Introduction to C++

Output

Printing in C++ is done through overloading the **<<** operator:

```
cout << 3;
```

If the left-hand argument of **<<** is an object of type **ostream** (*o*utput *stream*) and the right-hand argument is a **double**, **int**, **string**, etc., an appropriate function which prints the object is called

Output

The format is suggestive of what is happening:

- The objects are being *sent* to the **cout** (*console output*) object to be printed

```
cout << "The square of 3 is " << sqr(3) << endl;
```

The objects being printed are:

- a **string**
- an **int**
- a platform-independent end-of-line identifier

Output

How does

```
cout << "The square of 3 is " << sqr(3) << endl;
```

work?

This is equivalent to

```
((cout << "The square of 3 is ") << sqr(3)) << endl;
```

where **<<** is an operation (like **+**) which prints the object and returns the **cout** object

Output

```
((cout << "The square of 3 is ") << sqr(3)) << endl;
```

print `"The square of 3 is "` and return `cout`

```
(           cout                  << sqr(3)) << endl;
```

print the result of `sqr(3)` and return `cout`

```
                 cout                         << endl;
```

print an end-of-line character (and return `cout`)

```
cout;
```

Output

Another way to look at this is that

```
cout << "The square of 3 is " << sqr(3) << endl;
```

is the same as:

```
operator<<( operator<<( operator<<( cout, "The square of 3 is " ), sqr(3) ), endl );
```

This is how C++ treats these anyway…

Input

```
#include<iostream>

using namespace std;

int main(){
    int x;
    cin>>x;
    cout<<x<<endl;
    return 0;
}
```

```
3
3
```

```
#include<iostream>

using namespace std;

int main(){
    int x;
    int y;
    cin>>x>>y;
    cout<<x<<" "<<y<<endl;
}
```

```
3
4
3 4
```

Operators

Operators have similar functionality for built-in datatypes:

- Assignment =
- Arithmetic + - * / %
  += -= *= /= %=
- Autoincrement ++
- Autodecrement --
- Logical && || !
- Relational == != < <= >= >
- Comments /* */
  // to end of line
- Bitwise & | ^ ~
  &= |= ^=
- Bit shifting << >>
  <<= >>=

Control Statements

All control statements are similar

```
if ( statement ) {
    // ...
} else if ( statement ) {
    // ... } else
    // ...
}
```

```
for ( int i = 0; i < N; ++i ) {
        //...
}

do {
        // ...
} while ( statement );

while ( statement ) {
{
        // ...
}
```

Arrays

Accessing arrays is similar:

```
const int N = 10; // prevents reassignment
int arr[N];
arr[0] = 1;
for(int i=1; i<N; i++){
    arr[i] = 2*arr[i-1]+1;
}
for(int i=0;i<N;i++){
    cout<<arr[i]<<" ";
}
```

```
1 3 7 15 31 63 127 255 511 1023
```

Definition:

The *capacity* of an array is the entries it can hold

The *size* of an array is the number of useful entries

Function

Function calls are similar, however, the are not required to be part of a class:

```
#include <iostream>
using namespace std;
int sqr(int n) {
    return n*n;
}
int main() {
    cout << "The square of 3 is " << sqr(3) << endl;
    return 0;
}
```

Recursion

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

int func(int x){
    if(x<1)
            return 0;
    else
            return x+func(x-1);
}
int main() {
    cout<<func(5)<<endl;
    return 0;
}
```

15

Recursion

```
int func(int x){
        if(x<1)
                return 0;
        else
                return x+func(x-1);
}
```

func(5) = 5+func(4)

func(4) = 4+func(3)

func(3) = 3+func(2)

func(2) = 2+func(1)

func(1) = 1+ func(0)

func(0) = 0

15

func(5) = 5+func(4) = 5+10 = 15

func(4) = 4+func(3) = 4+6 = 10

func(3) = 3+func(2) = 3+3 = 6

func(2) = 2+func(1) = 2+1 = 3

func(1) = 1+ func(0) = 1+0 = 1

func(0) = 0

The C++ Preprocessor

C++ is based on C, which was written in the early 1970s

Any command starting with a **#** in the first column is not a C/C++ statement, but rather a preprocessor statement

- The preprocessor performed very basic text-based (or *lexical*) substitutions
- The output is sent to the compiler

The C++ Preprocessor

The sequence is:

file (filename.cpp) ⟶ preprocessor ⟶ compiler (g++)

Note, this is done automatically by the compiler: no additional steps are necessary

At the top of any C++ program, you will see one or more directives starting with a **#**, *e.g.*,

#include <iostream>

The C++ Preprocessor

iostream.h

intro.cpp

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

int sqr( int n ) {
    return n*n;
}

int main() {
    cout << "The square of 3 is " << sqr(3) << endl;
    return 0;
}
```

preprocessor

to the compiler

Libraries

You will write your code in a file such as Single_list.h where you will implement a data structure

