

2800ICT

Object Oriented Programming

Basic C++ Programming

Object Oriented Programming

C++

- Pre-requisite course
 - 1811ICT Programming Principles
 - Or 1305ENG Engineering Programming
 - Or 1013ICT Mathematics for Computer Science
 - Or 1806ICT Programming Fundamentals
- Students not meeting prerequisites must withdraw from this course ASAP in Week 1. Otherwise, the enrolment system will withdraw you from this course.
- If you don't know which alternative course to enrol in, please ask your Program Director for advice.

Classes

- Lectures (Monday 12:00 - 13:50)
 - 2-hour online lecture every week
- Common Time (Thursday 10:00 - 11:50)
 - Feedback on your last week assignment
 - In-person/on-line assistance with any problem you may have
- Labs/Workshops
 - 2-hour online/in-person workshop every week.
 - Problem set contains a few programming questions

Assessment

Assessment	Title	Weight	Marked out of	Due Date
1	Weekly Assignments	30%	30 marks	Each Week
2	Milestone	30%	30 marks	Week 11
3	Final Examination	40%	40 marks	Exam Week

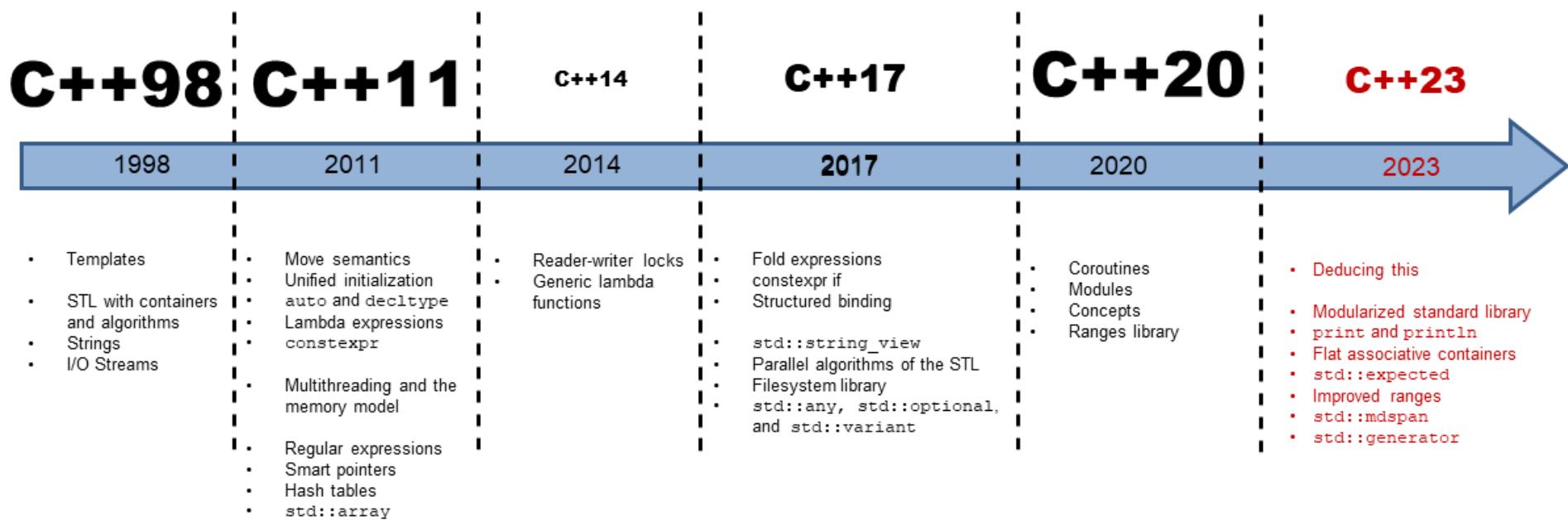
- Weekly Assignment (3 marks each week)
- Milestone has 1 programming question
- Final Exam has 3 programming questions, performed on-line in exam week. (not ProctorU)



Topics

- C++ History
- C++ Basic Input/Output & C++ Building Process
- Basic C++ Programming
- Three commonly-used data types:
 - std::string, std::array, std::vector
- C++ Function
 - Definition, Overloading, Templates, Default Arguments
- Reference variables

C++ History



C++ History

Q: What is C++ good at?

C++ excels in situations where high performance and precise control over memory and other resources is needed. Here are a few common types of applications that most likely would be written in C++:

- Video games
- Real-time systems (e.g. for transportation, manufacturing, etc...)
- High-performance financial applications (e.g. high frequency trading)
- Graphical applications and simulations
- Productivity / office applications
- Embedded software
- Audio and video processing
- Artificial intelligence and neural networks

TIOBE Programming Index for February 2024

Programming Language	2024	2019	2014	2009	2004	1999	1994	1989
Python	1	4	8	6	11	27	22	-
C	2	2	1	2	2	1	1	1
C++	3	3	4	3	3	2	2	3
Java	4	1	2	1	1	13	-	-

About C & C++

- C is a Procedural Orientated language, whereas C++ is an Object-Oriented Programming language.
 - Enhanced Type Safety and Abstraction
 - Standard Template Library (STL)
- C++ is a **superset** of C and uses a different set of include files.
- C supports built-in data types whereas C++ supports built-in as well as user-defined data types.

Topics

- C++ History
- C++ Basic Input/Output & C++ Building Process
- Basic C++ Programming
- Three commonly-used data types:
 - std::string, std::array, std::vector
- C++ Function
 - Definition, Overloading, Templates, Default Arguments
- Reference variables

C++ Development Environment

- Recommended: setup vscode on your PC
 - Powerful functions/features
 - Setup vscode C++ on Win11.pdf (refer to L@G)
 - Setup vscode C++ on macOS.pdf (refer to L@G)
- Griffith ELF
 - Easy to setup, but no debugging function.
 - Setup vscode C++ on Griffith ELF.pdf (refer to L@G)

Tips

- Folder Name
 - Avoid using spaces in folder names
- Quite Commonly used Terminal cmd

Function	Windows 11 (cmd / PowerShell)	macOS (Zsh / Bash)
List files in the current directory	dir	ls
Change directory	cd <folder>	cd <folder>
Print current directory	cd	pwd

- Shortkey for vscode
 - Ctrl+/: Toggle line comment
 - Shift+Alt+F : Format the document

