Module 7 Guidance Notes

Structs,
File I/O & Recursion

ENGG1340/COMP2113

Computer Programming II/Programming Technologies

Estimated Time of Completion: 3 Hours

Outline

There are altogether 3 parts:

I. (P. 6 – 46) Structs – We've learned about the basic C++ built-in data types. With structs, you can define your own compound data type to facilitate data handling. We will also briefly touch upon C++ class which can be considered as an encapsulation of some data together with the operations allowed on the data.

II. (P. 47 – 78) File I/O – This is for reading and writing of data to a file external to your program which can be stored permanently on a hard drive. You will also learn about string stream as well as some I/O formatting here.

III. (P. 79 – 109) Recursion – Recursion is a very powerful method for solving a problem. If your solution to a problem can be defined in a smaller version (i.e., one that accepts a smaller input) of itself, then likely you can write a recursive function for it. A recursion function is usually simple and can thus enhance readability, but sometimes you will need to take note of its runtime complexity.

Before We Start

- We will deal with C++ only in this module.
- Important: We will be using the C++ 11 standard, so make sure that your compiler option is set appropriately. We suggest to use the following command to compile your C++ program:

```
g++ -pedantic-errors -std=c++11 your_program.cpp
```

The -pedantic-errors flag is to make sure that your code conforms to the ISO C/C++ standard. We will enforce this in your assignment submission too.
For more information about C/C++ standards, you may read https://en.wikipedia.org/wiki/ANSI_C and https://isocpp.org/std/the-standard

How to Use this Guidance Notes

- This guidance notes aim to lead you through the learning of the C/C++ materials. It also defines the scope of this course, i.e., what we expect you should know for the purpose of this course. (and which should not limit what you should know about C/C++ programming.)
- Pages marked with “Reference Only” means that they are not in the scope of assessment for this course.
- The corresponding textbook chapters that we expect you to read will also be given. The textbook may contain more details and information than we have here in this notes, and these extra textbook materials are considered references only.

How to Use this Guidance Notes

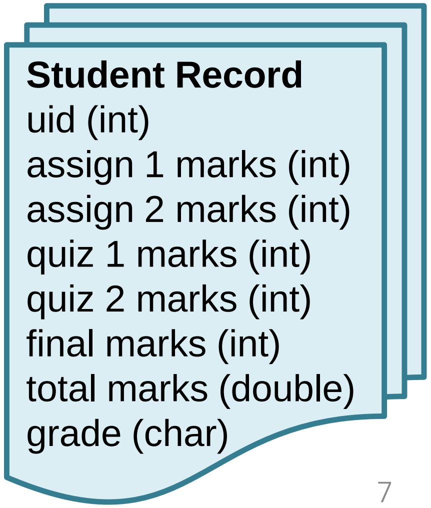
- We suggest you to copy the code segments in this notes to the coding environment and try run the program yourself.
- Also, try make change to the code, then observe the output and deduce the behavior of the code. This way of playing around with the code can help give you a better understanding of the programming language.

Part I

STRUCTS

Structures

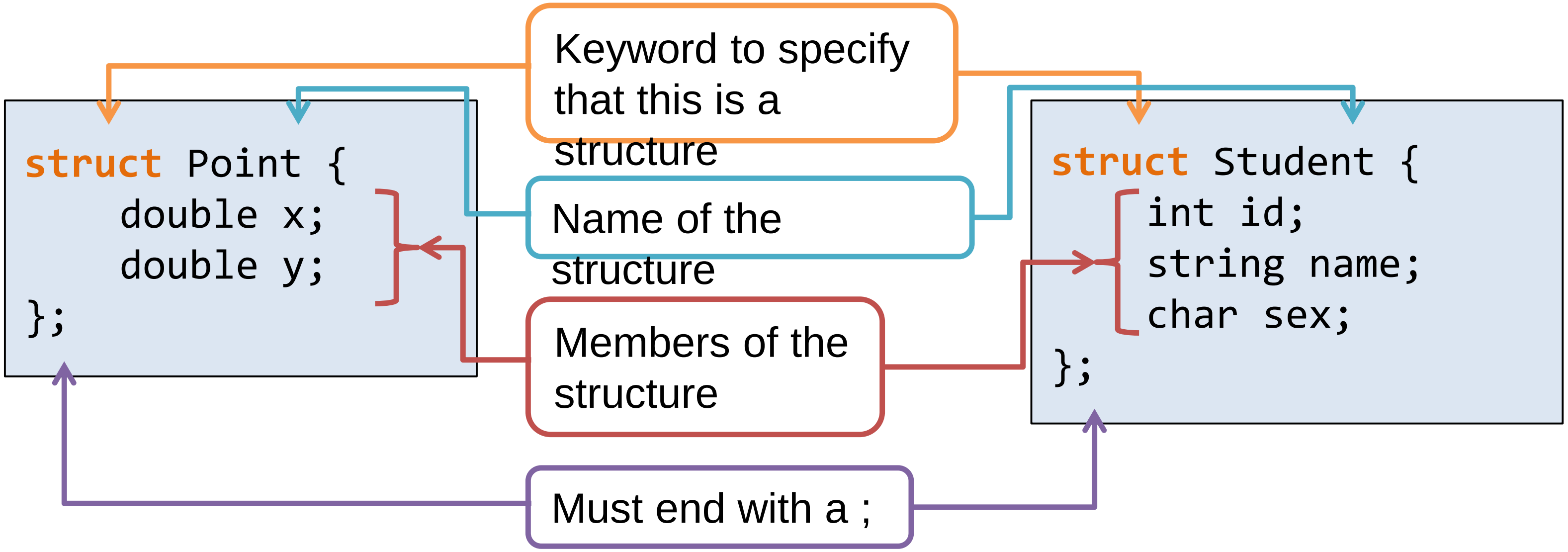
- A **structure** is a collection of one or more variables grouped together under a single name.
- The data elements in a structure are known as its member variables (or simply members), which can be of different types.
- Structures help organizing complex data
  - Allow a group of related variables to be treated as a single unit instead of separate entities
- Structures act like any basic data type
  - May be copied and assigned to variables
  - May be passed to and returned by functions



Student Record
uid (int)
assign 1 marks (int)
assign 2 marks (int)
quiz 1 marks (int)
quiz 2 marks (int)
final marks (int)
total marks (double)
grade (char)

Definition

- In C++, a structure is defined using the keyword **struct**, followed by a structure tag, a list of member variables (with types and identifiers) enclosed within a pair of braces { }, and a semicolon ;



```
struct Point {
    double x;
    double y;
};
```

Keyword to specify that this is a structure

Name of the structure

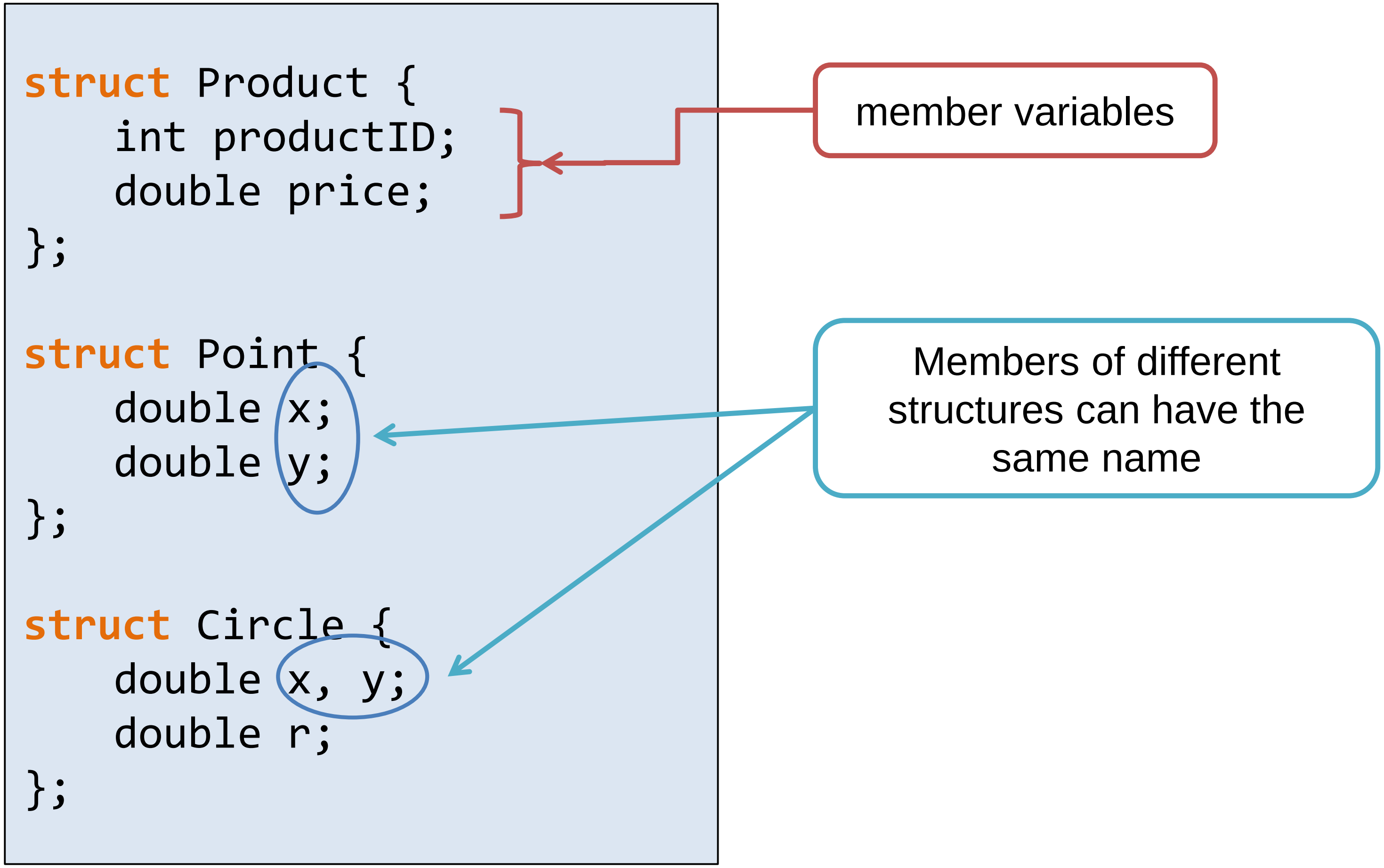
Members of the structure

Must end with a ;

```
struct Student {
    int id;
    string name;
    char sex;
};
```

Definition

- Examples



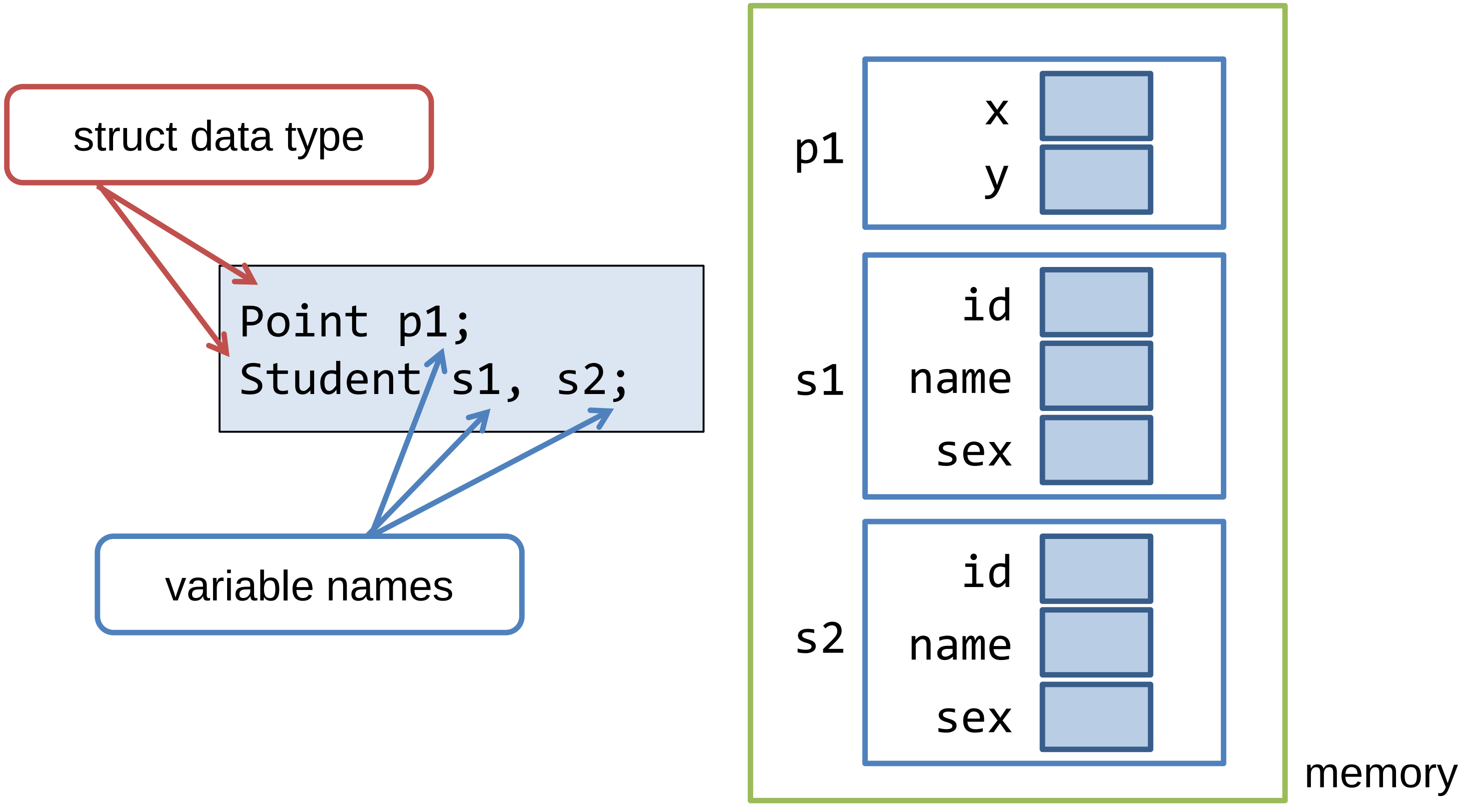
```
struct Product {
    int productID;
    double price;
};

struct Point {
    double x;
    double y;
};

struct Circle {
    double x, y;
    double r;
};
```

member variables

Members of different structures can have the same name

Declaration

- Structure variables can be declared just as what you do for the basic data types (e.g., int, char).
- So, a structure is just like a user-defined data type.



```
Point p1;
Student s1, s2;
```