This file will be included in our tester file Single_list_tester.h with a statement such as:

```
#include "Single_list.h"
```

The file Single_list_int_driver.cpp then includes the tester file:

```
#include "Single_list_tester.h"
```

Libraries

You will note the difference:

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

The first looks for a file iostream.h which is shipped with the compiler (the standard library)

The second looks in the current directory

Libraries

In this class, you will put all code in the header file

This is not normal practice:

- Usually the header (.h) file only contains declarations
- The definitions (the actual implementations) are stored in a related file and compiled into an object file

The C++ Preprocessor

mylib.h

```
#ifndef _MYLIB_H
#define _MYLIB_H

int sqr( int n );

int cube( int n );

int sumi( int n );

int sumi2( int n );

int factorial( int n );

#endif
```

included in source file

only the source code is compiled

mylib.cpp

```
#include "mylib.h"

int sqr( int n ) {
    return n*n;
}

int cube( int n ) {
    return n*n*n;
}

int sumi( int n ) {
    return (n*(n + 1))/2;
}

int sumi2( int n ) {
    return (n*(n + 1)*(2*n + 1))/6;
}

int factorial( int n ) {
    if ( n <= 1 ) {
        return 1;
    } else {
        return n*factorial(n - 1);
    }
}
```

compiled into a library

mylib.o
mylib.so

linked with library to generate the executable file

mylib.h

```
#ifndef _MYLIB_H
#define _MYLIB_H

int sqr( int n ) {
    return n*n;
}

int cube( int n ) {
    return n*n*n;
}

int sumi( int n ) {
    return (n*(n + 1))/2;
}

int sumi2( int n ) {
    return (n*(n + 1)*(2*n + 1))/6;
}

int factorial( int n ) {
    if ( n <= 1 ) {
        return 1;
    } else {
        return n*factorial(n - 1);
    }
}

#endif
```

included in source file

everything is compiled

executable is generated (no linking necessary)

The C++ Preprocessor

With all these includes, it is always necessary to avoid the same file being included twice, otherwise you have duplicate definitions

This is done with guard statements:

```
#ifndef SINGLE_LIST_H
#define SINGLE_LIST_H

template <typename Type>
class Single_list {
   ///...
};

#endif
```

The File as the Unit of Compilation

In C/C++, the file is the base unit of compilation:

- Any .cpp file may be compiled into object code
- Only files containing an int main() function can be compiled into an executable

The signature of main is:

```
int main () {
    // does some stuff
    return 0;
}
```

The operating system is expecting a return value

- Usually 0

The File as the Unit of Compilation

This file (**example.cpp**) contains two functions

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

int sqr( int n ) {      // Function declaration and definition
    return n*n;
}
int main() {
    cout << "The square of 3 is " << sqr(3) << endl;
    return 0;
}
```

The File as the Unit of Compilation

This is an alternate form:

```
#include<iostream>
using namespace std;
int sqr( int );                    // Function declaration
int main() {
    cout << "The square of 3 is " << sqr(3) << endl;
    return 0;
}
int sqr( int n ) {                 // Function definition
    return n*n;                    // The definition can be in another file
}
```

Namespaces

Variables defined:

- In functions are *local variables*
- In classes are *member variables*
- Elsewhere are *global variables*

Functions defined:

- In classes are *member functions*
- Elsewhere are *global functions*

In all these cases, the keyword **static** can modify the scope

Namespaces

A namespace adds an extra disambiguation between similar names

```
namespace ca_uwaterloo_dwharder {
    int n = 4;
    double mean = 2.34567;

    void init() {
        // Does something...
    }
}
```

There are two means of accessing these global variables and functions outside of this namespace:

- The namespace as a prefix:
  - `ca_uwaterloo_dwharder::init()`
- The using statement:
  - `using namespace ca_uwaterloo_dwharder;`

Namespaces

You will only need this for the standard name space

- All variables and functions in the standard library are in the **std** namespace

