Week 9 Lecture 1

NWEN 241
Systems Programming

Alvin Valera

Alvin.valera@ecs.vuw.ac.nz

Content

- Introduction to C++
  - Data Types
  - Scope and Namespace
  - Input Output
- C++ Classes

Programming with C++

Introduction to C++

- Developed by Bjarne Stroustrup in 1979.
  - Objective was to develop efficient and flexible language similar to C that also provided high-level features for program organization
  - C++ used to be initially called “C with Classes”
  - The first standard appeared in 1998 - C++98
- Latest standard - C++ 20
  - Over time, both C and C++ evolved and have diverged from one another.
  - Compatibility between the languages is has always been considered important

- Widely use for video games, embedded systems, IoT and resource-heavy VR and AI applications

Introduction to C++

- C++ extends C
- Is C++ a superset of C ?
  - **Not anymore**
- Not **every** C program can run on a C++ complier
- **Most** of the C programs run on a C++ compiler

Introduction to C++

C++ extends C with

- strong type checking
- new data types
- **Classes** for implementing user defined data types
- **Object-oriented programming** (also supports - procedural and functional).
- a **standard library** (STL) for frequently used data types (string, list, stack, queue, vector, hash,…)
- generic programming, i.e., parameterization of variable types via **templates**
- exceptions for error handling

Moving from C to C++

hello.cpp

```
#include<stdio.h>
int main()
{
   printf("Hello World");
}
```

Programs written in C may be valid in C++

Compile: g++ hello.cpp –o helloex

Run: ./helloex

Moving from C to C++

- **Header Files:**
  - In standard C++, library headers are **not** supposed to have an extension.
  - C++ standard neither requires nor forbids an extension for other headers. The common choices of extension of user defined headers are- ***.h / *.hh /* .hpp**

- **C Compatibility Header Files**
  - For some of the C standard library headers of the form `xxx.h`, the C++ standard library both includes an identically-named header and another header of the form `cxxx`.

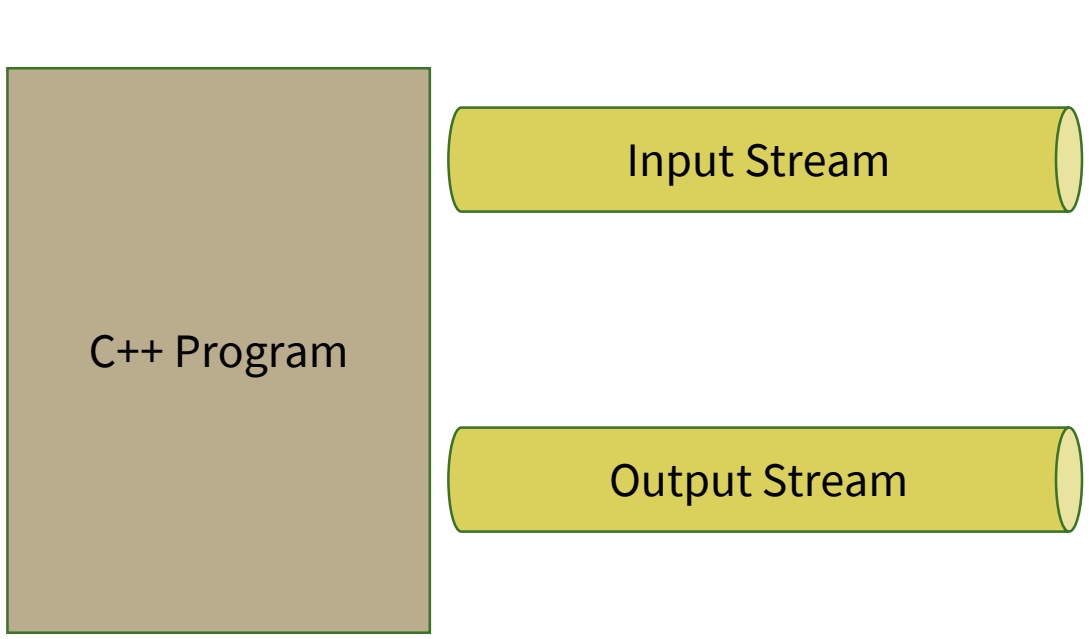
Primitive Data Types

Data Type	Keyword	Modifiers
Character	char	signed, unsigned
Integer	int	signed, unsigned, short, long, long long
Float	float	
Double	double	long
Boolean	bool	
Wide character representations	wchar_t	