Initialization

- A structure variable can be initialized in its declaration:

```
Point p1 = { 1.0, 2.0 };
Student s1 = { 3035123456, “Sze Ka Ka", 'F' };
Student s2 = s1;
```

Can be initialized with another variable of the same structure data type

Order of the members must be the same as that specified in the definition

```
Point p2 = { 1.0, 2.0, 3.0 };
```

✘

A compilation error will be generated, since there are more values than the number of members

```
Point p3 = { 1.0 };
```

There are fewer values than the number of members, remaining variables are set to zero of their data type.
`(x = 1.0, y = 0.0 )`

Member Variables

- A member variable can be used just as other variables of the basic data types.
- We may use the **dot operator .** to access the member variables of a structure.

```
Point pt1 = { 1.0, 2.0 };
Point pt2 = pt1;

pt1.x *= 2.0;    // pt1.x = pt1.x * 2.0
pt1.y /= 2.0;    // pt1.y = pt1.y / 2.0
pt2.x++;            // pt2.x = pt2.x + 1
pt2.y--;            // pt2.y = pt2.y – 1
```

What are the values of all the member variables?

result

```
pt1.x = 2.0
pt1.y = 1.0

pt2.x = 2.0
pt2.y = 1.0
```

the dot operator

```
Student s1 = { 3035123456, “Sze Ka Ka", 'F' };
int l = s1.name.length();
```

a string variable

What is the value of `l`?

```
l = 9
```

Member Variables

- Example

```
struct Student {
    int id;
    string name;
    char sex;
    double GPA;
};

Student s1;
```

What is the data type of each of the following?

s1.id	int
s1.sex	char
s1.name	string
s1	Student
Student.GPA	invalid. Student is a data type, not a variable
s1.GPA	double
s2.GPA	invalid. s2 is undeclared.

Operators

- Structure variables do not work with arithmetic (+/−), relational (>/<), equality (==) and logical operators (&&/||) by default.
  - because struct is user-defined
- All expressions below are therefore invalid.

```
Point pt1 = {1.0, 2.0}, pt2 = {3.0, 5.0};

Point pt3 = pt1 + pt2;
bool b = pt1 > pt2;
bool c = pt1 == pt2;
bool d = pt1 && pt2;
```

✗

The only operator that we may use is the assignment (=) operator

Assignment

- The assignment operator = can be used for copying a struct to another
- Example:

```
Point p1 = {1.0, 2.0}, p2;
```

```
p2 = p1;
```

is equivalent to

```
p2.x = p1.x;
p2.y = p2.y;
```

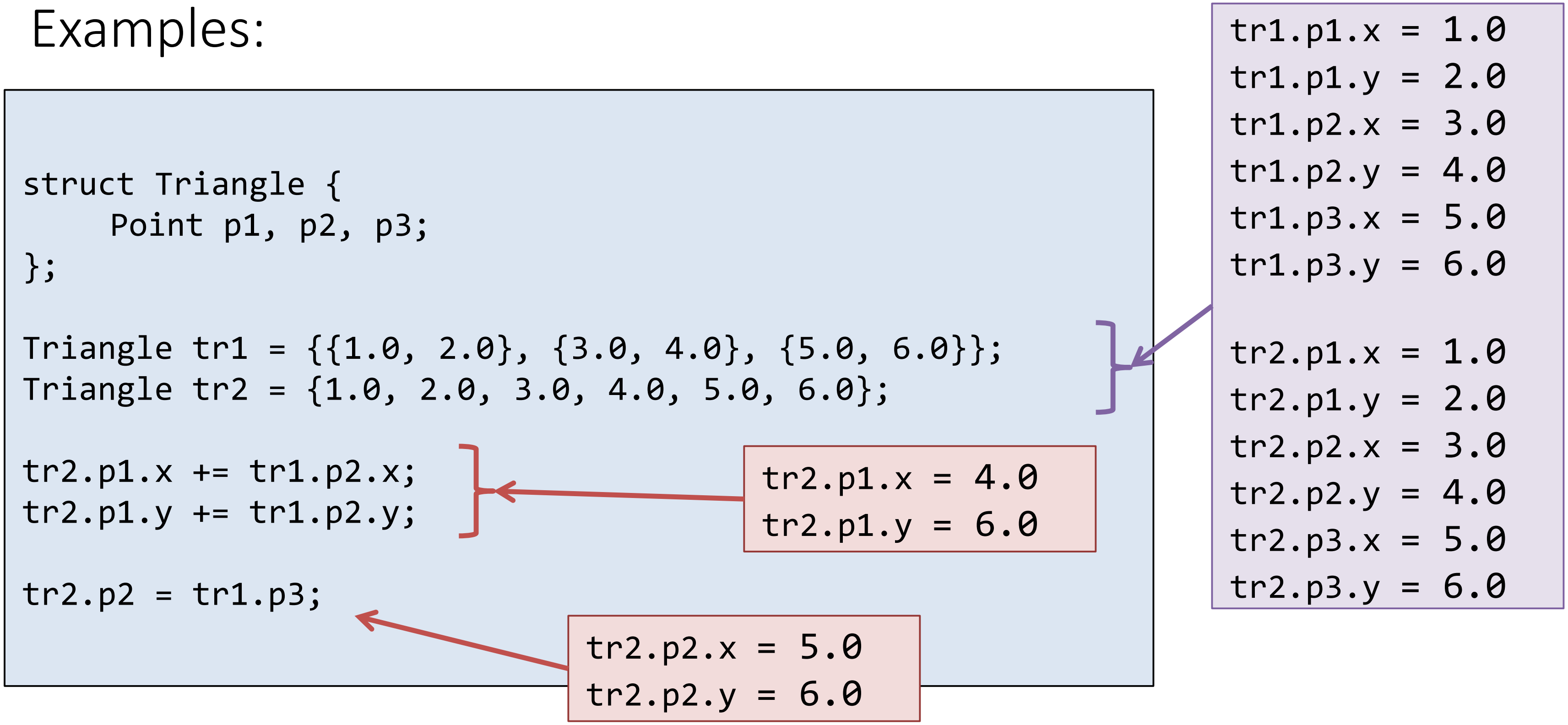
```
Point p1 = {1.0, 2.0}, p2;
p2.x = p1.y;
p2.y = p1.x;
p1 = p2;
cout << p1.x << ' ' << p1.y << endl;
```

Screen output

```
2 1
```

Nested Structures

- Structures can be nested, which means that a structure can be a member of another structure.
- Examples:



```
struct Triangle {
     Point p1, p2, p3;
};

Triangle tr1 = {{1.0, 2.0}, {3.0, 4.0}, {5.0, 6.0}};
Triangle tr2 = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};

tr2.p1.x += tr1.p2.x;
tr2.p1.y += tr1.p2.y;

tr2.p2 = tr1.p3;
```

```
tr1.p1.x = 1.0
tr1.p1.y = 2.0
tr1.p2.x = 3.0
tr1.p2.y = 4.0
tr1.p3.x = 5.0
tr1.p3.y = 6.0

tr2.p1.x = 1.0
tr2.p1.y = 2.0
tr2.p2.x = 3.0
tr2.p2.y = 4.0
tr2.p3.x = 5.0
tr2.p3.y = 6.0
```

```
tr2.p1.x = 4.0
tr2.p1.y = 6.0
```

```
tr2.p2.x = 5.0
tr2.p2.y = 6.0
```

Size of Structure

- The memory size needed for a structure may not necessarily be the total memory sizes of its variables, and the memory size may differ depending on the order of the variables too!

```
struct structA
{
   char   c;
   double d;
   int    s;
};

struct structB
{
   double d;
   int    s;
   char   c;
};
```

Try out struct_size.cpp

```
int main()
{
   cout << "sizeof(structA) = " <<  sizeof(struct structA) << endl;
   cout << "sizeof(structB) = " <<  sizeof(struct structB) << endl;

   return 0;
}
```

Given that the sizes of char, int, double are 1, 4, 8 bytes, respectively, what are the sizes of structA and structB?

Reference Only

Size of Structure

struct_size.cpp

```
int main()
{
   cout << "sizeof(structA) = " <<  sizeof(struct structA) << endl;
   cout << "sizeof(structB) = " <<  sizeof(struct structB) << endl;

   return 0;
}
```

```
struct structA
{
   char   c;
   double d;
   int    s;
};

struct structB
{
   double d;
   int    s;
   char   c;
};
```

- You will find that structA takes up 24 bytes while structB takes up 16 bytes only.
- The difference is due to how data is aligned and padded in the memory.
- In a 32-bit machine, data is stored with a 4-byte alignment and the different ordering will result in different padding. For more discussions, see: https://www.geeksforgeeks.org/data-structure-alignment/

Arrays of Structures

- Consider storing student records, we may use parallel arrays to store students' info and their marks :

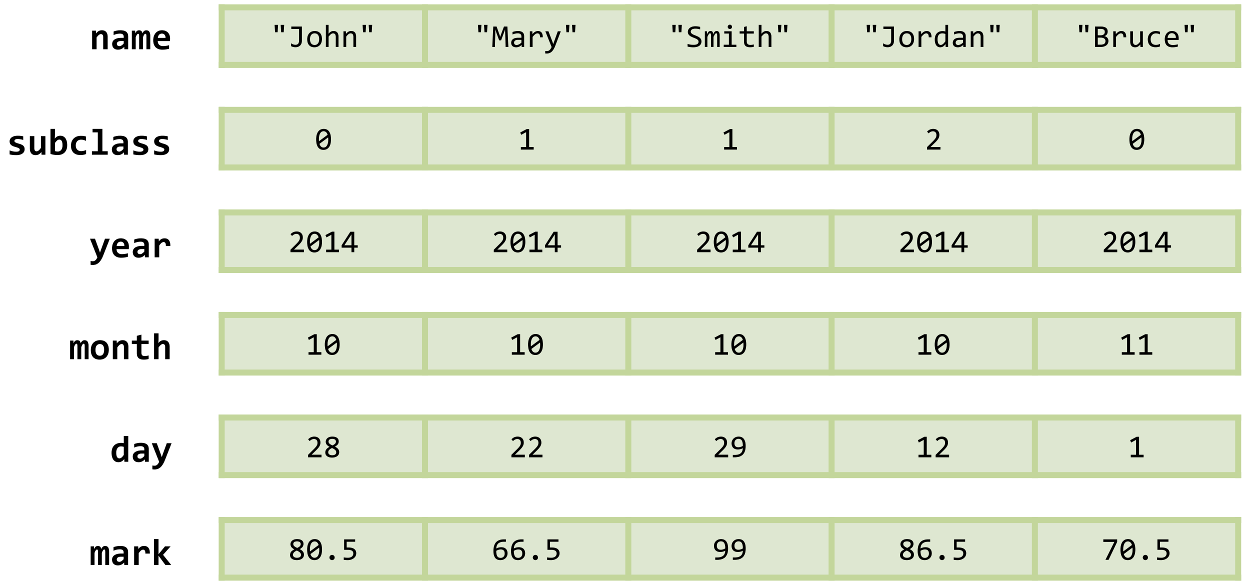
```
const int MAX = 200;

string name[MAX];
int subclass[MAX] = {0};
int year[MAX] = {0};
int month[MAX] = {0};
int day[MAX] = {0};
double mark[MAX] = {0};
```

Elements of the same index store the info for a particular student (e.g., `name[7]`, `subclass[7]`, `year[7]`, …)

- This is more often done using an array of `struct`, so that each element is a structure containing all the info for a student.

Parallel Arrays

```
string name[5];
int subclass[5];
int year[5];
int month[5];
int day[5];
double mark[5];
```

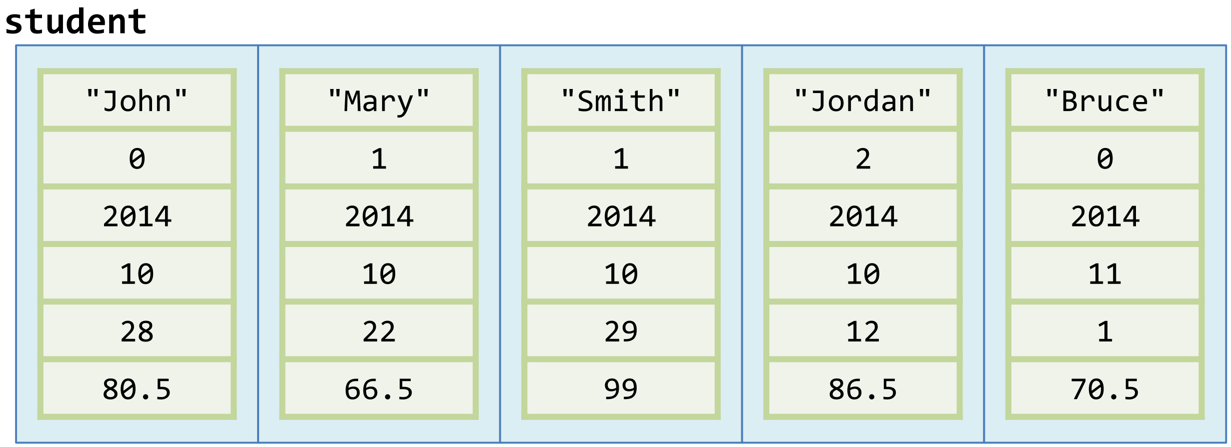


name	"John"	"Mary"	"Smith"	"Jordan"	"Bruce"
subclass	0	1	1	2	0
year	2014	2014	2014	2014	2014
month	10	10	10	10	11
day	28	22	29	12	1
mark	80.5	66.5	99	86.5	70.5

A record is referred to by `name[i]`, `subclass[i]`, `year[i]`, `month[i]`, `day[i]`, `mark[i]`

Array of Structures

```
struct Student_rec {
        string name;
        int subclass;
        int year;
        int month;
        int day;
        double mark;
};

Student_rec student[5];
```



student

"John"	"Mary"	"Smith"	"Jordan"	"Bruce"
0	1	1	2	0
2014	2014	2014	2014	2014
10	10	10	10	11
28	22	29	12	1
80.5	66.5	99	86.5	70.5

A record is referred to by `student[i].name`, `student[i].subclass`, `student[i].year`, `student[i].month`, `student[i].day`, `student[i].mark`

Arrays of Structures

- Student records stored in an array of struct:

```
const int MAX = 200;

struct Student_rec {
     string name;
     int subclass;
     int year;
     int month;
     int day;
     double mark;
};

Student_rec student[MAX];
```

This declares an array of size MAX, each element being a `Student_rec`.

array_structure.cpp

What is the data type of each of the following?