```
#include <iostream>
std::cout << "Hello world!" << std::endl;
```

```
#include <iostream>
using namespace std;          // never used in production code

cout << "Hello world!" << endl;
```

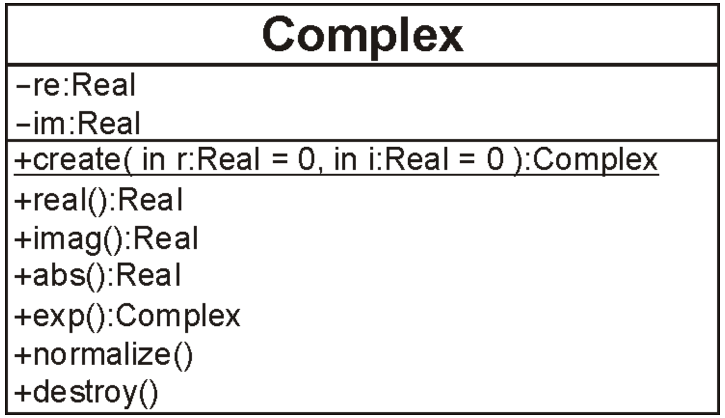
Classes

To begin, we will create a complex number class

To describe this class, we could use the following words:

- Store the real and imaginary components
- Allow the user to:
  - Create a complex number
  - Retrieve the real and imaginary parts
  - Find the absolute value and the exponential value
  - Normalize a non-zero complex number

Classes



An example of a C++ class declaration is:

```
class Complex {
   private:
      double re, im;
   public:
      Complex( double = 0.0, double = 0.0 );
      double real() const;
      double imag() const;
      double abs() const;
      Complex exp() const;
      void normalize();
};
```

Complex Class

```
#ifndef _COMPLEX_H
#define _COMPLEX_H
#include <cmath>
class Complex {
    private:
        double re, im;
    public:
        Complex( double = 0.0, double = 0.0 );
        // Accessors
        double real() const;
        double imag() const;
        double abs() const;
        Complex exp() const;
        // Mutators
        void normalize();
};
```

Complex Class

Associates functions back to the class

```
// Constructor
Complex::Complex( double r, double i ):
re( r ),
im( i ) {
    // empty constructor
}
```

Each member variable should be assigned

The order must be the same as the order in which the member variables are defined in the class

For built-in datatypes, this is a simple assignment. For member variables that are objects, this is a call to a constructor.

For built-in datatypes, the above is equivalent to:

```
// Constructor
Complex::Complex( double r, double i ):re( 0 ), im( 0 ) {
    re = r;
    im = i;
}
```

Complex Class

```
// return the real component
double Complex::real() const {
    return re;
}

// return the imaginary component
double Complex::imag() const {
    return im;
}

// return the absolute value
double Complex::abs() const {
    return std::sqrt( re*re + im*im );
}
```

Refers to the member variables re and im of this class

Complex Class

```
// Return the exponential of the complex value
Complex Complex::exp() const {
    double exp_re = std::exp( re );

    return Complex( exp_re*std::cos(im),
exp_re*std::sin(im) );
}
```

Complex Class

```
// Normalize the complex number (giving it unit absolute value, |z| = 1)
void Complex::normalize() {
    if ( re == 0 && im == 0 ) {
        return;
    }

    double absval = abs();
    re /= absval;
    im /= absval;
}

#endif
```

This calls the member function `double abs() const` from the `Complex` class on the object on which `void normalize()` was called

Complex Class

Visibility in C# and Java is described by placing public/private/protected in front of each class member or member function

In C++, this is described by a block prefixed by one of

private:

protected:

public:

Complex Class

```
class Complex {
    private:
        double re, im;
    public:
        Complex( double, double );
        double real() const;
        double imag() const;
        double abs() const;
        Complex exp() const;
        void normalize();
};
```

Visibility

The reason for the change in Java/C# was that the C++ version has been noted to be a source of errors

Code could be cut-and-paste from one location to another, and a poorly placed paste could change the visibility of some code:

public $\longrightarrow$ private automatically caught

private $\longrightarrow$ public difficult to catch and dangerous

Visibility

It is possible for a class to indicate that another class is allowed to access its **private** members

If class ClassX declares class ClassY to be a friend, then class ClassY can access (and modify) the private members of ClassX

Visibility