I / O in C++

I/O Basics

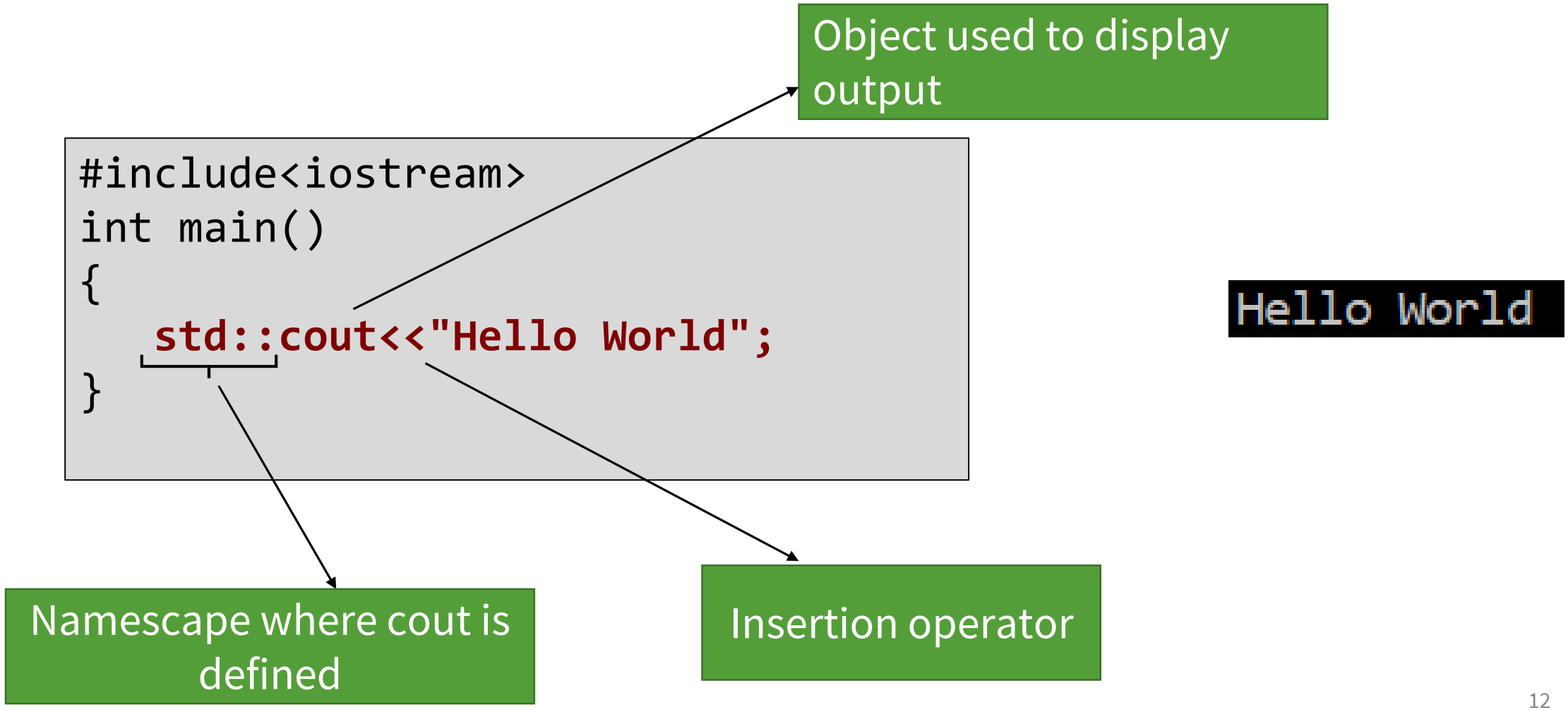
- C/C++ I/O are based on *streams*, which are sequence of bytes flowing in and out of the programs
- Streams acts as an intermediaries between the programs and the actual I/O devices