- `student` — Array of `Student_rec`
- `student[2]` — `Student_rec`
- `student[4].year` — `int`
- `Student_rec.day` — invalid. `Student_rec` is a data type, not a variable
- `student.mark` — invalid. `student` is an array, not a struct and hence no member to access

Arrays of Structures

- Examples:

```
// to print out the student records
for (int i = 0; i < 10; ++i) {
    cout << student[i].name << ' '
         << student[i].subclass << ' '
         << student[i].mark << endl;
}
```

```
// to copy student records
student[10] = student[5];
```

Think about this: How would you copy student records if they are stored using parallel arrays?

Take a look at array_structure.cpp which serves the same purpose as processmarks.cpp but using arrays of structures instead.

Structures and Functions

- Structure variables can be passed to a function either by value or by reference, and can be returned by a function like regular variables.

Pass-by-value

```
// distance between two points p and q
double point_distance( Point p, Point q ) {
    double dx = p.x – q.x;
    double dy = p.y – q.y;
    return sqrt( dx * dx + dy * dy );
}
```

spoint.cpp

Compare with this:

```
// distance between two points (x1, y1), (x2, y2)
double distance( double x1, double y1,
                    double x2, double y2) {
    …
}
```

Using structure as parameters is clearer and more structural

Structures and Functions

Pass-by-reference

```
// swap two points p and q
void swap( Point &p, Point &q ) {
     Point temp = p;
     p = q;
     q = temp;
}
```

```
// get a point from user input
Point input_point() {
     double x, y;
     cin >> x >> y;
     Point p = { x, y };
     return p;
}
```

Return a structure

spoint.cpp

More examples on struct and function

- Consider `struct Circle`, and we are to implement the following three functions:

```
struct Circle {
     double x, y;
     double r;
};
```

Function 1: To compute the area of a circle

Function 2: To enlarge a circle (i.e., increase its radius)

Function 3: To check whether a circle overlaps with another circle

What would possibly be the function prototypes for the above functions?

Think about the input and output of these functions.

More examples on struct and function

Function prototypes

- Function 1: To compute the area of a circle
  - Input: a circle, output: the area

```
double CircleArea(Circle c);
```

- Function 2: To enlarge a circle (i.e., increase its radius)
  - Input: a circle, the increment in radius; the circle radius needs to be modified

```
void EnlargeCircle(Circle &c, double radius_to_add);
```

- Function 3: To check whether a circle overlaps with another circle
  - Input: two circles, output: whether they overlaps

```
bool IsCircleOverlap(Circle c1, Circle c2);
```

More examples on struct and function

```
struct Circle {
    double x, y;
    double r;
};
```

Implementation of the three functions

- Function 1: To compute the area of a circle

```
double CircleArea(Circle c) {
    const double PI = 3.14159265358979323846;
    return PI * c.r * c.r;
}
```

circle.cpp

More examples on struct and function

```
struct Circle {
     double x, y;
     double r;
};
```

Implementation of the three functions

- Function 2: To enlarge a circle (i.e., increase its radius)

```
void EnlargeCircle(Circle &c, double radius_to_add) {
     c.r += radius_to_add;
}
```

circle.cpp

More examples on struct and function

```
struct Circle {
     double x, y;
     double r;
};
```

Implementation of the three functions

- Function 3: To check whether a circle overlaps with another circle

circle.cpp

```
bool IsCircleOverlap(Circle c1, Circle c2) {
    double dx = c1.x - c2.x;
    double dy = c1.y - c2.y;
    double centre_dist = sqrt(dx*dx + dy*dy);
    return (centre_dist <= (c1.r + c2.r));
}
```

Now, we have implemented a structure Circle and also three functions that operates on the structure. As mentioned, the structure with member variables only and all three functions are valid in both C and C++.

More examples on struct and function

- Example use of the three functions

```
int main() {
    Circle p = {1,1,2}, q = {2,2,1};

    EnlargeCircle(p, 5);
    cout << "new radius of p: " << p.r << endl;

    cout << "area of q: " << CircleArea(q) << endl;

    cout << "p and q overlap? " <<
            (IsCircleOverlap(p, q) ? "Yes" : "No") << endl;

    return 0;
}
```

circle.cpp

Structs with member variables only

- The example structs which we can come across so far contain member variables only:

```
struct Student {
     int id;
     string name;
     char sex;
     double GPA;
};
```

```
struct Product {
     int productID;
     double price;
};

struct Point {
     double x;
     double y;
};
```

```
struct Circle {
     double x, y;
     double r;
};
```

So these structure definitions are valid in both C and C++.

- In C, a struct can only contain member variable.
- In C++, you may also define member functions for struct.

Structs with Member Functions

- Let’s take a look at how we can implement member functions for structure in C++.
- Again consider the structure Circle:

```
struct Circle {
	double x, y;
	double r;
};
```

- We can implement a member function for the structure to compute the area of the circle.

Structs with Member Functions

```
struct Circle {
    double x, y;
    double r;

    double Area() {
        const double PI = 3.14159265358979323846;
        return PI * r * r;
    }
};
```

circle_structfunc.cpp

- Note how we may define a function within a struct body.
- The member function can access the member variable of the structure.
- Therefore, the function Area() does not need to take any input, and it can use the member variable r directly to compute the area.
- Compare this to the implementation of Function 1.

Structs with Member Functions

- We may also write only the function prototype inside the struct and move the function definition outside of the struct:

```
struct Circle {
    double x, y;
    double r;

    double Area();
};

double Circle::Area()
{
    const double PI = 3.14159265358979323846;
    return PI * r * r;
}
```

The scope resolution operator "::" indicates that this function Area() belongs to the structure Circle.

circle_structfunc.cpp

Structs with Member Functions

- To implement Function 2 as a member function of Circle:

```
struct Circle {
    double x, y;
    double r;

    double Area();

    void EnlargeCircle(double radius_to_add) {
        r += radius_to_add;
    };
}

double Circle::Area()
{
    const double PI = 3.14159265358979323846;
    return PI * r * r;
}
```

We’ll just leave this member function here inside the struct body, without moving it out.

Update r directly

circle_structfunc.cpp

Structs with Member Functions

- To implement Function 3 as a member function of Circle:

```
struct Circle {
    double x, y;
    double r;

    double Area();
    void EnlargeCircle(double radius_to_add) {
        r += radius_to_add;
    };

    bool IsOverlap(Circle c);
};

bool Circle::IsOverlap(Circle c) {
    double dx = x - c.x;
    double dy = y - c.y;
    double centre_dist = sqrt(dx*dx + dy*dy);
    return (centre_dist <= (r + c.r));
}
```

Check if this circle (i.e., the circle whose member function is called) overlaps with the input circle c.

circle_structfunc.cpp

Structs with Member Functions

- Example use of the three member functions

```
int main() {

    Circle p = {1,1,2}, q = {2,2,1};

    p.EnlargeCircle(5);
    cout << "new radius of p: " << p.r << endl;

    cout << "area of q: " << q.Area() << endl;

    cout << "p and q overlap? " <<
            (p.IsOverlap(q) ? "Yes" : "No") << endl;

    return 0;
}
```

Again we use the dot operator . to access the member functions of a structure

circle_structfunc.cpp

Important: This topic is optional. You are not required to write code to implement a class at this stage. The concept of class is closely related to the concept of object-oriented programming. If you want to know more, please take the course COMP2396 Object-oriented Programming and Java.

CLASSES

Optional

Abstract Data Types

- Sometimes we would like a certain data type to be associated with specific operations.
  - Integers: +, –, *, /
  - Points: translate, distance
  - Strings: length, substring, replace
- An **abstract data type** (ADT) encapsulates both the data and the methods (i.e., operations) into a package, so that users are restricted to perform only certain operations against the data inside. Also, the implementation details (how the data is stored, how the operations are carried out) of an ADT is hidden from the user (a.k.a. encapsulation or information hiding).

Optional

Abstract Data Types

- **When you want to use an ADT to solve a problem**, we only care about what can be done with them (i.e., the operations / interface), but not how they are done (i.e., the implementation).

```
string s = "I am mysterious";

cout << s.length() << endl;
cout << s.substr(0, 5) << endl;
cout << s.find("am") << endl;
```

When you use a string object, do you need to know how the string is stored internally, and how a substring is extracted by the .substr() function?

As a user for the string class, we only care about what operations are available.

This is like when we use a function, we only need to know what it does by looking at its prototype, e.g.,

```
double sqrt( double x);
```

but we don't care about how it comes up with the result.

Optional

Abstract Data Types

Compare with this: When we use `struct Point`, we need to know how the coordinates are stored if we need to write a function to do anything on them.

```
struct Point {
     double x;
     double y;
};
```

```
// distance between two points p and q
double point_distance( Point p, Point q ) {
     double dx = p.x – q.x;
     double dy = p.y – q.y;
     return sqrt( dx * dx + dy * dy );
}
```

What if we later change our mind and want to use an array of 2 doubles instead to store x and y? Then any function making use of Point (e.g., point_distance()) will need to be modified.

```
struct Point {
     double v[2];
};
```

Hence, it would be great if an ADT can provide “interfaces” for accessing its data, so other developers who want to use the ADT do not need to care about the internal representation/implementation (i.e., even if these changes, one doesn’t need to change his code that uses the ADT).

Optional

Classes

- ADTs are implemented using classes in C++. A class contains data (member variables) and methods (member functions) and is divided into two sections.

```
class Point {
public:
    double getX();
    double getY();
    double distance(Point & p);
    void   translate(Point & p);

private:
    double x;
    double y;

};
```

member functions

member variables

Private Section
Data/methods here can only be accessible by the member functions of the same class.
Implementation of the ADT

Public Section
Data/methods here are accessible by the application program/code using an object of this class.
Abstraction of the ADT

The public and private sections may contain both member variables and functions. However, we should avoid defining member variables in the public section in order to enforce access of ADT internal data via member functions. Hence the member functions serve as the "interfaces" which we mentioned earlier. You will learn more about class design in the course COMP2396 Object-oriented Programming and Java.

For C++ struct, member variables and functions are by default, while they are by default private for class.

Optional

Class Definitions

Keyword for defining a class

```cpp
class Point {
public:
    double getX() { return x; }
    double getY() { return y; }
    void setCoord(double s, double t) {
        x = s;
        y = t;
    }

    double distance(Point & p);
    void   translate(Point & p);

private:
    double x;
    double y;

};
```

A member function can access the private variable of the class. Since it is defined under the public section, others can have “access” to the private variables via this function.

Member function definitions

Access specifier

Member function prototypes. Note that these functions are not defined yet (i.e., we need to define them somewhere else).

As designer of a class, you may choose whether to include the definition or just the prototype for a member function inside a class definition. There are some design considerations, but we won’t go in the details here (again we’ll leave it to the course COMP2396).

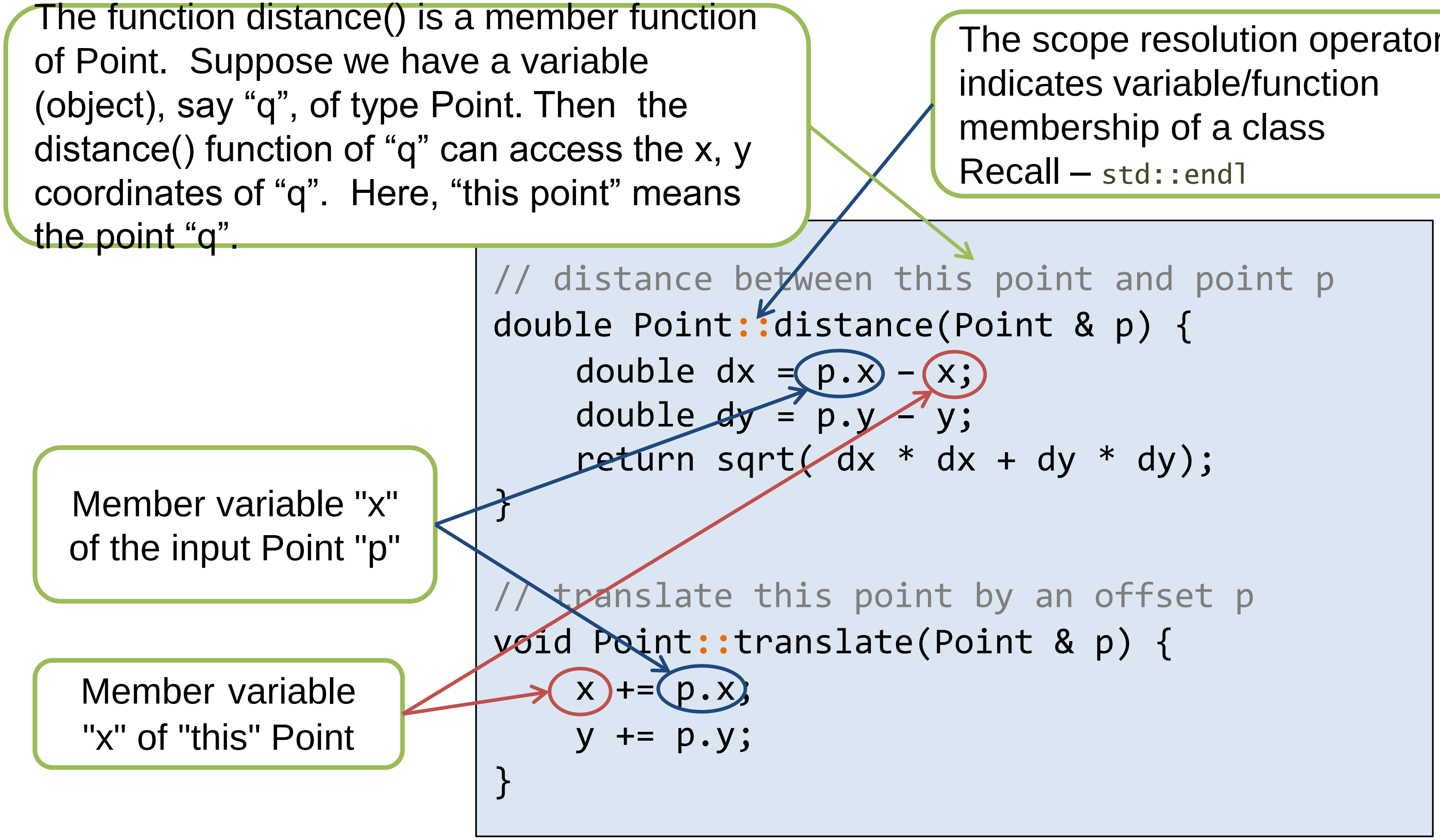
Member variable declarations

Ends with a ;

Optional

Member Functions

Member functions can be defined outside the class body:



The function distance() is a member function of Point. Suppose we have a variable (object), say “q”, of type Point. Then the distance() function of “q” can access the x, y coordinates of “q”. Here, “this point” means the point “q”.