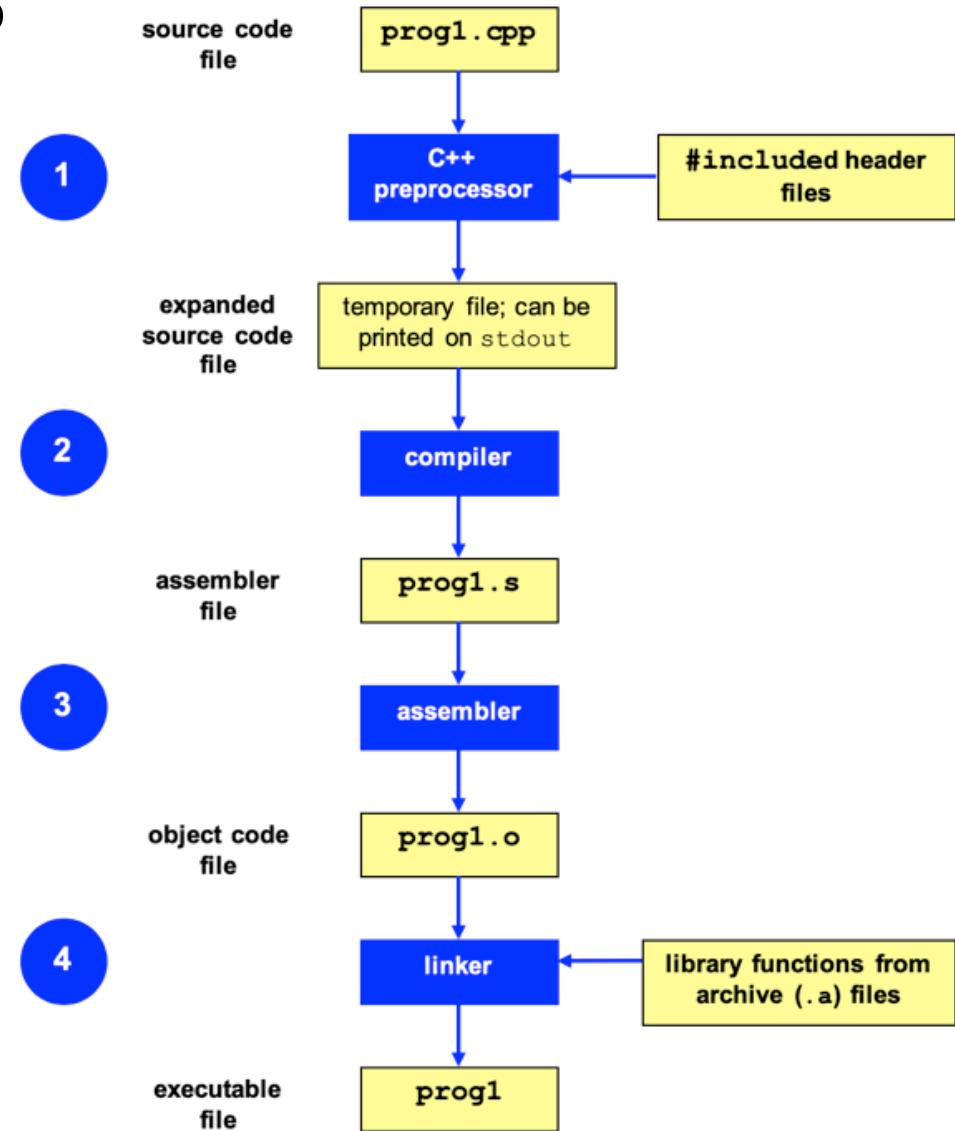
C++ Basic Input / Output

```
#include <iostream>
/* This program inputs two numbers x and y and outputs their sum */
int main()
{
    int x{}, y{1};
    std::cout << "Please enter two numbers: ";
    std::cin >> x >> y;
    int sum = x + y;
    std::cout << x << " + " << y << " = " << sum << std::endl;
    return 0;
}
```

basic_io_1.cpp

C++ Build Process

```
g++ -std=c++20 -o prog1 prog1.cpp
```



C++ Basic Input / Output

```
#include <iostream>
using namespace std;
/* This program inputs two numbers x and y and outputs their
sum */
int main()
{
    int x, y;
    cout << "Please enter two numbers: ";
    cin >> x >> y;
    int sum = x + y;
    cout << x << " + " << y << " = " << sum << endl;
    return 0;
}
```

basic_io_2.cpp

C++ Variable Definition

- Variable definition defines **a named storage location** (variable) in memory, specifying its **data type** and optionally initializing it with a value.

```
int x{}, y{1};
```

- In C++, you **must** define/declare a variable before using it.

C++ - auto

- By using auto, we let the compiler deduce the type of the sum variable
 - which makes the code a bit cleaner and easier to maintain.

```
#include <iostream>
using namespace std;
/* This program inputs two numbers x and y and outputs their sum using auto */
int main()
{
    int x, y;
    cout << "Please enter two numbers: ";
    cin >> x >> y;
    auto sum = x + y; // The compiler deduces the type of sum based on x and y
    cout << x << " + " << y << " = " << sum << endl;
    return 0;
}
```

C++11 – decltype

The term **decltype** is a keyword used to query the type of an expression.

Examples of **decltype**:

```
int a = 1; // `a` is declared as type `int`
decltype(a) b = a; // `decltype(a)` is `int`

decltype(123) e = 123; // `decltype(123)` is `int`
```

```
// decltype_demo.cpp

#include <iostream>

// Generic multiply function using decltype to deduce the return type
template <typename T, typename U>
auto multiply(T t, U u) -> decltype(t * u) {
    return t * u;
}

int main() {
    int a = 5;
    double b = 2.5;

    // The compiler deduces that the result is of type double
    auto result = multiply(a, b);

    std::cout << a << " * " << b << " = " << result << std::endl;
    return 0;
}
```

C++ Basic Input / Output

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << setw(3) << 6 << endl
        << setw(3) << 18 << endl
        << setw(3) << 124 << endl
        << "----\n"
        << (6 + 18 + 124) << endl;
    return 0;
}
```

The output of this program is:

```
6
18
124
----
148
```

basic_io_3.cpp

C++ Basic Input / Output

Manipulator	Action
<code>setw(n)</code>	(not sticky) Sets the field width for the next output item to n characters. If the output is shorter than n, it is padded with the fill character (default: space).
<code>setfill('x')</code>	Sets the fill character (c) for padding when <code>setw()</code> is used.
<code>showpoint / noshowpoint</code>	Forces floating-point numbers to display a decimal point and trailing zeros (even if unnecessary).
<code>fixed / scientific / defaultfloat</code>	Forces fixed-point notation (e.g., 3.14). Uses scientific notation (e.g., 3.141593e+00). Resets to the compiler's default floating-point format.
<code>setprecision(n)</code>	If the fixed manipulator is designated, n specifies the total number of displayed digits after the decimal point; otherwise, n specifies the total number of significant digits displayed (integer plus fractional digits).
<code>left / right</code>	Sets text alignment within the field width
<code>boolalpha</code>	Displays boolean values as "true"/"false" instead of 1/0.

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

int main() {
    cout << left << setfill('-'); // Left-align, fill with '-'
    cout << setw(10) << "Name" << setw(10) << "Price" << setw(10) << "Stock" << endl;
    cout << setw(10) << "Apple"
        << fixed << setprecision(2) << setw(10) << 2.5
        << boolalpha << setw(10) << true << endl;
    return 0;
}
```