With **input** streams, the **extraction operator (>>)** is used to remove values from the stream

With **output** streams, the **insertion operator (<<)** is used to put values in the stream

Standard output



Standard Input

```
#include<iostream>
int main()
 {
   int i;
   std::cin>>i;

   std::cout<<"You entered"<<i;
 }
```

Object used to read input

extraction operator

Namespace where cin in defined

```
125
You entered 125
```

Standard I/O in C++

- C++ comes with **four** predefined standard stream objects
- `cerr` and `clog` are output streams that essentially work like `cout`, with the only difference being that they identify streams for specific purposes: error messages and logging.
- Can also be individually redirected to other I/O devices.

Stream	Description	Remarks
cin	Tied to the standard input	Typically tied to the keyboard
cout	Tied to the standard output	Typically tied to the monitor
cerr	Tied to the standard error, providing unbuffered* output	Typically tied to the monitor
clog	Tied to the standard error, providing buffered* output	Typically tied to the monitor

* Unbuffered output is typically handled immediately, where as buffered output is typically stored

Scope and Namespace

Scope

```
int foo;          // global variable

int some_function ()
{
  int bar;        // local variable
  bar = 0;
}

int other_function ()
{
  foo = 1; // ok: foo is a global
variable
  bar = 2; // wrong: bar is not
           // visible here
}
```

When we declare a program element such as a variable, function or a class its name can only be “visible" and used in certain parts of your program.

The context in which a name is visible is called its scope.

An entity declared outside any block has global scope, meaning that its name is valid anywhere in the code.

While an entity declared within a block, such as a function or a selective statement, has block scope, and is only visible within the specific block in which it is declared, but not outside it. Variables with block scope are known as local variables.

Namespace

- If our code base includes multiple libraries there may be name conflicts.
- **Namespaces** are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope.
- General syntax:

```
namespace namespace_name
{
        members
}
```

- Members can be constants, variables, functions, classes, or another namespace (nested namespaces)

Example:

```
namespace myns
{
    const int N = 100;
    int count = 0;
    void printResult();
 }
```

Namespace

- The scope of a namespace member is **local** to that namespace. All identifiers at namespace scope are visible to one another without qualification.
- Members are **not** visible **outside** its namespace.
- Everything not declared in another namespace/scope is in the global (program-wide) namespace.
- Two ways to access a namespace member outside its namespace:
  - Use `namespace_name::identifier` syntax
  - Use the `using` keyword to access specific or all members of a namespace

Example:

```
namespace myns
{
    const int N = 100;
    int count = 0;
    void printResult(){
    cout<<N;
    }
 }
```

Example

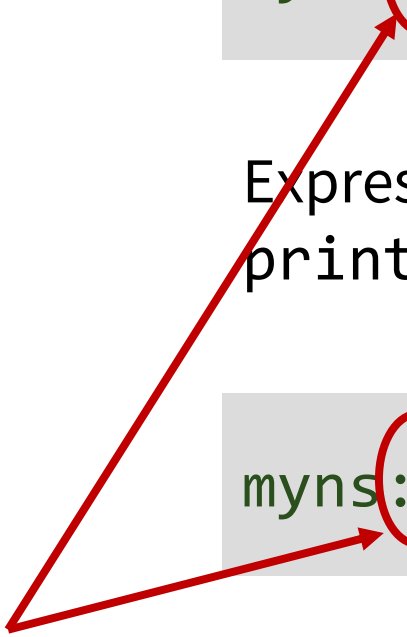
```
namespace myns
{
    const int N = 100;
    int count = 0;
    void printResult(){
    cout<<N;
    }
 }
```