The scope resolution operator "::" indicates variable/function membership of a class
Recall – `std::endl`

Member variable "x" of the input Point "p"

Member variable "x" of "this" Point

```
// distance between this point and point p
double Point::distance(Point & p) {
    double dx = p.x - x;
    double dy = p.y - y;
    return sqrt( dx * dx + dy * dy);
}

// translate this point by an offset p
void Point::translate(Point & p) {
    x += p.x;
    y += p.y;
}
```

Optional

Class Declaration

- To declare an object (variable) for a class:

```
Class_name    object_name1, object_name2, …;
```

Examples:

```
Point p1, p2;
string s1("abc");
```

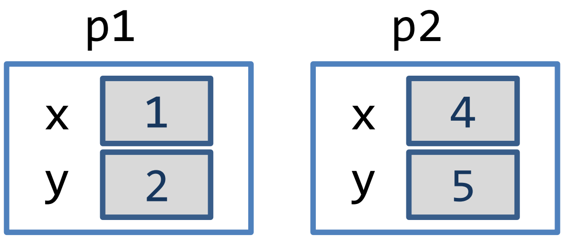
Since class are just user-defined data types, you can declare just like how you declare an int, a double, etc.

"p1", "p2" are Point **objects**, "s1" is a string **object** (YES, string is just a class)

- Each object can then retain their own values for each member variables

Examples:

```
p1.setCoord(1, 2);
p2.setCoord(4, 5);
```



Optional

Multiple Files Compilation for Class Implementation

- It is a common practice to put the codes for a class in a separate file, so that the class can be reused by another file or program.
- We also further separate the definition and implementation of a class in .h and .cpp files, respectively. Doing so allows users of a class to focus only on the class interface (which defines how to use the class) in the header file (.h)

```
#include "point.h"

int main()
{
	Point p, q;
	…
	p.distance(q);
	…
	return 0;
}
```

main.cpp

Main program

```
class Point
{
public:
	…

private:
	…
};
```

point.h

Class interface

```
double Point::distance(Point & p) {
	…
}

void Point::translate(Point & p)
{
	…
}
```

point.cpp

Class implementation

Any other program that wants to use Point can just include "point.h".

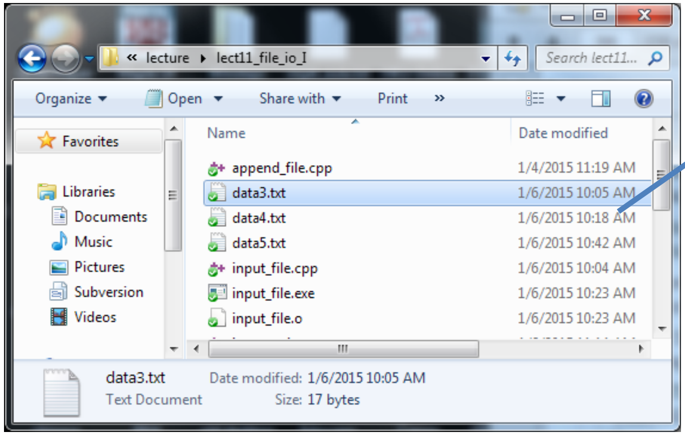
Check the sample programs and also the Makefile for this example.

Part II

FILE I/O

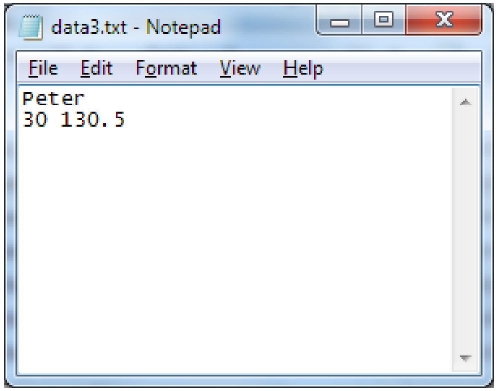
File Input/Output

- Files are used for storing data permanently. The data is stored in the hard drive of your computer and you can read and write from it with your program.



Name	Date modified
append_file.cpp	1/4/2015 11:19 AM
data3.txt	1/6/2015 10:05 AM
data4.txt	1/6/2015 10:18 AM
data5.txt	1/6/2015 10:42 AM
input_file.cpp	1/6/2015 10:04 AM
input_file.exe	1/6/2015 10:23 AM
input_file.o	1/6/2015 10:23 AM

data3.txt Date modified: 1/6/2015 10:05 AM
Text Document Size: 17 bytes



```
Peter
30 130.5
```

Contents of "data3.txt"

- C++ simply views a file as a sequence of bytes:

'P'	'e'	't'	'e'	'r'	'\n'	'3'	'0'	' '	'1'	'3'	'0'	'.'	'5'	eof

A file in the file system named "data3.txt"

end of file marker

Streams

- C++ uses a convenient abstraction called streams to perform input and output operations in sequential media, e.g.,
  - cout is a stream object for sending output to the screen
  - cin is a stream object for taking input from keyboard
- C++ provides two classes, namely ofstream and ifstream, for writing and reading data to and from files
- To use the classes ofstream and ifstream, simply include the header file fstream, i.e.,

```
#include <fstream>
```

WRITE TO FILE

Output File Stream

- A basic example for **creating and writing** to a file

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
    ofstream fout;
    fout.open("data1.txt");

    if ( fout.fail() ) {
        cout << "Error in file opening!"
              << endl;
        exit(1);
    }
```

```
    string name = "Peter";
    int age = 30;
    double weight =

    fout << name
         << weig
    fout.close()

    return 0;
}
```

Include the file stream header file

Create an **ofstream** (output file stream) object and connect it to an **external file** named "data1.txt"

These two statements can be replaced by:
ofstream fout ("data1.txt");

After executing these two statements, a file will be created in your hard drive (in the same directory as your program executable):

data1.txt

output_file.cpp

Output File Stream

- A basic example for **creating and writing** to a file

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
   ofstream fout;
   fout.open("data1.txt");

   if ( fout.fail() ) {
      cout << "Error in file opening!"
           << endl;
      exit(1);
   }
```

```
    string name = "Peter";
    int age =
    double w

    fout << name << " " << age << " "
         << weight << endl;
    fout.close();

    return 0;
}
```

This **if** block serves to exit the program if unable to create file.

Function **exit** forces a program to terminate immediately, and is often used to terminate a program when an error is detected in the input or if a file to be processed by the program cannot be opened.

data1.txt

output_file.cpp

Output File Stream

- A basic example for **creating and writing** to a file

```
#include <iostream>
#include <fstream>
```

Write to the file stream **fout** using the insertion operator
<<
(just as what we do with cout)

```
int main()
{
   ofstream fout;
   fout.open("data1.txt");

   if ( fout.fail() ) {
      cout << "Error in file openi
               << endl;
      exit(1);
   }
```

```
   string name = "Peter";
   int age = 30;
   double weight = 130.5;

   fout << name << " " << age << " "
        << weight << endl;
   fout.close();

   return 0;
}
```

Finally disconnects the file stream **fout** from the external file

data1.txt

```
Peter 30 130.5\n
eof
```

output_file.cpp

Summary
Steps for Creating and Writing to a File

1. Declare an output stream variable.

```
ofstream fout;
```

2. Open the file

```
fout.open("data.txt");
```

3. Check if there is any error in opening the file

```
if (fout.fail())
```

4. Use the insertion operator **<<** to write to file

```
fout << "12345";
```

5. Close the file

```
fout.close();
```

```
string filename = "data.txt";
fout.open(filename.c_str());
```

if the file name is stored as string

Appending Data to a File

- When opening a file for output using the member function **open()**, a new file will be created if the file does not already exist, otherwise the content of the existing file will be erased
- To keep the content of the existing file and append new data to it, supply the constant value **ios::app** as a second argument to the member function open(), e.g.,

```
fout.open("data2.txt", ios::app)
```

Appending Data to a File

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
   ofstream fout;
   fout.open("data2.txt", ios::app);

   if (fout.fail()) {
      cout << "Error in file opening!"
         << endl;
      exit(1);
   }
```

```
   string name = "John";
   int age = 25;
   double weight = 129.3;

   fout << name << " " << age << " "
      << weight << endl;
   fout.close();

   return 0;
}
```

append_file.cpp

data2.txt
(before executing the program)

```
Peter 30 130.5\n
eof
```

data2.txt
(after executing the program)

```
Peter 30 130.5\n
John 25 129.3\n
eof
```

READ FROM FILE

Input File Stream

- A basic example for reading from an existing file

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
    char filename[80] = "data3.txt";
    ifstream fin;
    fin.open(filename);

    if

    }
```

input_file.cpp

```
    string name;
    int age
    double

                          >> weight;

                          ' << age << ", "

}
```

Include the file stream header file

Create an **ifstream** (input file stream) object and connect it to an **external file** named "data3.txt"

These few statements can be replaced by:
ifstream fin ("data3.txt");

Since the **open()** function accepts only a C-string as the input parameter, if the file name is stored in a string class, we will need to write:
string filename = "data3.txt"
ifstream fin(filename.c_str());

data3.txt

```
Peter\n
30 130.5\n
eof
```

Input File Stream

- A basic example for **reading from an existing file**

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
   char filename[80] = "data3.txt";
   ifstream fin;
   fin.open(filename);

   if ( fin.fail() ){
      cout << "Error in file opening!"
         << endl;
      exit(1);
   }
```

input_file.cpp

```
   string name;
   int age;
   double weight;

   fin >> name >> age >> weight;
   fin.close();

                 " << age << ", "
                dl;
   return 0;
}
```

Exit the program if the file does not exist

data3.txt

```
Peter\n
30 130.5\n
eof
```

Input File Stream

- A basic example for **reading from an existing file**

```
#include <iostream>
#include <fstream>

int main()
{
                                    ;

    if ( fin.fail() ){
        cout << "Error in file opening!"
            << endl;
        exit(1);
    }
```

input_file.cpp

Read from the file stream **fin** using the extraction operator >>
(just as what we do with cin)

Finally disconnects the file stream **fin** from the external file

```
    string name;
    int age;
    double weight;

    fin >> name >> age >> weight;
    fin.close();

    cout << name << ", " << age << ", "
        << weight << endl;
    return 0;
}
```

data3.txt

```
Peter\n
30 130.5\n
eof
```

Screen output

```
Peter, 30, 130.5
```

Summary

Steps for Reading Input from a File

1. Declare an **ifstream** object.

```
ifstream fin;
```

2. Open the file

```
fin.open("data.txt");
```

3. Check if there is any error in opening the file

```
if (fin.fail())
```

4. Read data from file using the extraction operator **>>**

```
fin >> x;
```

5. Close the file

```
fin.close();
```

Reading until End of File (EOF)

- Very often, data have to be extracted sequentially from an input file until the end of file (eof) has been reached (because we don't know the length of a file in advance)
- This can be done by using a while loop as follows:

```
while (fin >> x)
{
    …
}
```

- The return value of the expression fin >> x:
  - A nonzero (true) value indicates a datum has been read successfully
  - A zero (false) value indicates the eof has been reached and no datum has been read

Reading until End of File (EOF)

- Example

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
    ifstream fin;
    fin.open("data4.txt");
    if (fin.fail()) {
        cout << "Error in file opening!"
            << endl;
        exit(1);
    }
```

```
    double x, sum = 0;

    while (fin >> x) {
        sum += x;
    }

    fin.close();
    cout << "Total = " << sum
          << endl;
    return 0;
}
```

read_till_eof.cpp

Read and sum until end of file

data4.txt

```
20.0 40.0 60.0 eof
```

Screen output

```
Total = 120
```

Reading Lines From a File

- Sometimes, data in a file may need to be processed in a line by line manner, e.g., each line stores the record of one person
- The library function getline() can be used to read in a line from an input file stream object and store it as a string object, e.g.,

```
getline(fin, str);
```

fin is an input file stream object and str is a string object (both are call-by-reference parameters)

- Similarly, the return value of getline() can be used to check if the eof has been reached
  - A nonzero (true) value indicates a line has been read successfully
  - A zero (false) value indicates the eof has been reached and no line has been read

Reading Lines From a File

- Example:

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
    ifstream fin;
    fin.open("data5.txt");
    if (fin.fail()) {
        cout << "Error in file opening!"
            << endl;
        exit(1);
    }
```

```
    string line;

    while ( getline(fin, line) ) {
        cout << line << endl;
    }

    fin.close();
    return 0;
}
```

read_line_file.cpp

data5.txt

```
Peter 30 130.5\n
John 129.3\n
eof
```

Screen output

```
Peter 30 130.5
John 129.3
```

Input String Stream

- While C++ considers file as a stream of characters, it can also take strings as a stream of characters too. The class istringstream is provided for extracting data from a string. To use this class, simply include the header file <sstream>, i.e.,

```
#include <sstream>
```

- An input string stream object can be declared using the class name istringstream and initialized with a string object as follows

```
string str;
istringstream iss(str);
```

- Data can then be extracted from the input string stream using the extraction operator >>

```
int age;
iss >> age;
```

Input String Stream

- Similarly, data can be extracted sequentially from the stream until the end of string has been reached by checking the return value of the expression

```
input_string_stream >> variable
```

  - A nonzero (true) value indicates a datum has been read successfully
  - A zero (false) value indicates the end of string has been reached and no datum has been read

Input String Stream

- Example

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

int main()
{
    string line=" apple orange banana ", word;

    istringstream line_in(line);

    while ( line_in >> word ) {
        cout << "\"" << word << "\""
            << endl;
    }

    return 0;
}
```

input_string_stream.cpp

Screen output

```
"apple"
"orange"
"banana"
```

Stream Output Formatting

- Sometimes you may want to have the output from your program to be displayed (on screen) or stored (in file) in a specific format
  - Floating-point numbers: **0.00001** or **1e-5**? **15** or **15.000**?
  - Formatted tabular output:

```
Peter  30   130.5
John    6   129.3
Mary   18    34.5
```

How to set the width of each column?