```
class ClassY;                    // declare that ClassY is a class
class ClassX {
    private:
        int privy;               // the variable privy is private

    friend class ClassY;         // ClassY is a "friend" of ClassX
};
class ClassY {                   // define ClassY
    private:
        ClassX value;            // Y stores one instance of X
    public:
        void set_x() {
            value.privy = 42;    // a member function of ClassY can
        }                        // access and modify the private
};                               // member privy of "value"
```

Accessors and Mutators

We can classify member functions into two categories:

- Those leaving the object unchanged
- Those modifying the member variables of the object

Respectively, these are referred to as:

- **Accessors:** we are accessing and using the class members
- **Mutators:** we are changing—mutating—the class members

Accessors and Mutators

Good programming practice is to enforce that a routine specified to be an accessor cannot be accidentally changed to a mutator

This is done with the const keyword after the parameter list

```
double abs() const;
```

Accessors and Mutators

If a junior programmer were to try change

```
double Complex::abs() const {
   return std::sqrt( re*re + im*im );
}
```

to

```
double Complex::abs() const {
    re = 1.0;            // modifying (mutating) 're'
    return std::sqrt( re*re + im*im );
}
```

the compiler would signal an error

Accessors and Mutators

As an example from a previous project, a student did this:

```
template <typename Type>
int Double_sentinel_list<Type>::count( Type const &obj ) const {
    for ( Double_node<Type> *temp = head(); temp != nullptr; temp = temp->next() ) {
        if ( temp->retrieve() == obj ) {
            ++list_size;
        }
    }
    return list_size;
}
```

Here, list_size was a member variable of the class

This code did not compile: the compiler issued a warning that a member variable was being modified in a read-only member function

Accessors and Mutators

What the student wanted was a local variable:

```
template <typename Type>
int Double_sentinel_list<Type>::count( Type const &obj ) const {
    int obj_count = 0;

    for ( Double_node<Type> *temp = head(); temp != nullptr; temp = temp->next() ) {
        if ( temp->retrieve() == obj ) {
            ++obj_count;
        }
    }
    return obj_count;
}
```

Templates

Suppose you want to build a general linked list which could hold anything

- In C#, you could have it store instance of the class Object

Because there is no ultimate Object class, to avoid re-implementing each class for each class we are interested in storing, we must have a different mechanism

Templates

This mechanism uses a tool called templates

- A function has parameters which are of a specific type
- A template is like a function, however, the parameters themselves are types

Templates

That mechanism is called a template:

```
template <typename Type>
Type sqr( Type x ) {
    return x*x;
}
```

This creates a function which returns something of the same type as the argument

Templates

To tell the compiler what that type is, we must suffix the function:

```
int n = sqr<int>( 3 );

double x = sqr<double>( 3.141592653589793 );
```

Usually, the compiler can determine the appropriate template without it being explicitly stated

Templates

Example:

```
#include<iostream>
using namespace std;
template <typename Type>
Type sqr( Type x ) {
    return x*x;
}
int main() {
    cout << "3 squared is " << sqr<int>( 3 ) << endl;
    cout << "Pi squared is " << sqr<double>( 3.141592653589793 ) << endl;
    return 0;
}
```

Output:

```
3 squared is 9
Pi squared is 9.8696
```

Templates

```
#ifndef _COMPLEX_H
#define _COMPLEX_H
#include <cmath>
template <typename Type>
class Complex {
    private:
        Type re, im;
    public:
        Complex( Type const & = Type(), Type const & = Type() );
        // Accessors
        Type real() const;
        Type imag() const;
        Type abs() const;
        Complex exp() const;
        // Mutators
        void normalize();
};
```

Templates

The modifier template <typename Type> applies only to the following statement, so each time we define a function, we must restate that Type is a templated symbol:

```
// Constructor
template <typename Type>
Complex<Type>::Complex( Type const &r, Type const &i ):re(r), im(i) {
    // empty constructor
}
```

Templates

```
// return the real component
template <typename Type>
Type Complex<Type>::real() const {
    return re;
}
// return the imaginary component
template <typename Type>
Type Complex<Type>::imag() const {
    return im;
}
```

```
// return the absolute value
template <typename Type>
Type Complex<Type>::abs() const {
    return std::sqrt( re*re + im*im );
}
```

Templates

```
// Return the exponential of the complex value
template <typename Type>
Complex<Type> Complex<Type>::exp() const {
    Type exp_re = std::exp( re );
    return Complex<Type>( exp_re*std::cos(im), exp_re*std::sin(im) );
}
// Normalize the complex number (giving it unit norm, |z| = 1)
template <typename Type>
void Complex<Type>:noramlize() {
    if ( re == 0 && im == 0 ) {
        return;
    }
    Type absval = abs();
    re /= absval;
    im /= absval;
}
#endif
```

Templates

Example:

```
#include <iostream>
#include "Complex.h"
using namespace std;
int main() {
    Complex<double> z( 3.7, 4.2 );
    Complex<float>  w( 3.7, 4.2 );
    cout.precision( 20 ); // Print up to 20 digits
    cout << "|z| = " << z.abs() << endl;
    cout << "|w| = " << w.abs() << endl;
    z.normalize();
    w.normalize();
    cout << "After normalization, |z| = " << z.abs() << endl;
    cout << "After normalization, |w| = " << w.abs() << endl;
    return 0;
}
```

Output:

```
|z| = 5.5973207876626123181
|w| = 5.597320556640625
After normalization, |z| =
1.0000000412736744781
After normalization, |w| = 1
```

Pointers

We could simply have an ‘address’ type:

```
address ptr;    // store an address
                // THIS IS WRONG
```

however, the compiler does not know what it is an address of (is it the address of an int, a double, etc.)

Instead, we have to indicate what it is pointing to:

```
int *ptr;   // a pointer to an integer
            // the address of the integer variable 'ptr'
```

Pointers

First we must get the address of a variable

This is done with the **&** operator

(*a*mpersand/*a*ddress of)

For example,

```
int m = 5;    // m is an int storing 5
int *ptr;     // a pointer to an int
ptr = &m;     // assign to ptr the
              // address of m
```

Pointers

We can even print the addresses:

```
int m = 5;    // m is an int storing 5
int *ptr;     // a pointer to an int
ptr = &m;     // assign to ptr the
              // address of m
cout << ptr << endl;
```

prints 0xffffd352, a 32-bit number

- The computer uses 32-bit addresses

Pointers

We have pointers: we would now like to manipulate what is stored at that address

We can access/modify what is stored at that memory location by using the * operator (dereference)

```
int m = 5;
int *ptr;
ptr = &m;
cout << *ptr << endl;  // prints 5
```

Pointers

Similarly, we can modify values stored at an address:

```
int m = 5;
int *ptr;
ptr = &m;

*ptr = 3;    // store 3 at that memory location
cout << m << endl;  // prints 3
```

Pointers

Pointers to objects must, similarly be dereferenced:

```
Complex z( 3, 4 );
Complex *pz;
pz = &z;
cout << z.abs() << endl;
cout << (*pz).abs() << endl;
```

Pointers

One short hand for this is to replace

```
(*pz).abs();
```

with

```
pz->abs();
```

Memory Allocation

Memory allocation in C++ and Java is done through the **new** operator

This is an explicit request to the operating system for memory

- This is a very expensive operation
- The OS must:
  - Find the appropriate amount of memory,
  - Indicate that it has been allocated, and
  - Return the address of the first memory location

Memory Allocation

Inside a function, memory allocation of declared variables is dealt with by the compiler

```
int my_func() {
    Complex<double> z(3, 4);  // calls constructor with 3, 4
                              // creates 3 + 4j
                         // 16 bytes are allocated by the compiler
    double r = z.abs(); // 8 bytes are allocated by the compiler
    return 0;           // The compiler reclaims the 24 bytes
}
```

Memory Allocation

Memory for a single instance of a class (one object) is allocated using the new operator, *e.g.*,

```
Complex<double> *pz = new Complex<double>( 3, 4 );
```

The new operator returns the address of the first byte of the memory allocated

Memory Allocation

We can even print the address to the screen

If we were to execute

```
cout << "The address pz is " << pz << endl;
```

we would see output like:

The address pz is 0x00ef3b40

Memory Allocation

Next, to deallocate the memory (once we're finished with it) we must explicitly tell the operating system using the delete operator:

```
delete pz;
```

Memory Allocation

Consider a linked list where each node is allocated:

new Node<Type>(obj)

Such a call will be made each time a new element is added to the linked list

For each new, there must be a corresponding delete:

- Each removal of an object requires a call to delete
- If a non-empty list is itself being deleted, the destructor must call delete on all remaining nodes

Memory Allocation

Memory deallocation differs, however:

- Java uses automatic garbage collection
- C++ requires the user to explicitly deallocate memory

Note however, that:

- *managed C++* has garbage collection
- other tools are also available for C++ to perform automatic garbage collection

Reference

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/