Expression to access N:

```
myns::N
```

Expression to invoke printResult():

```
myns::printResult();
```



Scope resolution operator

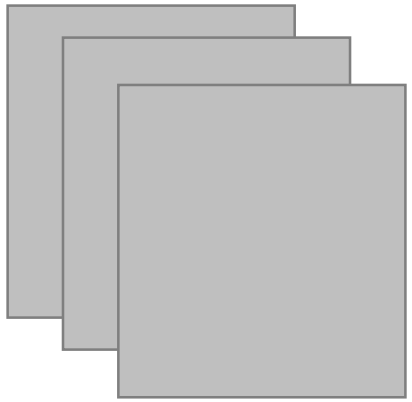
Example

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

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}
```

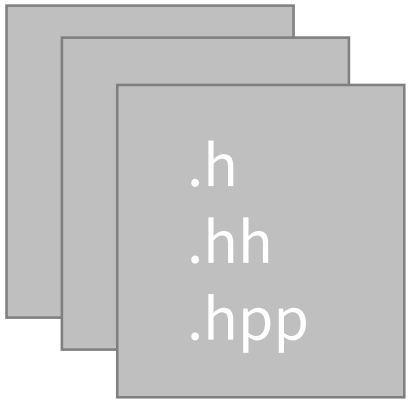
```
int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```

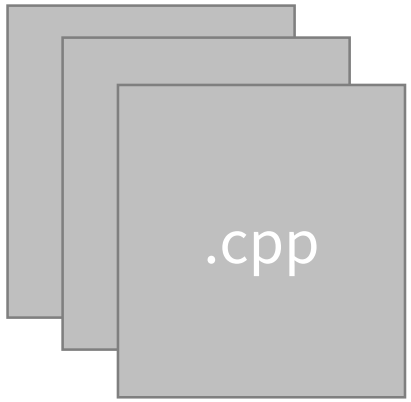
A Large C++ Program



Header files
from standard
C++ libraries



Own header files



Source files

Program Structure

- A typical C++ program consists of
  - 1 or more **header** files
  - 1 or more C++ **source** files

```
#include <iostream>

int main(void)
{
   std::cout << "Hello world\n";

   return 0;
}
```

Preprocessor directive to include `iostream` header file which contains `std::cout`

`main` function *definition*, invoking `std::cout` to display “Hello, world”, and return 0

Another example program (C++)

```
/* Program to calculate the area of a circle */
#include <iostream>
#define PI 3.14

float sq(float);

int main(void)
{
   float radius, area;

   /* Ask user to input */
   std::out << "Radius = ";
   std::in >> radius;

   area = PI * sq(radius);
   std::out << "Area = " << area << "\n";
   return 0;
}

float sq(float r)
{
   return (r * r);
}
```

Preprocessor directives

Function prototype

main function

Function definition

Classes

Classes generalizes user defined data types in an object-oriented sense:

- Classes are types representing groups of similar instances
- Each instance has certain fields that define it (instance variables)
- Instances also have functions that can be applied to them– also known as *methods* in OOP
- Access to parts of the class can be limited

Classes allow the combination of data and operations in a single unit

Defining a Class

- A class is a collection of fixed number of components called **members** of the class

- General syntax for defining a class:

```
class class_identifier {
      class_member_list
};
```

  - `class_member_list` consists of variable declarations and/or methods

Example

```
class Time {
public:
        void set(int, int, int);
        void print() const;
        Time();
        Time(int, int, int);

private:
        int hour;
        int minute;
        int second;
};
```

Member access specifiers

Possible specifiers:

- `private`
- `protected`
- `public`

Member Access Specifier

- **Private members** – can only be accessed by member functions (and friends) and not accessible by descendant classes

- **Public members** – can be accessed outside the class and inherited by descendant classes

- **Protected members** – can only be accessed by member functions (and friends) and inherited by descendant classes

- When member access specifier is not indicated, default access is `private`

Example

```
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

Constructors

- Named after class name
- Similar to Java.

When class performs dynamic memory allocation, **destructor** is also needed

Next Lecture

- More on Classes in C++