How to set the column alignment?

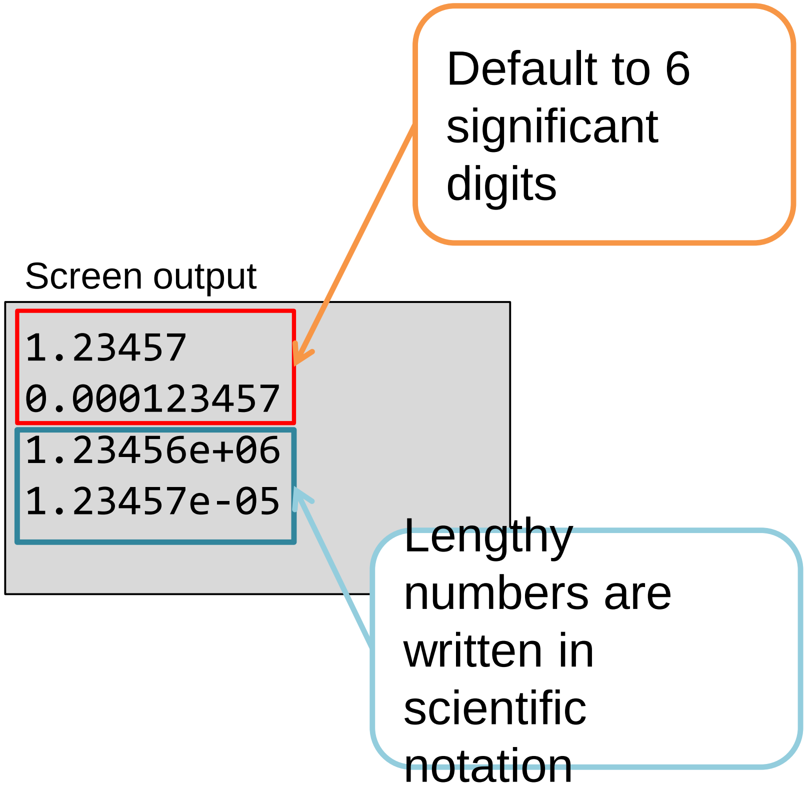
- We may use the output manipulators to format the output. We've come across some examples:
  - **endl**, to move the insertion point to the beginning of the next line
  - **setw**, to set the width of the column for the next output value

Default floating-point notation

- Example

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

int main()
{
    double a = 1.2345678;
    double b = 0.00012345678;
    double c = 1234567.8;
    double d = 0.000012345678;

    cout << a << endl << b << endl
        << c << endl << d << endl;
    return 0;
}
```

default_float.cpp



Screen output

```
1.23457
0.000123457
1.23456e+06
1.23457e-05
```

Default to 6 significant digits

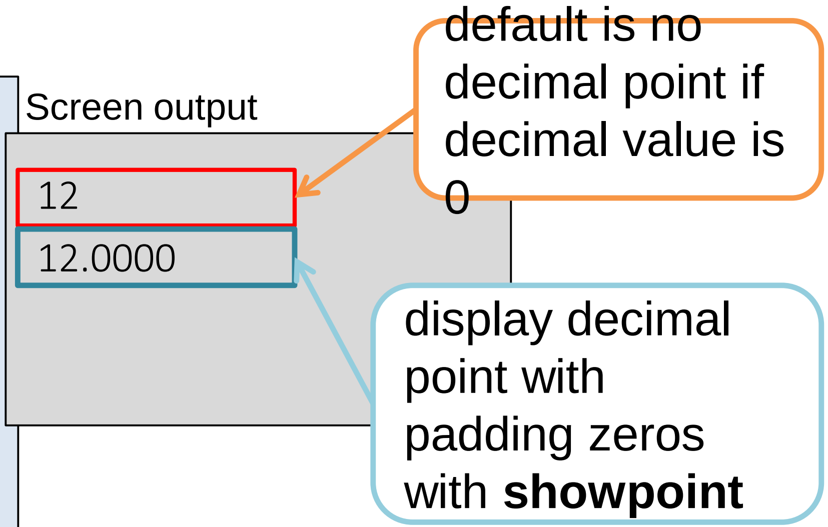
Lengthy numbers are written in scientific notation

showpoint Manipulator

- Example

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

int main()
{
    double e = 12.0;

    cout << e << endl;
    cout << showpoint << e << endl;

    return 0;
}
```

default_float.cpp



Screen output

```
12
12.0000
```

default is no decimal point if decimal value is 0

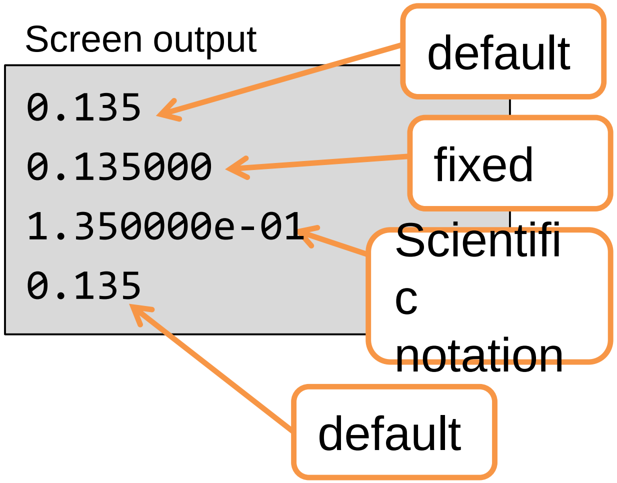
display decimal point with padding zeros with **showpoint**

can be unset with the **noshowpoint** manipulator

fixed / scientific Manipulators

- **fixed** to write floating-point numbers as fixed decimal
- **scientific** to output floating-point numbers in scientific notation

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

int main()
{
    double f = 0.135;
    cout << f << endl;
    cout << fixed << f << endl;
    cout << scientific << f << endl;

    cout.unsetf(ios_base::floatfield);
    cout << f << endl;
    return 0;
}
```



Screen output

```
0.135
0.135000
1.350000e-01
0.135
```

default

fixed

Scientific notation

default

manipulator_fixed.cpp

setprecision Manipulator

- With the default floating-point notation, **setprecision** specifies the maximum number of meaningful digits before and after the decimal point.

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

int main()
{
    double a = 1.2345678;
    double b = 1234567.8;
    cout << a << '\n' << b << "\n\n";

    cout << setprecision(2);
    cout << a << '\n' << b << '\n';

    return 0;
}
```

Screen output

```
1.23457
1.23457e+006

1.2
1.2e+006
```

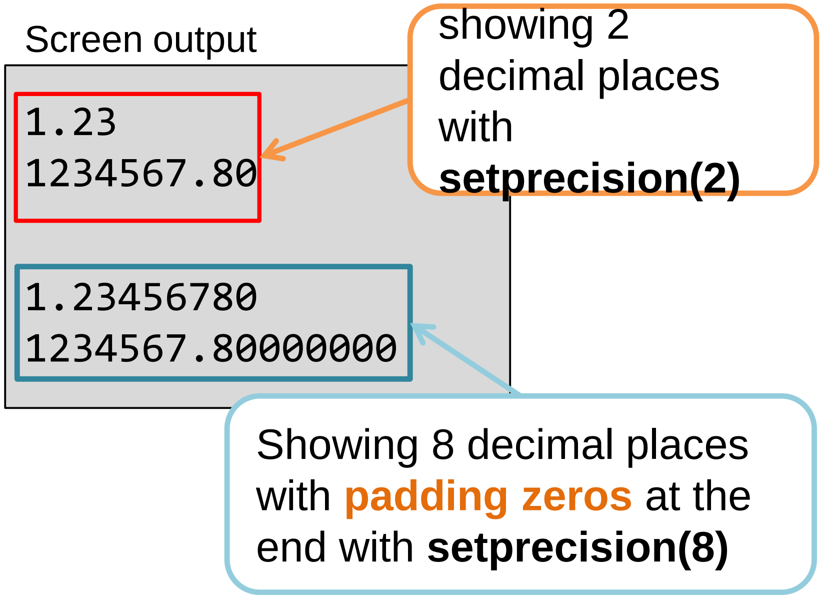
showing 2 significant digits with **setprecision(2)**

manipulator_setprecision.cpp

setprecision Manipulator

- With the fixed or scientific notation, **setprecision** specifies the exact number of digits after the decimal point. By default, 6 decimal places are used.

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

int main()
{
    double a = 1.2345678;
    double b = 1234567.8;

    cout << fixed << setprecision(2);
    cout << a << '\n' << b << "\n\n";

    cout << setprecision(8);
    cout << a << '\n' << b << '\n';
    return 0;
}
```



Screen output

```
1.23
1234567.80

1.23456780
1234567.80000000
```

showing 2 decimal places with **setprecision(2)**

Showing 8 decimal places with **padding zeros** at the end with **setprecision(8)**

Try using setprecision with scientific notation

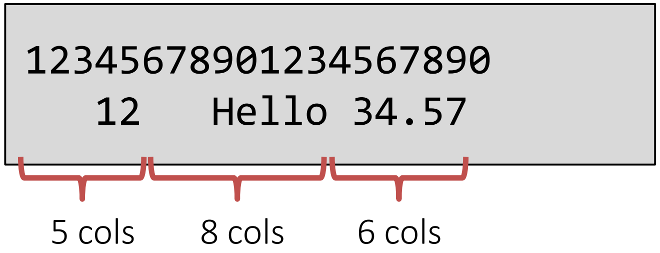
manipulator_setprecision.cpp

setw Manipulator

- Use setw to output a string or a number in a specific number of columns (the output is right-justified).

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

int main()
{
    int x = 12;
    string a = "Hello";
    double b = 34.567;

    cout << fixed << setprecision(2);
    cout << "12345678901234567890\n";

    cout << setw(5) << x << setw(8) << a;
    cout << setw(6) << b << endl;

    return 0;
}
```

Screen output



```
12345678901234567890
   12   Hello 34.57
```

For those manipulators that accept parameters such as setw(x), include the **<iomanip>** header; otherwise for those manipulator without parameters such as fixed, include the **<iostream>** header

manipulator_setw.cpp

setfill Manipulator

- With setw, if the specified number of columns > the required number of columns, the unused columns are filled with spaces. We may use setfill to fill the unused columns with other characters.

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

int main()
{
    int x = 12;
    string a = "Hello";
    double b = 34.567;

    cout << fixed << setprecision(2);
    cout << "12345678901234567890\n";
```

```
    cout << setfill('*');
    cout << setw(5) << x << setw(8) << a;
    cout << setw(6) << b << endl;

    return 0;
}
```

Screen output

```
12345678901234567890
***12***Hello*34.57
```

manipulator_setw.cpp

left / right Manipulators

- With setw, the default output is right-justified within a column. Use the **left** and **right** manipulators to set the output to be left-justified or right-justified, respectively.