Name-----Price-----Stock-----
Apple-----2.50-----true-----

Topics

- C++ History
- C++ Basic Input/Output & C++ Building Process
- Basic C++ Programming
- Three commonly-used data types:
 - std::string, std::array, std::vector
- C++ Function
 - Definition, Overloading, Templates, Default Arguments
- Reference variables

Basic C++ Programming

- Data types
- Arithmetic operators
- Logical operators
- Control Flow
 - **if-else**
 - switch
 - **while loop**
 - **for loop**

Fundamental data types

Types	Category	Meaning	Example
float double	Floating Point	a number with a fractional part	3.14159
long double			
bool	Integral (Boolean)	true or false	true
char wchar_t char8_t (C++20) char16_t (C++11) char32_t (C++11)	Integral (Character)	a single character of text	'c'
short int int	Integral (Integer)	positive and negative whole numbers, including 0	64
long int			
long long int (C++11)			
std::nullptr_t (C++11)	Null Pointer	a null pointer	nullptr
void	Void	no type	n/a

```
#include <iostream>

int main()
{
    float a = 3.14159f;           // 'f' suffix ensures it is a float
    double b = 3.1415926535;     // Default floating type in C++
    std::cout << "Float: " << a << "\n";
    std::cout << "Double: " << b << "\n";

    return 0;
}
```

arithmetic operators

Operator	Symbol	Form	Operation
Addition	+	$x + y$	x plus y
Subtraction	-	$x - y$	x minus y
Multiplication	*	$x * y$	x multiplied by y
Division	/	x / y	x divided by y
Remainder	%	$x \% y$	The remainder of x divided by y

Operator	Symbol	Form	Operation
Assignment	=	$x = y$	Assign value y to x
Addition assignment	$+=$	$x += y$	Add y to x
Subtraction assignment	$-=$	$x -= y$	Subtract y from x
Multiplication assignment	$*=$	$x *= y$	Multiply x by y
Division assignment	$/=$	$x /= y$	Divide x by y
Remainder assignment	$\%=$	$x \%= y$	Put the remainder of x / y in x

Logical operators

Operator	Symbol	Example Usage	Operation
Logical NOT	!	<code>!x</code>	true if x is false, or false if x is true
Logical AND	<code>&&</code>	<code>x && y</code>	true if both x and y are true, false otherwise
Logical OR	<code> </code>	<code>x y</code>	true if either x or y are true, false otherwise

x	y	<code>!x</code>	<code>!y</code>	<code>!(x && y)</code>	<code>!x !y</code>
false	false	true	true	true	true
false	true	true	false	true	true
true	false	false	true	true	true
true	true	false	false	false	false

Logical operators

Operator	Symbol	Example Usage	Operation
Logical NOT	!	<code>!x</code>	true if x is false, or false if x is true
Logical AND	<code>&&</code>	<code>x && y</code>	true if both x and y are true, false otherwise
Logical OR	<code> </code>	<code>x y</code>	true if either x or y are true, false otherwise

Operator name	Keyword alternate name
<code>&&</code>	<code>and</code>
<code> </code>	<code>or</code>
<code>!</code>	<code>not</code>

This means the following are identical:

```
1 | std::cout << !a && (b || c);  
2 | std::cout << not a and (b or c);
```

Control Flow – if-else

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";
    int x{}; //declare and initialize to 0
    std::cin >> x;

    if (x >= 10)
    {
        std::cout << x << " is greater than 10\n";
    }
    else if (x > 5)
    {
        std::cout << x << " is greater than 5 and less than 10\n";
    }
    else
    {
        std::cout << x << " is not greater than 5\n";
    }

    return 0;
}
```

Control Flow – switch

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

int main()
{
    int day = 4;
    switch (day)
    {
        case 1:
            cout << "Monday";
            break;
        case 2:
            cout << "Tuesday";
            break;
        case 3:
            cout << "Wednesday";
            break;
        case 4:
            cout << "Thursday";
            break;
        case 5:
            cout << "Friday";
            break;
        case 6:
            cout << "Saturday";
            break;
        case 7:
            cout << "Sunday";
            break;
    }
    return 0;
}
```

Control Flow – while-loop

```
#include <iostream>

int main()
{
    char c{'y'};
    while (c == 'y') // infinite loop
    {
        std::cout << "Loop again (y/n)? ";
        std::cin >> c;
    }

    return 0;
}
```

Control Flow – traditional for-loop

```
#include <iostream>
int main()
{
    for (int i{1}; i <= 10; ++i)
    {
        std::cout << i << ' ';
    }
    std::cout << '\n';
    return 0;
}
```

The `for statement` looks pretty simple in abstract:

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

Control Flow – range-based for-loop

```
for (dataType rangeVariable : array)
{
    statement;
}
```

- **dataType** is the data type of the range variable.
- **rangeVariable** is the name of the range variable. This variable will receive the value of a different array element during each loop iteration.
- **array** is the name of an array on which you wish the loop to operate.
- **statement** is a statement that executes during a loop iteration. If you need to execute more than one statement in the loop, enclose the statements in a set of braces.

```
#include <iostream>
using namespace std;
int main()
{
    // Define an array of integers.
    int numbers[] = {10, 20, 30, 40, 50};
    // Display the values in the array.
    for (auto val : numbers)
        cout << val << endl;
    return 0;
}
```

for_range1.cpp

Topics

- C++ History
- C++ Basic Input/Output & C++ Building Process
- Basic C++ Programming
- Three commonly-used data types:
 - std::string, std::array, std::vector
- C++ Function
 - Definition, Overloading, Templates, Default Arguments
- Reference variables

string

- string: sequence of characters stored in adjacent memory
- Defined in the **<string>** header
 - To use std::string, include the <string> header

