database or network connections, custom data structures, etc.

Destructors

- Destructors are called when an object is destroyed.
- Destructors have no return type.
- There is only **one** destructor allowed per class.
- Objects are destroyed when they go out of *scope*.
- Destructors are never called explicitly by the programmer. Calls to destructors are inserted automatically by the compiler.

This class just has 2 floats as members which are automatically removed from memory by the compiler.

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

~House() destructor

House object

Destructors

- Example:

```
class Example {
  public:
        Example() = delete ;
        Example(int count) ;

        virtual ~Example() ;

        // A pointer to some
memory
       // that will be allocated.
        float *values = nullptr ;
};
```

```
Example::Example(int count) {
        // Allocate memory to store
"count"
        // floats.
        values = new float[count];
}

Example::~Example() {
        // The destructor must free this
        // memory. Only do so if values
is not
       // null.
        if (values) {
                delete[] values ;
        }
}
```

Scope

- Scope is the region where a variable is valid.
- Constructors are called when an object is created.
- Destructors are only ever called implicitly.

```
int main() { // Start of a code block
        // in main function scope
        float x ;  // No constructors for built-in types
        ClassOne c1 ;   // c1 constructor ClassOne() is called.
        if (1){  // Start of an inner code block
            // scope of c2 is this inner code block
                ClassOne c2 ; //c2 constructor ClassOne() is called.
        }   // c2 destructor ~ClassOne() is called.
        ClassOne c3 ;  // c3 constructor ClassOne() is called.
}   // leaving program, call destructors for c3 and c1 ~ClassOne()
    // variable x: no destructor for built-in type
```

Reference

- Reading Assignment: Chapters 1 and 2 of "C++ How to Program"
- See also:
  - https://cplusplus.com/
  - https://www.w3schools.com/cpp/