```
…
cout << "12345678901234567890\n";
cout << setfill('-');

cout << left;
cout << setw(5) << x << setw(8) << a;
cout << setw(6) << b << endl;

cout << right;
cout << setw(5) << x << setw(8) << a;
cout << setw(6) << b << endl;
…
```

manipulator_setw.cpp

Screen output

```
12345678901234567890
12---Hello---34.57-
---12---Hello-34.57
```

`left` and `right` are defined in <iostream>

Further References on File I/O

- C++ Language Tutorial: Input/Output with files
  http://www.cplusplus.com/doc/tutorial/files/
- C++ Library Reference: ifstream class
  http://www.cplusplus.com/reference/fstream/ifstream/
- C++ Library Reference: istringstream class
  http://www.cplusplus.com/reference/sstream/istringstream/
- C++ Library Reference: ofstream class
  http://www.cplusplus.com/reference/fstream/ofstream/
- C++ Library Reference: ofstream class
  http://www.cplusplus.com/reference/library/manipulators/

Part III

RECURSION

What are we going to learn?

- Recursive definition
- Recursive functions in C++
- Flow of control in recursive functions
- General structure of a recursive function
- Examples of recursive functions
- Stack overflow problem
- Recursion versus iteration

Recursive Definition

- Some problems are **recursive** by nature, i.e., it has a recursive definition which means that the problem can be defined in terms of a smaller version of itself.

Consider the factorial of a nonnegative integer:

Definition 1

0! = 1

n! = n × (n–1) × (n–2) × … × 2 × 1,   if n > 0

An iterative definition

Definition 2

0! = 1

n! = n × (n–1)!,   if n > 0

A recursive definition

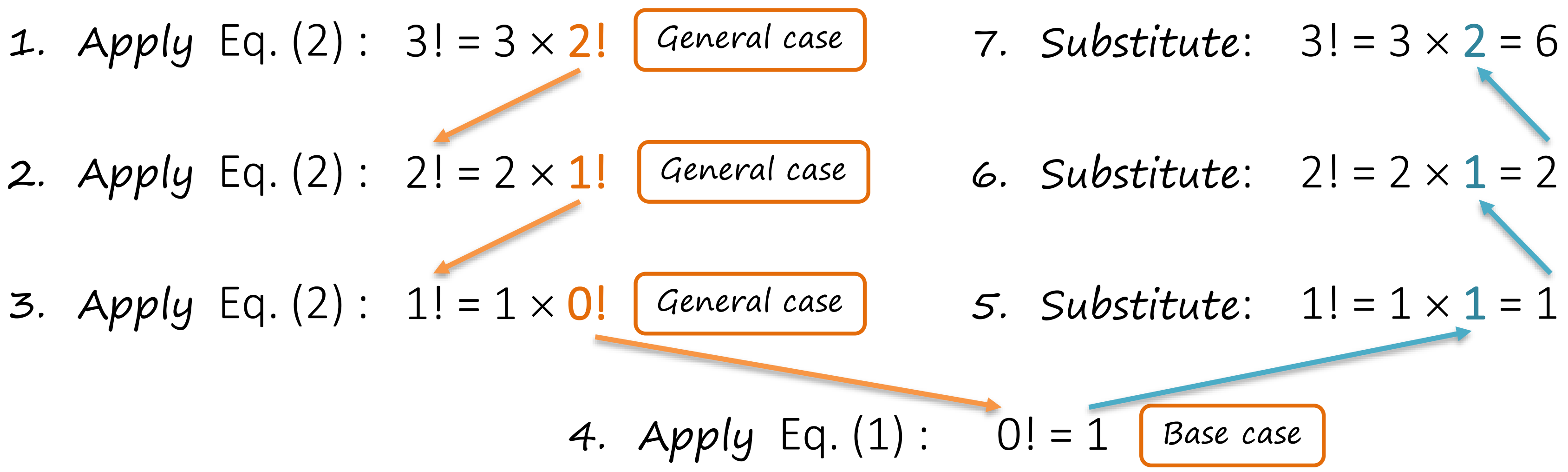
Recursive Definition

- How does a recursive definition work?

Base case Eq. (1): $0! = 1$

General case Eq. (2): $n! = n \times (n-1)!$, if $n > 0$

To calculate 3! :



1. *Apply* Eq. (2) : $3! = 3 \times 2!$ *General case*

2. *Apply* Eq. (2) : $2! = 2 \times 1!$ *General case*

3. *Apply* Eq. (2) : $1! = 1 \times 0!$ *General case*

4. *Apply* Eq. (1) : $0! = 1$ *Base case*

5. *Substitute*: $1! = 1 \times 1 = 1$

6. *Substitute*: $2! = 2 \times 1 = 2$

7. *Substitute*: $3! = 3 \times 2 = 6$

Recursive Definition

- Properties for a recursive definition
  - Must have one (or more) base cases
  - The general case must be reduced to a base case eventually
  - The base case terminates the recursion

- Some more examples of recursive problems
  - Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
    - $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, $F_1 = 1$

General case: a number is the sum of its previous two numbers

  - Tower of Hanoi

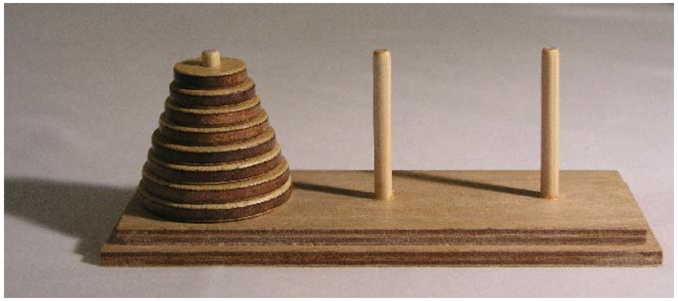


Image from Wikimedia Commons

Recursive Function

- In C/C++, we may write recursive function to implement recursion.
- A recursive function is one that contains a call to itself.

```
int factorial(int num)
{
   if (num == 0)
       return 1;
   else
       return num * factorial(num - 1);
}
```

Base case

General case

factorial.cpp

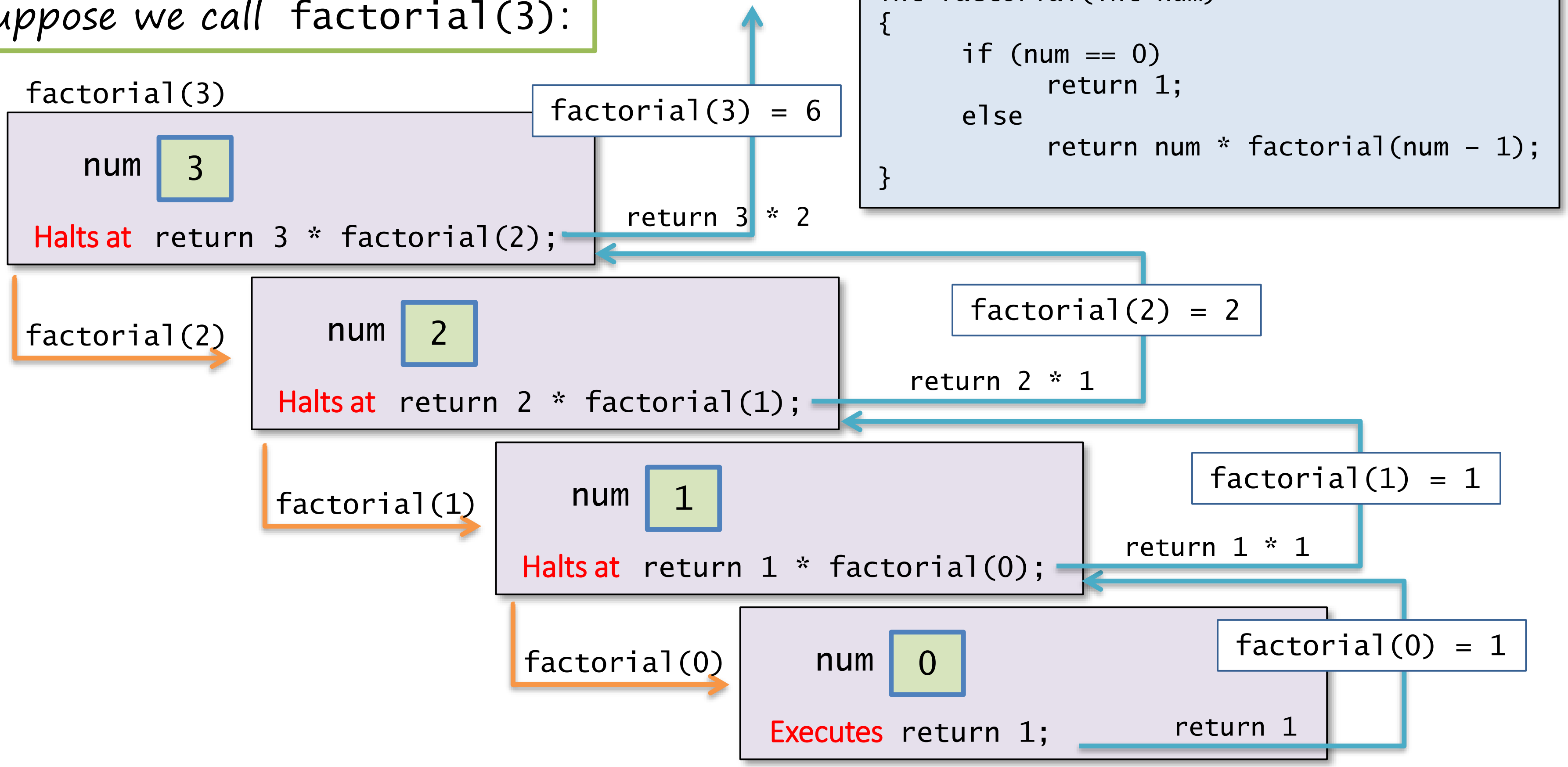
Since the argument passed to the functions keeps decrementing by 1, we are certain that the base case will be reached eventually which stops the recursion.

Flow of Control

- Flow of control is essentially the same as function calls, except that the same function is repeatedly called.

Suppose we call factorial(3):

```
int factorial(int num)
{
      if (num == 0)
            return 1;
      else
            return num * factorial(num – 1);
}
```



factorial(3)

num 3

Halts at return 3 * factorial(2);

factorial(3) = 6

return 3 * 2

factorial(2)

num 2

Halts at return 2 * factorial(1);

factorial(2) = 2

return 2 * 1

factorial(1)

num 1

Halts at return 1 * factorial(0);

factorial(1) = 1

return 1 * 1

factorial(0)

num 0

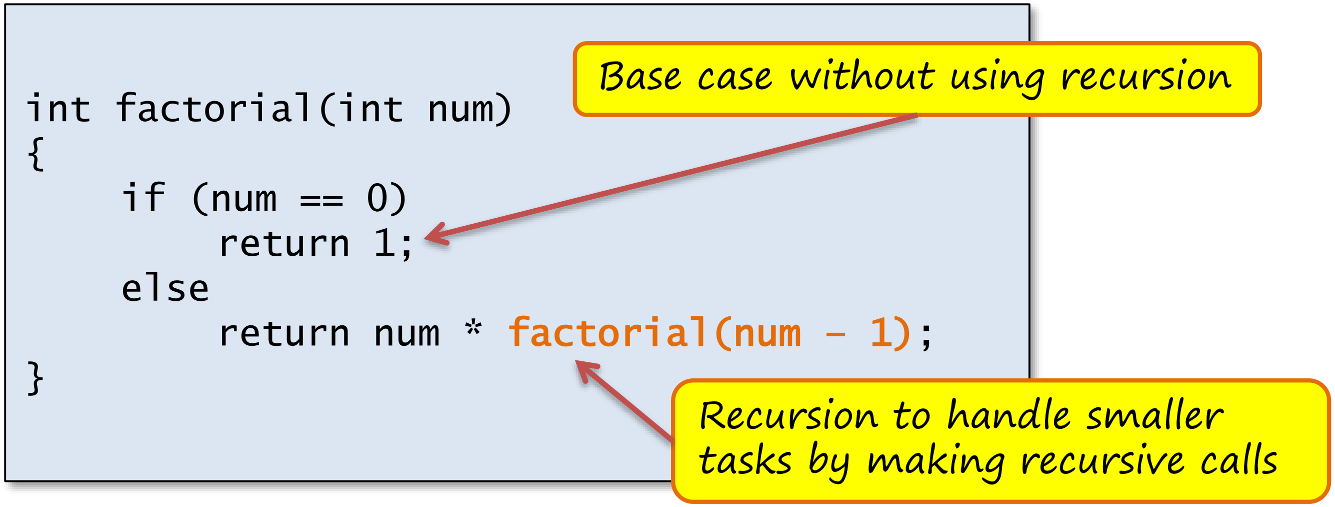
Executes return 1;

factorial(0) = 1

return 1

General Structure

- The process of calling a function itself recursively can be repeated any number of times.
- How to avoid infinite recursion?
- General structure for a recursive function definition:
  - Having one or more recursive calls to itself to accomplish smaller tasks
  - Having one or more base cases without using recursive calls to terminate the recursion



```
int factorial(int num)
{
    if (num == 0)
        return 1;
    else
        return num * factorial(num – 1);
}
```

Base case without using recursion

Recursion to handle smaller tasks by making recursive calls

Example: Fibonacci Sequence

Recursive definition for the problem:

$F_0 = 0, \; F_1 = 1$

Base case

$F_n = F_{n-1} + F_{n-2}, \; \text{if } n > 1$

Recursion

The sequence:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