C++ string class – build string

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string str1; // an empty string
    string str2{"Good Morning"};
    string str3 = "Hot Dog";
    string str4{str3};
    string str5(str4, 4);
    string str6 = "linear";
    string str7(str6, 3, 3);
    cout << "str1 is: " << str1 << endl;
    cout << "str2 is: " << str2 << endl;
    cout << "str3 is: " << str3 << endl;
    cout << "str4 is: " << str4 << endl;
    cout << "str5 is: " << str5 << endl;
    cout << "str6 is: " << str6 << endl;
    cout << "str7 is: " << str7 << endl;
    return 0;
}
```

```
str1 is:
str2 is: Good Morning
str3 is: Hot Dog
str4 is: Hot Dog
str5 is: Dog
str6 is: linear
str7 is: ear
```

string1.cpp

C++ string class – string processing

Function/Operation	Description	Example
at(ind)	provides safe access to an individual character at index of ind	s1.at(5)
int length()	Returns the length of the string	s1.length()
int compare(str)	Compares the given string to str; returns a negative value if the given string is less than str, a 0 if they are equal, and a positive value if the given string is greater than str	s1.compare(s2)
int find(str)	Returns the index of the first occurrence of str in the complete string	s1.find("the")
void replace (ind, n, str)	Removes n characters in the string object, starting at index position ind , and inserts the string str at index position ind	s1.replace(2, 4, "okay")

C++ string class – string processing

Function/Operation	Description	Example
string substr(ind, n)	Returns a string consisting of n characters extracted from the string, starting at index ind; if n is greater than the remaining number of characters, the rest of the string is used	s2 = s1.substr(0,10)
+	Concatenates two strings	s1 + s2
to_string(v)	Converts v to a string	s = to_string(28.23);
stof<type>(s)	Converts s to a float value	float f1 = stof("12.34");

C++ string class – read strings

```
#include <iostream>
#include <string>
#include <limits>
#include <sstream>
using namespace std;
int main()
{
    int value;
    string message;
    cout << "Enter text: ";
    cin >> message; // Stops on first blank
    // cin.clear() // reset any error flags
    // cin.ignore(100, '\n');
    // to ignore up to 100 characters, or until the newline is reached.
    cout << "The text entered is: " << message << endl;

    cout << "Enter a number: ";
    cin >> value; // Leaves \n in buffer
    // cin.clear();
    // cin.ignore(numeric_limits<streamsize>::max(), '\n');
    // to ignore any characters in the input buffer until we find an enter
    cout << "The number entered is: " << value << endl;

    cout << "Enter text: ";
    getline(cin, message); // Reads all, including blanks and the \n
    cout << "The text entered is:" << message << endl;
    cout << int(message.length()) << endl;
    return 0;
}
```

Tips:

- Don't mix `cin` with `getline()` inputs in the same program.
- Always follow the `cin` input with the `cin.ignore()`
 - `cin.ignore(n, '\n')` : to ignore up to **n** characters, or until the **newline '\n'** is reached.

string2.cpp

C++11 – std::array

std::array is a fixed-size sequence container introduced in C++11 as part of the Standard Template Library (STL).

Key Features

- **Fixed Size:** Size is specified at compile time (cannot change at runtime).
- **Contiguous Storage:** Elements are stored in contiguous memory.
- **STL Compatibility:** Supports iterators, algorithms (e.g., std::sort), and member functions.
- **Bounds Checking:** Safe access via **at()**.

Function	Description
size()	Returns the number of elements.
at(index)	Access element with bounds checking.

C++11 – std::array definition

std::array is a fixed-size sequence container introduced in C++11 as part of the Standard Template Library (STL).

```
#include <array> // for std::array
#include <iostream>

int main()
{
    std::array<int, 5> a;
// Members default initialized (int elements are left uninitialized)
    std::array<int, 5> b{};
// Members value initialized (int elements are zero initialized) (preferred)

    std::array<int, 5> prime { 2, 3, 5, 7, 11 };
// list initialization using braced list (preferred)

// Subscripting std::array using operator[], at(), std::get()
    std::cout << prime[1] << prime.at(1) << std::get<1>(prime) << '\n';

    return 0;
}
```

STL vector

- A data type defined in the Standard Template Library
- Can hold values of any type:
`vector<int> scores;`
- Automatically adds space as more is needed
 - no need to determine size at definition

Declaring Vectors

- You must `#include<vector>`
- Declare a vector to hold `int` element:
`vector<int> scores;`
- Declare a vector with initial size 30:
`vector<int> scores(30);`
- Declare a vector and initialize **all elements to 0**:
`vector<int> scores(30, 0);`
- Declare a vector initialized to size and contents of another vector:
`vector<int> finals(scores);`

Vector - Element Access

- `vec[index]` Fast access (no bounds checking).
- `vec.at(index)` Safer: Bounds-checked access.
- `vec.front()` First element.
- `vec.back()` Last element.

Vector - Common Member Functions

Function	Description
push_back(value)	Add an element to the end.
pop_back()	Remove the last element.
size()	Return the number of elements.
capacity()	Return the current allocated storage capacity.
empty()	Return true if the vector is empty.
clear()	Remove all elements.
insert(iterator, value)	Insert an element at a specific position.
erase(iterator)	Remove an element at a specific position.
resize(n)	Change the number of elements to n.
reserve(n)	Preallocate memory for n elements (optimization).
data()	Get a raw pointer to the underlying array.

```
// vector_basic.cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {10, 20, 30};

    // Add elements
    numbers.push_back(40);           // [10, 20, 30, 40]
    numbers.pop_back();             // [10, 20, 30]

    // Access elements
    std::cout << numbers[1];        // 20
    std::cout << numbers.at(2);     // 30

    // Iterate using range-based loop
    for (int num : numbers) {
        std::cout << num << " "; // 10 20 30
    }

    // Resize and reserve
    numbers.reserve(100);           // Preallocate memory
    numbers.resize(5);              // [10, 20, 30, 0, 0]

    return 0;
}
```

Topics

- C++ History
- C++ Basic Input/Output & C++ Building Process
- Basic C++ Programming
- Three commonly-used data types:
 - std::string, std::array, std::vector
- C++ Function
 - Definition, Overloading, Templates, Default Arguments
- Reference variables

C++ Function Definition

- Functions improve modularity, readability, and code reusability.
 - A block of code that perform specific tasks

```
returnType functionName(parameter1Type parameter1, parameter2Type parameter2, ...)  
{  
    // Code to execute  
    return value; // Optional (only if returnType is not void)  
}
```

```
// function_demo.cpp  
#include <iostream>  
using namespace std;  
  