```
int fib(int num)
{
    if (num < 2)
        return num;
    else
        return fib(num-1) + fib(num-2);
}
```

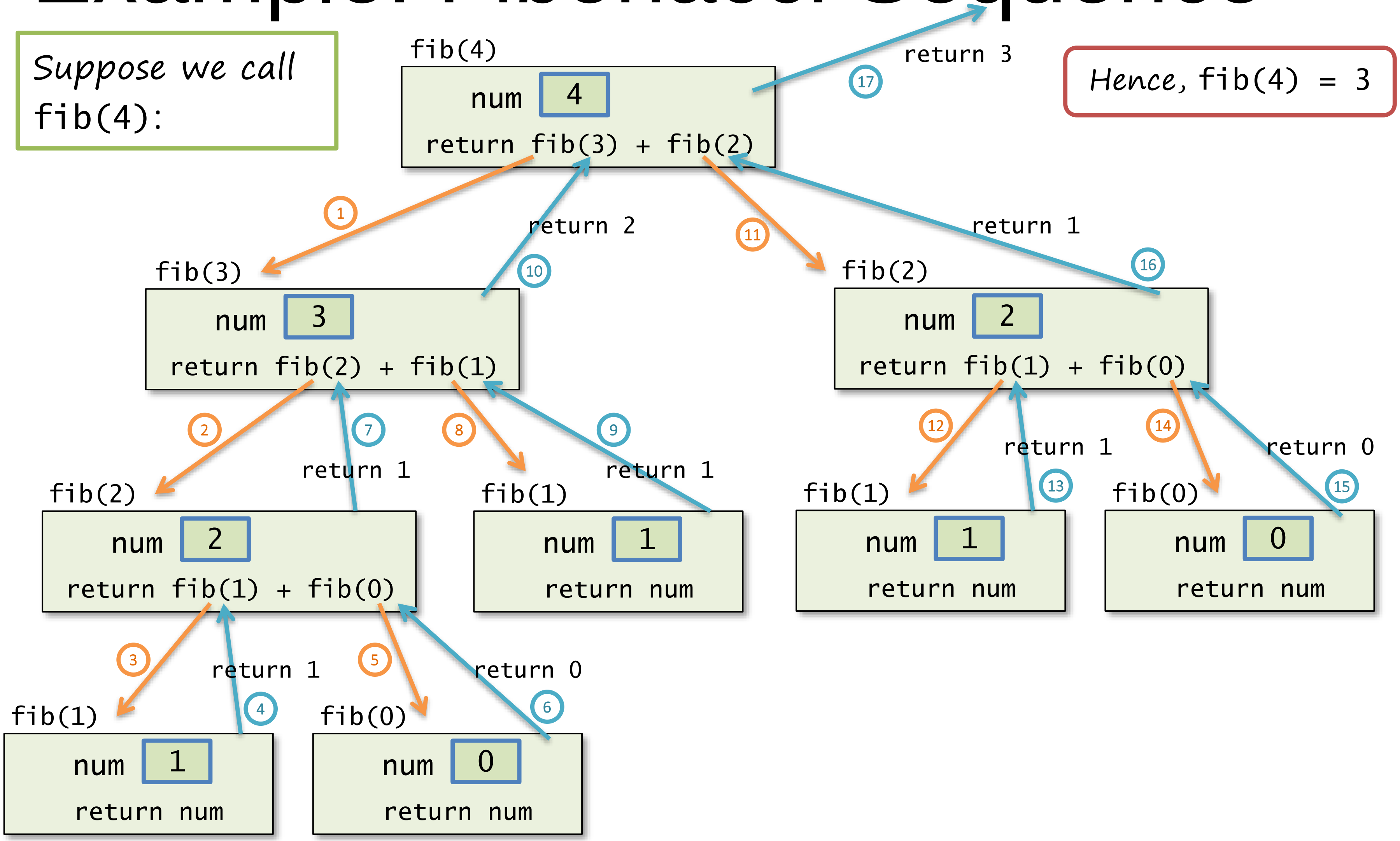
fibonacci.cpp

Base case without using recursion

Recursion to handle smaller tasks by making recursive calls

Example: Fibonacci Sequence

Suppose we call fib(4):



fib(4)

num 4

return fib(3) + fib(2)

17 return 3

1

10 return 2

11

16 return 1

fib(3)

num 3

return fib(2) + fib(1)

fib(2)

num 2

return fib(1) + fib(0)

2

7 return 1

8

9 return 1

12

13 return 1

14

15 return 0

fib(2)

num 2

return fib(1) + fib(0)

fib(1)

num 1

return num

fib(1)

num 1

return num

fib(0)

num 0

return num

3

4 return 1

5

6 return 0

fib(1)

num 1

return num

fib(0)

num 0

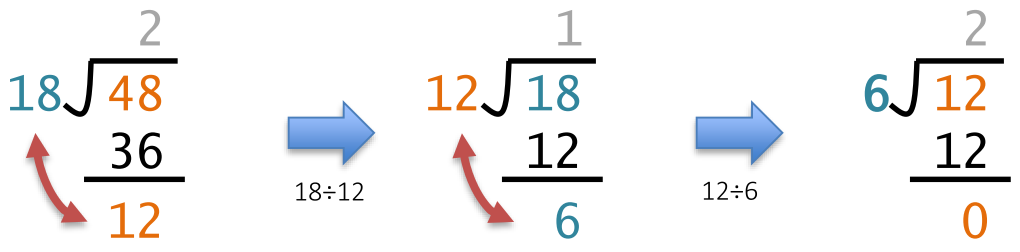
return num

Hence, fib(4) = 3

Example: Greatest Common Divisor

- Euclidean algorithm

E.g., gcd of 48 and 18:



gcd = 6

A recursive definition

$$\mathrm{gcd}(x, y) = \begin{cases} x, & \textit{if } y = 0 \\ \mathrm{gcd}(\, y, \text{remainder of } x \,/\, y\,), & \textit{otherwise} \end{cases}$$

```
int gcd(int x, int y)
{
    if (y == 0)
        return x;
    else
        return gcd(y, x%y);
}
```

gcd.cpp

Example: Palindrome

- Recall that a palindrome is a word that reads the same forward and backward, e.g., level, noon, racecar

r	a	c	e	c	a	r

Recursive algorithm

To check if a string `s[0..n-1]` is a palindrome,

1. if **n** < 2, **s** is a palindrome
2. otherwise, **s** is a palindrome if and only if
   `s[0]` is the same as `s[n-1]` and `s[1..n-2]` is a palindrome

```
bool is_palindrome( string s )
{
    if (s.length() < 2)
        return true;
    else
        return (s[0] == s[s.length()-1])
            && is_palindrome(s.substr(1,s.length()-2));
}
```

palindrome_recursive.cpp

Example: Tower of Hanoi

- The Tower of Hanoi is a mathematical game, consisting of three rods and disks of different sizes which can slide onto any rod.
- The puzzle starts with the disks neatly stacked in order of size on one rod, the smallest at the top, thus making a conical shape.
- Objectives: To move the entire stack to another rod.
- Rules:
  - Only one disk may be moved at a time
  - The removed disk must be placed on one of the rods
  - No disk may be placed on top of a smaller disk

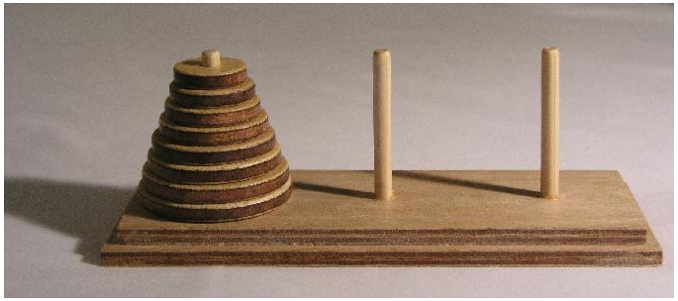
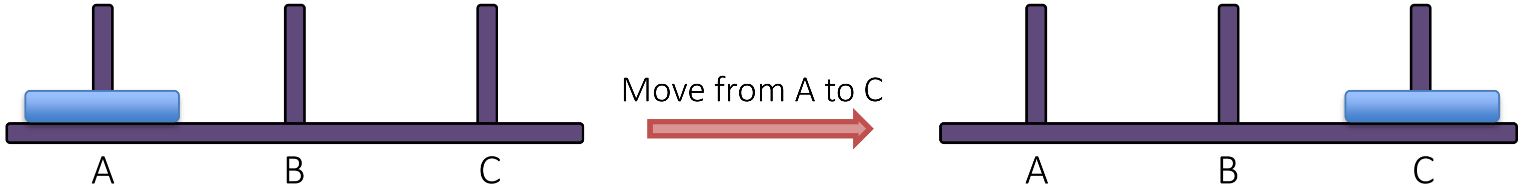


Image from Wikimedia Commons

Tower of Hanoi

Suppose the task is to move the stack from rod A to rod C

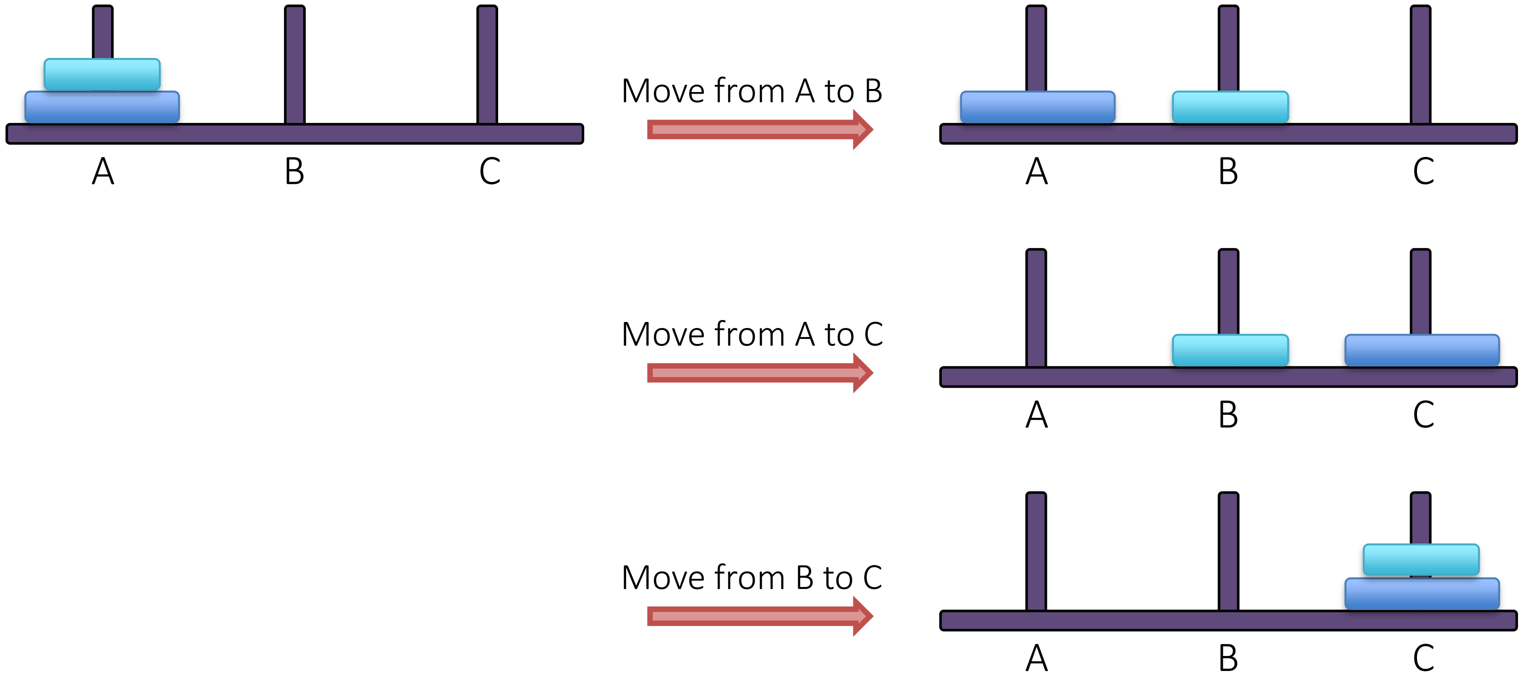
What if the initial stack contains 1 disk only?



Move from A to C

Done! 1 step

What if the initial stack contains 2 disks?



Move from A to B

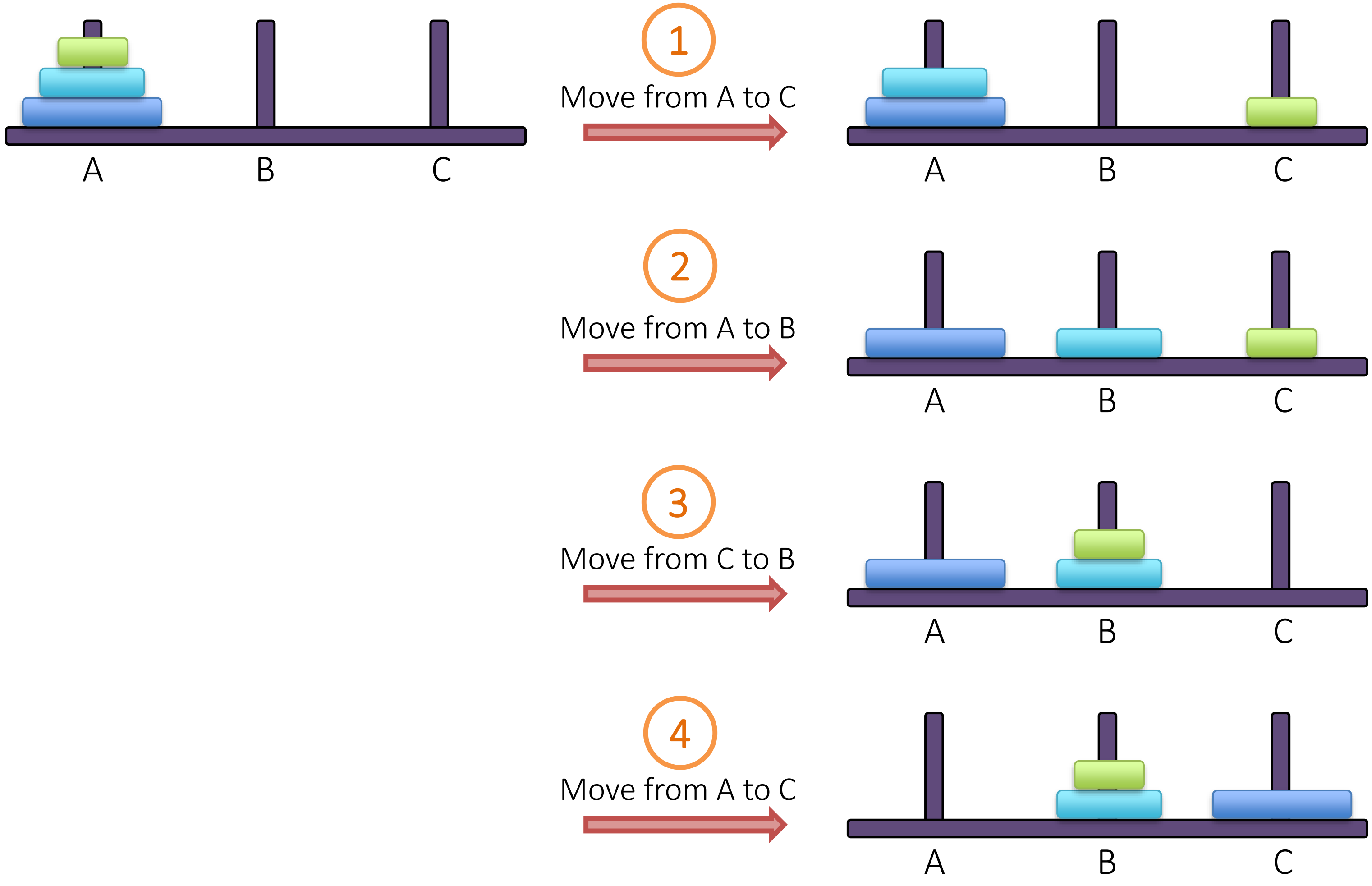
Move from A to C

Move from B to C

Done! 3 steps

Example: Tower of Hanoi

What if the initial stack contains 3 disks?



1

Move from A to C

2

Move from A to B

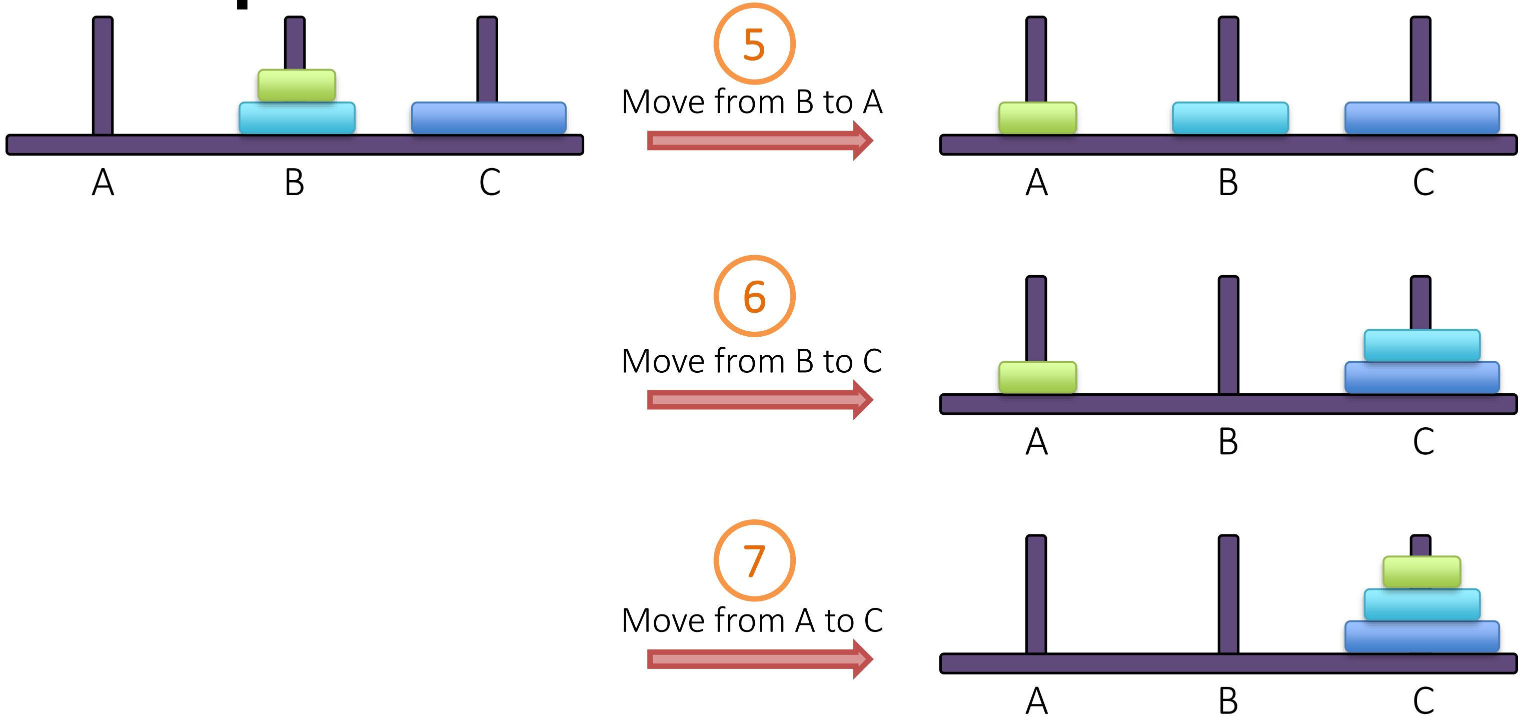
3

Move from C to B

4

Move from A to C

Example: Tower of Hanoi



5 Move from B to A

6 Move from B to C

7 Move from A to C

Done! 7 steps

What if the initial stack contains 64 disks???

Look at the example for moving 3 disks:
Steps 1 to 3 essentially move a stack of 2 disks from A to B
Step 4 moves a disk (the lowest of the initial stack) from A to C
Steps 5 to 7 essentially move a stack of 2 disks from B to C

A recursive algorithm!

Example: Tower of Hanoi

Recursive algorithm

To move a stack of n disks from rod A to rod C, n >= 1

1. Move the top n – 1 disks from A to B, using C as an intermediate rod
2. Move the remaining 1 disk from A to C
3. Move the top n – 1 disks from B to C, using A as an intermediate rod

No. of disks to move *Source rod* *Destination rod* *Intermediate rod*

```
void move(int n, char src, char des, char tmp)
{
   if (n == 1)
       cout << "Move disk from " << src << " to " << des << endl;

   else {
       move( n-1, src, tmp, des);

       move( 1, src, des, tmp);

       move( n-1, tmp, des, src);
   }
}
```

hanoi.cpp

Example: Tower of Hanoi

How many steps does it take to move 64 disks?

No. of steps to move n disks

$$
\begin{aligned}
T(n) &= 2\,T(n-1) + 1 \\
&= 2\,[\,2\,T(n-2) + 1\,] + 1 \\
&= 2^2\,T(n-2) + 2 + 1 \\
&= 2^2\,[\,2\,T(n-3) + 1\,] + 2 + 1 \\
&= 2^3\,T(n-3) + 2^2 + 2 + 1 \\
&= \ldots \\
&= 2^{n-1}\,T(n-(n-1)) + 2^{n-2} + \ldots + 2^2 + 2 + 1 \\
&= 2^{n-1}\,T(1) + 2^{n-2} + \ldots + 2^2 + 2 + 1 \\
&= 2^{n-1} + 2^{n-1} + \ldots + 2^2 + 2 + 1 \\
&= 2^n - 1
\end{aligned}
$$

Hence, it takes
$2^{64} - 1 \approx 1.6 \times 10^{19}$
steps to move 64 disks

If it takes 1 second to move a disk physically by hand, it would take 5×10^{11} years to finish.

If a computer can generate 10^9 moves per second, it still takes 500 years to generate all the moves!

Stack Overflow

- Each function call entails additional memory space (function call stack).
- There is always some limit to the memory size.
- If there is excessively long chain of recursive call, e.g., infinite recursion, stack overflow error may occur

Try the Tower of Hanoi program and see what's the largest *n* that will crash your machine ☺

Recursion vs. Iteration

- Recursion is NOT absolutely necessary.
- Any task that can be accomplished using recursion can also be done in some other way without using recursion.
- The non-recursive version of a function typically uses a loop of some sort in place of recursion, hence often being referred to as **iterative version**.
- A recursively written function will usually run slower and use more storage than an equivalent iterative version (due to extra work in memory management for function calls (aka stack management).
- Nonetheless, using recursion can sometimes make the job of programming easier and produce code that is easier to understand.

TUTORIALS

Did you find the concept of recursion too complicated? Don’t worry! Let’s have some tutorials.

Tutorial 1

SUM OF NATURAL NUMBERS

Sum of Natural Numbers

- Write a program that calculates the sum of the first **n** natural numbers, i.e., 1 + 2 + … + n.
- Create a new file and save it as **sum.cpp**
- Write a **main** function that
  - ask a user to input a positive integer **n**
  - call a function **sum(n)** to calculate the sum
  - output the result
- Write a **sum()** function (see also next slide) that
  - takes an integer n as input parameter
  - return the result of 1 + 2 + … + n

Sample output (user input in orange):

```
Enter a positive integer: 5
Sum of first 5 natural numbers = 15
```

sum_complete.cpp provides the completed version of this tutorial problem. You may compile and run it to see the expected results first.

Sum of Natural Numbers

- First version of **sum()** – iterative version
  - Write a **sum()** function so that it makes use of a loop to calculate the sum
  - Run and test your program
- Second version of **sum()** – recursive version
  - Write a **sum()** function which makes use of recursion to calculate the sum

$$\text{Idea: } \underbrace{1+2+\ldots+n}_{\text{This is sum(n)}} = (\underbrace{1+2+\ldots+n-1}_{\text{So what is this?}}) + n$$

  - What is the base case? What is the general case?
  - Run and test your program

Go to see Hints if you want the answer to these two questions

Hints

Note that these are only suggestions. You may come up with other solutions that work as well!

- *Sum of Natural Numbers*

```
sum = 1,                     if n = 1
sum(n) = sum(n–1) + n,       if n > 1
```

- *Largest Element in an Array*

```
largest(array[0..n–1]) = –1,                                           if n < 1
largest(array[0..n–1]) = max(largest(array[0..n–2]), array[n–1]) ,    otherwise
```

- *Reversing a String*

```
reverse(s[0..n–1]) = s,                                 if length of s = 0,
reverse(s[0..n–1]) = s[n–1] + reverse(s[0..n–2]),      otherwise
```

Tutorial 2

LARGEST ELEMENT IN AN ARRAY

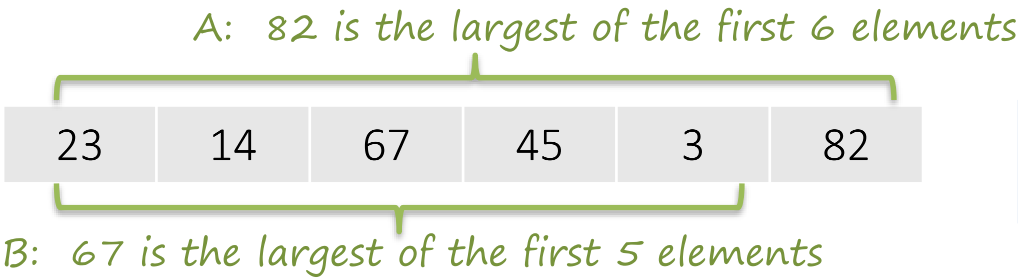
Largest Element in an Array

- Write a program to find the largest element in an array
- Open **largest_element_incomplete.cpp**
- Study the **main** function. It
  - generates a set of random positive numbers in an array
  - outputs the numbers to the screen
  - determines the largest element in the array by calling **largest()**
  - outputs the largest element

largest_element.cpp provides the complete version of this tutorial problem.

Largest Element in an Array

- Write the **largest_element()** function that uses a loop to determine the largest element in an array
  - First determine the function prototype. Look at how it is called in **main()**. What should be the input parameters? What should be the return value?
  - Finish the function body. Compile and run the program.
- Write the **largest_element()** function that uses recursion to determine the largest element in an array



A: 82 is the largest of the first 6 elements

23	14	67	45	3	82

B: 67 is the largest of the first 5 elements

How to determine A using the results of B?

  - What is the base case? What is the general case?

Go to see Hints if you want the answer to these two questions

Tutorial 3

REVERSING A STRING

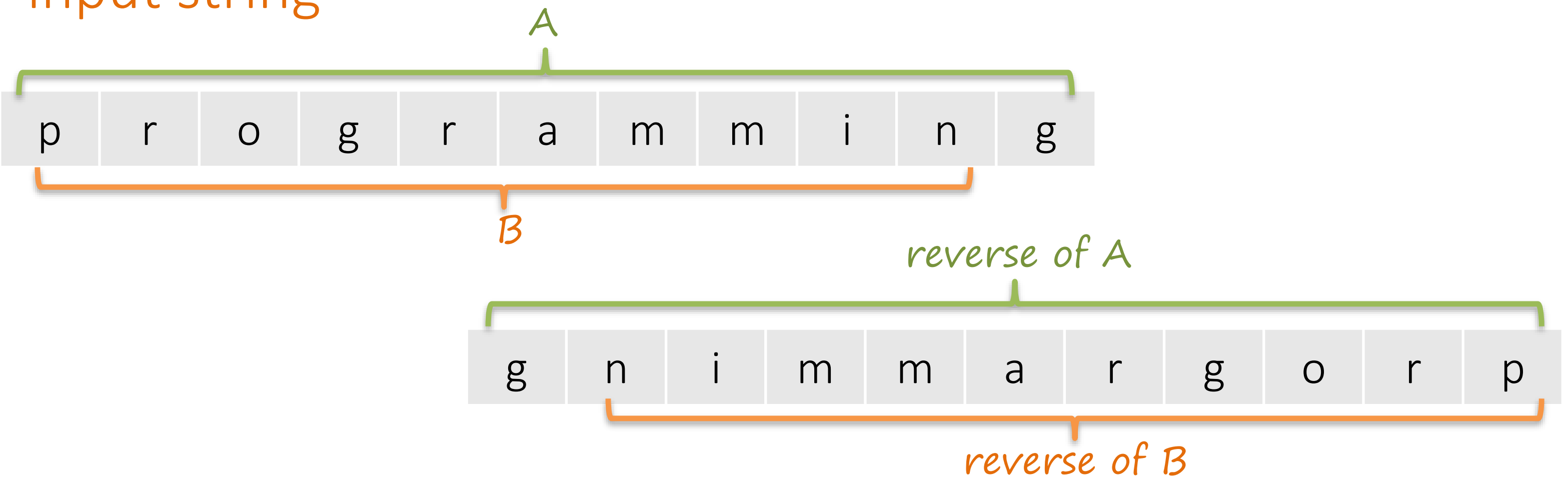
Reversing a String

- Write a program to reverse an input string.
- Open **string_reverse_incomplete.cpp**
- Study the **main** function. It
  - asks the user to input a string
  - reverse the string by calling **reverse()**
  - print out the reversed string
- Write the **reverse()** function that uses a loop to reverse an input string
  - First determine the function prototype. Look at how it is called in **main()**. What should be the input parameters? What should be the return value?
  - Finish the function body. Compile and run the program.

string_reverse.cpp provides the complete version of this tutorial problem.

Reversing a String

- Write the **reverse()** function that uses recursion to reverse an input string



p	r	o	g	r	a	m	m	i	n	g

g	n	i	m	m	a	r	g	o	r	p

- What is the base case?
  What is the general case?

How is reverse of A and reverse of B related?

Go to see Hints if you want the answer to these two questions

We are happy to help you!

“If you face any problems in understanding the materials, **please feel free to contact me, our TAs or student TAs**.
We are very happy to help you!
We wish you enjoy learning programming in this class ☺.”