// Function to add two integers  
int add(int a, int b) {  
    return a + b;  
}  
  
int main() {  
    int result = add(3, 5); // Call the function  
    cout << "Sum: " << result; // Output: Sum: 8  
    return 0;  
}
```

C++ Function Definition

- Scenario
 - Imagine you're developing an arithmetic library that must support operations like addition for integers, float, and double numbers.
- What will you do?

Solution I - C++ Function Overloading

- C++ provides the capability of using the **same function name** for more than one function, referred to as function overloading.
- The compiler will determine which function to use based on **the parameters' data types**.

Overloading Method	Example
Different Parameter Types	void func(int); vs. void func(double);
Different Number of Parameters	void func(); vs. void func(int, double);
Different Parameter Order	void func(int, double); vs. void func(double, int);

```
//function_overloading.cpp
#include <iostream>

int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

double add(int ai, double a, double b) {
    return ai + a + b;
}

int main() {
    int intResult = add(3, 5);
    double doubleResult = add(3.2, 4.4);
    double result = add(1,1.0,1.5);

    std::cout << "Sum of integers: " << intResult << std::endl;
    std::cout << "Sum of doubles: " << doubleResult << std::endl;
    std::cout << "Sum of mix: " << result << std::endl;

    return 0;
}
```

Solution II - C++ Function Templates

- Function template: a pattern for a function that can work with many data types

```
template <typename T> // T is a placeholder for any type
ReturnType functionName(T parameter1, T parameter2, ...) {
    // Code using type T
}
```

```
// function_template.cpp
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    int intSum = add(10, 20);

    double doubleSum = add(10.5, 20.3);

    std::cout << "Integer Sum: " << intSum << std::endl;
    std::cout << "Double Sum: " << doubleSum << std::endl;

    return 0;
}
```

Function Template

- Question
 - a function template that adds two numbers of different types, such as an int and a float.

```
// function_template_different_type.cpp
#include <iostream>

// Function template that adds two numbers of potentially different types
template <typename T, typename U>
auto add(T a, U b) {
    return a + b;
}

int main() {
    int i = 5;
    float f = 3.5f;

    auto result = add(i, f); // Compiler deduces the return type based on a + b

    std::cout << "Sum of int and float: " << result << std::endl;
    return 0;
}
```

Default Arguments

A Default argument is an argument that is passed automatically to a parameter if the argument is missing on the function call.

- Must be a constant declared in prototype:
`void evenOrOdd(int = 0);`
- Can be declared in header if no prototype
- Multi-parameter functions may have default arguments for some or all of them:
`int getSum(int, int=0, int=0);`

Default Arguments

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

// Function prototype with default arguments
void displayStars(int cols = 10, int rows = 1);

int main()
{
    displayStars(); // Use default values for cols and rows.
    cout << endl;
    displayStars(5); // Use default value for rows.
    cout << endl;
    displayStars(7, 3); // Use 7 for cols and 3 for rows.
    return 0;
}

void displayStars(int cols, int rows)
{
    // Nested loop. The outer loop controls the rows
    // and the inner loop controls the columns.
    for (int down = 0; down < rows; down++)
    {
        for (int across = 0; across < cols; across++)
        {
            cout << "*";
        }
        cout << endl;
    }
}
```



displaystars.cpp

Default Arguments

- If not all parameters to a function have default values, the **defaultless** ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0); // OK  
int getSum(int, int=0, int); // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
sum = getSum(num1, num2); // OK  
sum = getSum(num1, , num3); // NO
```

C++11 – initializer lists

A lightweight array-like container of elements created using a "braced list" syntax.

For example, `{1, 2, 3}` creates a sequences of integers, that has type `std::initializer_list<int>`. Useful as a replacement to passing a vector of objects to a function.

```
// function_init_list.cpp
#include <iostream>
#include <initializer_list>

void printNumbers(std::initializer_list<int> numbers) {
    for (auto num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

int main() {
    // Calling the function with an initializer list
    printNumbers({1, 2, 3, 4, 5});
    return 0;
}
```

Topics

- C++ History
- C++ Basic Input/Output & C++ Building Process
- Basic C++ Programming
- Three commonly-used data types:
 - std::string, std::array, std::vector
- C++ Function
 - Definition, Overloading, Templates, Default Arguments
- Reference variables

Reference Variables

- A reference variable is an alias, that is, another name for an existing variable.

```
dataType &referenceName = originalVariable;
```

- **sharing the same memory address**
 - allows you to manipulate the original variable indirectly

- Key Characteristics

- Must be initialized

```
int x = 10;  
int &ref = x; // Valid: ref is an alias for x  
int &invalidRef; // Error: reference must be initialized
```

- Cannot be reassigned

```
int a = 5, b = 20;  
int &ref = a;  
ref = b; // Assigns b's value to a (a = 20)
```

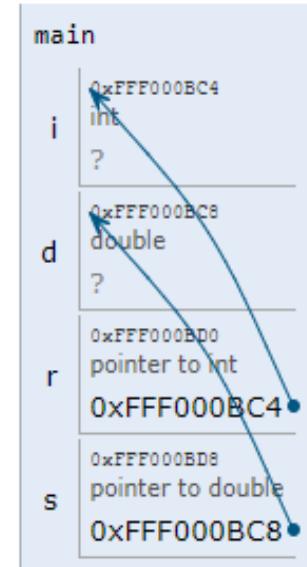
Reference Variables

Stack

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    double d;
    int &r = i;
    double &s = d;
    i = 5;
    cout << "Value of i : " << i << endl;
    cout << "Value of i reference : " << r << endl;

    r = 7;
    cout << "Value of i : " << i << endl;
    cout << "Value of i reference : " << r << endl;

    d = 11.7;
    cout << "Value of d : " << d << endl;
    cout << "Value of d reference : " << s << endl;
    return 0;
}
```



Value of i : 5
Value of i reference : 5
Value of i : 7
Value of i reference : 7
Value of d : 11.7
Value of d reference : 11.7

Using Reference Variables as Parameters

- A mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to ‘return’ more than one value

Using Reference Variables as Parameters

- A reference variable is an alias for another variable
- Defined with an ampersand (**&**)
`void getDimensions(int&, int&);`
- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

Using Reference Variables as Parameters

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

// Function prototype. The parameter is a reference variable.
void doubleNum(int &);

int main()                                Must use & in both prototype and definition
{
    int value = 4;

    cout << "In main, value is " << value << endl;
    cout << "Now calling doubleNum..." << endl;
    doubleNum(value);
    cout << "Now back in main. value is " << value << endl;
    return 0;
}

// Function definition
void doubleNum(int &refVar)
{
    refVar *= 2;
}
```

reference2.cpp

Modifying an array with a Range-based for Loop

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

int main()
{
    const int SIZE = 5;
    int numbers[5];

    // Get values for the array.
    for (int &val : numbers)
    {
        cout << "Enter an integer value: ";
        cin >> val;
    }

    // Display the values in the array.
    cout << "Here are the values you entered:\n";
    for (int val : numbers)
        cout << val << endl;

    return 0;
}
```

for_range2.cpp

Modifying an array with a Range-based for Loop

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

int main()
{
    const int SIZE = 5;
    int numbers[5];

    // Get values for the array.
    for (int &val : numbers)
    {
        cout << "Enter an integer value: ";
        cin >> val;
    }

    // Display the values in the array.
    cout << "Here are the values you entered:\n";
    for (int val : numbers)
        cout << val << endl;

    return 0;
}
```

Modern C++:

Any improvement on data type?

for_range2.cpp

C++ Include Files

Library	Files	Function
Utilities	<cstdlib>	General purpose utilities: program control, dynamic memory allocation, random numbers, sort and search
	<ctime>	C-style time/date utilites
Numeric limits	<climits>	Limits of integral types
	<cfloat>	Limits of floating-point types
	<limits>	Uniform way to query properties of arithmetic types
Input/output	<iostream>	Several standard stream objects
	<iomanip>	Helper functions to control the format of input and output
	<fstream>	std::basic_fstream, std::basic_ifstream, std::basic_ofstream class templates and several typedefs
	<sstream>	std::basic_stringstream, std::basic_istringstream, std::basic_ostringstream class templates and several typedefs
Error handling	<exception>	Exception handling utilities

C++ Include Files

Library	Files	Function
Strings	<cctype>	Functions to determine the category of narrow characters
	<cstring>	Various narrow character string handling functions
	<string>	std::basic_string class template
Containers	<vector>	std::vector container
	<deque>	std::deque container
	<list>	std::list container
	<set>	std::set and std::multiset associative containers
	<map>	std::map and std::multimap associative containers
	<stack>	std::stack container adaptor
	<queue>	std::queue and std::priority_queue container adaptors
	<iterator>	Range iterators
Algorithms	<algorithm>	Algorithms that operate on ranges
Numerics	<cmath>	Common mathematics functions
	<complex>	Complex number type

Online C++ Help

- <https://devdocs.io/cpp/>
- <https://cplusplus.com/reference/>

2800ICT

Object Oriented Programming

STL – Container, Iterator

Review Week1

- C++ History
- C++ Basic Input/Output & C++ Building Process
- Basic C++ Programming
- Three commonly-used data types:
 - std::string, std::array, std::vector
- C++ Function
 - Definition, Overloading, Templates, Default Arguments
- Reference variables

Topics

- Introduction to STL
- STL Container Fundamentals
- STL Iterator Fundamentals
- The **array** Class
- The **vector** Class
- The **set** Classes
- The **map** Classes <next week>

The Standard Template Library

- **The Standard Template Library (STL):** an extensive library of generic templates for classes and functions.
- Categories of Templates:
 - **Containers:** Class templates for objects that store and organize data (vector, array, set, map)
 - **Algorithms:** Function templates that perform various operations on elements of containers (sort, find, count)
 - **Iterators:** Class templates for objects that behave like pointers, and are used to access the individual data elements in a container. **The bridge between containers and algorithms.**

Topic

STL Container Fundamentals

Containers

- **Sequence Containers**
 - Stores data sequentially in memory, in a fashion similar to an array
- **Associative Containers**
 - Stores data in a nonsequential way that makes it faster to locate elements
- Container Adapters
 - Wrap sequence/associative containers to make them appear as another type of data structure.

Sequence Containers

Container Class	Description
array	A fixed size container that is similar to an array
deque	A double ended queue. Similar to a vector but designed so values can be quickly added or removed from front and back.
forward_list	A singly linked list of data elements.
list	A doubly linked list of data elements
vector	A container that works like an expandable array.

Associative Containers

Container Class	Description
set	A set of unique values that are sorted. No duplicates.
multiset	A set of unique values that are sorted. Duplicates allowed.
map	Maps a set of keys to data elements. Only one key per data element are allowed. No duplicates. Elements sorted in key order.
multimap	Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates allowed. Elements sorted in key order.
unordered_set	Like a set but not sorted.
unordered_multiset	Like a multiset but not sorted
unordered_map	Like a map but not sorted
unordered_multimap	Like a multimap but not sorted

Container Adapters

Container Adapter Class	Description
stack	An adapter class that stores elements in a deque (by default) and acts as a LIFO container.
queue	An adapter class that stores elements in a deque (by default) and acts as a FIFO container.
priority_queue	An adapter class that stores its elements in a vector. A data structure in which the element that you retrieve is always the element with the greatest value.

STL Header Files

Header File	Classes
<array>	array
<deque>	deque
<forward_list>	forward_list
<list>	list
<map>	map, multimap
<queue>	queue, priority_queue
<set>	set, multiset
<stack>	stack
<unordered_map>	unordered_map, unordered_multimap
<unordered_set>	unordered_set, unordered_multiset
<vector>	vector

Topic

The **array** Class

(as an example, most comments apply to all
container classes)

The `array` Class Template

- The `array` class is declared in the `<array>` header file.
- A fixed-size container that holds elements of the same data type.
- `array` objects have a `size()` member function that returns the number of elements contained in the object.

The `array` Class Template

- When defining an `array` object, you specify the data type of its elements, and the number of elements.
- Examples:

```
std::array<int, 5> numbers;  
std::array<string, 4> names;
```

The `array` Class Template

- Initializing an `array` object:

```
std::array<int, 5> numbers{1, 2, 3, 4, 5};  
std::array<string, 4> names{"Jamie", "Ashley", "Doug", "Claire"};
```

The `array` Class Template

- `at()` and `get()` will check bounds.
- You can use the `[]` operator to access elements using a subscript, just as you would with a regular array.
- The `[]` operator does not perform bounds checking. Be careful not to use a subscript that is out of bounds.

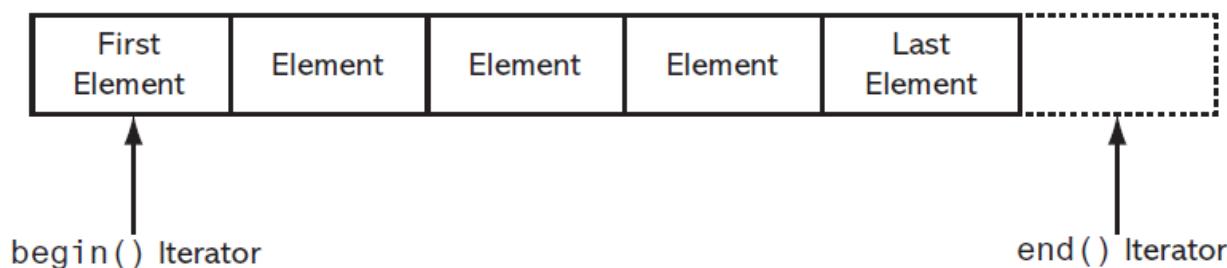
```
std::array<int, 5> prime { 2, 3, 5, 7, 11 };
std::cout << prime[1] << std::endl;
std::cout << prime.at(1) << std::endl; //best practice
std::cout << std::get<1>(prime) << std::endl; // best practice
```

Topic

STL Iterator Fundamentals

Iterators

- All of the STL containers have `begin()` and `end()` member functions that returns an iterator pointing to the first position and the position *after* the container's last element, respectively.



Iterators

- To define an iterator, you must know what type of container you will be using it with.
- The general format of an iterator definition:

containerType::iterator *iteratorName*;

Where *containerType* is the STL container type, and *iteratorName* is the name of the iterator variable that you are defining.

```
std::array<int, 5> myArray = {1, 2, 3, 4, 5};  
std::array<int, 5>::iterator iter1;  
  
std::vector<int>::iterator iter2;
```

Iterators for array

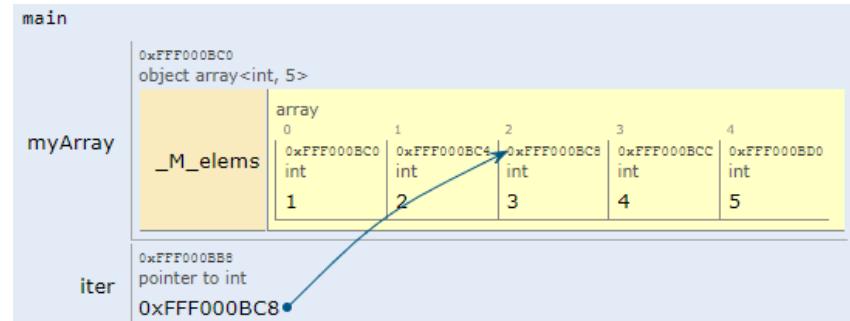
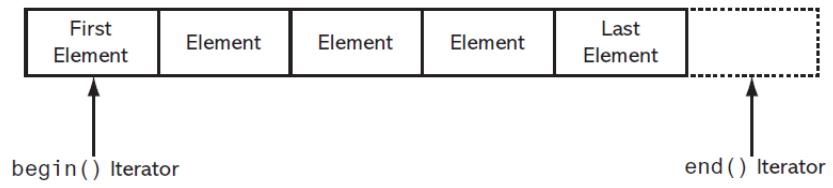
- To declare an iterator for an std::array, you first need to specify the type of the iterator **based on the type and size of the array**.

```
#include <array>
#include <iostream>
using namespace std;
int main()
{
    // Declare an std::array of type int with 5 elements
    std::array<int, 5> myArray = {1, 2, 3, 4, 5};

    // Declare an iterator for the std::array
    std::array<int, 5>::iterator iter;

    // Use the iterator to iterate through the array
    for (iter = myArray.begin(); iter != myArray.end(); ++iter)
    {
        std::cout << *iter << " ";
    }
    std::cout << std::endl;

    return 0;
}
```



Iterators - auto

- You can use the `auto` keyword to simplify the definition of an iterator.

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

int main() {
    // Declare an std::array of type int with 5 elements
    std::array<int, 5> myArray = {1, 2, 3, 4, 5};

    // Use the iterator to iterate through the array
    for (auto it = myArray.begin(); it != myArray.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Mutable Iterators

- An iterator of the `iterator` type gives you read/write access to the element to which the iterator points.
- This is commonly known as a mutable iterator.

```
// Define an array object.  
array<int, 5> numbers = {1, 2, 3, 4, 5};  
  
// Define an iterator for the array object.  
array<int, 5>::iterator it;  
  
// Make the iterator point to the array object's first element.  
it = numbers.begin();  
  
// Use the iterator to change the element.  
*it = 99;
```

Constant Iterators

- An iterator of the `const_iterator` type provides read-only access to the element to which the iterator points.
- The STL containers provide a `cbegin()` member function and a `cend()` member function.
 - The `cbegin()` member function returns a `const_iterator` pointing to the first element in a container.
 - The `cend()` member function returns a `const_iterator` pointing to the end of the container.
 - When working with `const_iterators`, simply use the container class's `cbegin()` and `cend()` member functions instead of the `begin()` and `end()` member functions.

Constant Iterators

```
#include <iostream>
#include <array>

int main()
{
    // Declare an std::array of type int with 5 elements
    std::array<int, 5> myArray = {1, 2, 3, 4, 5};

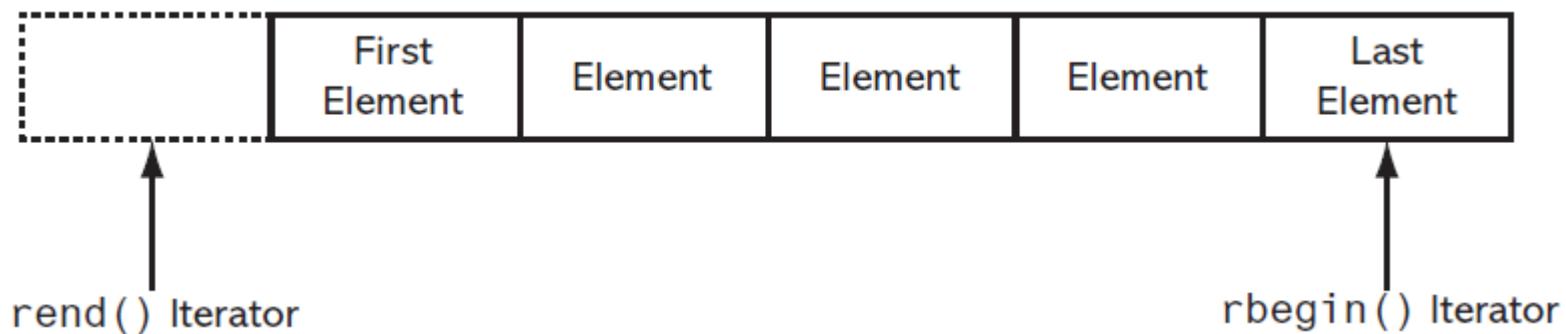
    // Declare a constant iterator for the array
    // std::array<int, 5>::const_iterator cit;

    std::cout << "elements are: ";
    // Use the constant iterator to access and print elements of the array
    for (auto cit = myArray.cbegin(); cit != myArray.cend(); ++cit)
    {
        std::cout << *cit << " ";
        // *cit = *cit * 2; // Uncommenting this line will cause a compilation error
    }
    std::cout << std::endl;

    return 0;
}
```

Reverse Iterators

- A *reverse iterator* works in reverse, allowing you to iterate backward over the elements in a container.



Reverse Iterators

- The following STL containers support reverse iterators:
 - `array`
 - `vector`
 - `map`
 - `set`
 - `multimap`
 - `multiset`
- All of these classes provide an `rbegin()` member function and an `rend()` member function.

Reverse Iterators

```
#include <iostream>
#include <array>

int main()
{
    // Declare an std::array of type int with 5 elements
    std::array<int, 5> myArray = {1, 2, 3, 4, 5};

    // Declare a reverse iterator for the array
    // std::array<int, 5>::reverse_iterator rit;

    std::cout << "elements are: ";
    // Use the reverse iterator to access and print elements of the array
    for (auto rit = myArray.rbegin(); rit != myArray.rend(); ++rit)
    {
        std::cout << *rit << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Topic

The **vector** Class

The `vector` Class

- A `vector` is a sequence container that works like an array, but is dynamic in size.
- The `vector` class is declared in the `<vector>` header file.

Accessing Elements

- Overloaded [] operator provides access to existing elements without bounds checking.
- You can use the `at()` member function to retrieve a `vector` element by its index with bounds checking:

```
std::vector<int> vec = {1, 2, 3, 4, 5};  
cout << vec[5] << endl;  
try  
{  
    std::cout << vec.at(5) << std::endl;  
}  
catch (const std::out_of_range &e)  
{  
    std::cerr << "My Error: " << e.what() << std::endl;  
}
```

-1414812757

My Error: vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)

vector Class Constructors

Default Constructor

`vector<dataType> name;`

Creates an empty `vector`.

Fill Constructor

`vector<dataType> name(size);`

Creates a `vector` of `size` elements. If the elements are objects, they are initialized via their default constructor. Otherwise, initialized with 0.

Fill Constructor

`vector<dataType> name(size, value);`

Creates a `vector` of `size` elements, each initialized with `value`.

vector Class Constructors

Range Constructor

`vector<dataType> name(iterator1, iterator2);`
Creates a `vector` that is initialized with a range of values from another container. `iterator1` marks the beginning of the range and `iterator2` marks the end.

Copy Constructor

`vector<dataType> name(vector2);`
Creates a `vector` that is a copy of `vector2`.

Initializing a **vector**

- In C++ 11 and later, you can initialize a **vector** object:

```
vector<int> numbers = {1, 2, 3, 4, 5};
```

or

```
vector<int> numbers {1, 2, 3, 4, 5};
```

Adding New Elements to a vector

- The `push_back` member function adds a new element to **the end of a vector**:

```
vector<int> numbers;  
numbers.push_back(10);  
numbers.push_back(20);  
numbers.push_back(30);
```

Using an Iterator With a `vector`

- `vectors` have `begin()` and `end()` member functions that return iterators pointing to the beginning and end of the container:

```
#include <iostream>
#include <vector>
int main()
{
    // Initialize a vector with some values
    std::vector<int> vec = {10, 20, 30, 40, 50};

    // Declare a iterator for the vector
    // std::vector<int>::iterator iter;

    std::cout << "Vector elements are: ";
    // Use the iterator to access and print elements of the vector
    for (auto iter = vec.begin(); iter != vec.end(); ++iter)
    {
        std::cout << *iter << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Using an Iterator With a `vector`

- The `begin()` and `end()` member functions return a random-access iterator of the `iterator` type
- The `cbegin()` and `cend()` member functions return a random-access iterator of the `const_iterator` type
- The `rbegin()` and `rend()` member functions return a reverse iterator of the `reverse_iterator` type
- The `crbegin()` and `crend()` member functions return a reverse iterator of the `const_reverse_iterator` type

Inserting Elements with the `insert()` Member Function

- You can use the `insert()` member function, along with an iterator, to insert an element at a specific position.
- General format:

```
vectorName.insert(it, value);
```

Iterator pointing to
an element in the
vector

Value to insert before
the element that *it*
points to

Overloaded Versions of the `insert()` Member Function

<code>insert(it, value)</code>	Inserts <i>value</i> just before the element pointed to by <i>it</i> . The function returns an iterator pointing to the newly inserted element.
<code>insert(it, n, value)</code>	Inserts <i>n</i> elements just before the element pointed to by <i>it</i> . Each of the new elements will be initialized with <i>value</i> . The function returns an iterator pointing to the first element of the newly inserted elements.
<code>insert(iterator1, iterator2, iterator3)</code>	Inserts a range of new elements. <i>iterator1</i> points to an existing element in the container. The range of new elements will be inserted before the element pointed to by <i>iterator1</i> . <i>iterator2</i> and <i>iterator3</i> mark the beginning and end of a range of values that will be inserted. (The element pointed to by <i>iterator3</i> will not be included in the range.) The function returns an iterator pointing to the first element of the newly inserted range.

Inserting Elements with the `insert()` Member Function

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

int main() {
    std::vector<int> vec = {10, 20, 30, 40, 50};

    // Insert a new element (25) before the third element (30)
    std::vector<int>::iterator it = vec.begin() + 2;
    vec.insert(it, 25);

    // Display the vector after insertion
    std::cout << "After inserting 25: ";
    for(auto num : vec) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    // Insert three copies of the number 15 at the beginning
    vec.insert(vec.begin(), 3, 15);

    // Display the vector after inserting three 15s at the beginning
    std::cout << "After inserting three 15s at the beginning: ";
    for(auto num : vec) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

After inserting 25: 10 20 25 30 40 50
After inserting three 15s at the beginning: 15 15 15 10 20 25 30 40 50

Vector - Common Member Functions

Function	Description
push_back(value)	Add an element to the end.
pop_back()	Remove the last element.
size()	Return the number of elements.
capacity()	Return the current allocated storage capacity.
empty()	Return true if the vector is empty.
clear()	Remove all elements.
insert(iterator, value)	Insert an element at a specific position.
erase(iterator)	Remove an element at a specific position.
erase(iterator first, iterator last)	Removes elements in the range [first, last)
resize(n)	Change the number of elements to n.
reserve(n)	Preallocate memory for n elements (optimization).
shrink_to_fit()	Reduces capacity to match size.

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

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(30);
    vec.pop_back(); // Removes 30
    std::cout << "Size: " << vec.size() << std::endl;
    std::cout << "Capacity: " << vec.capacity() << std::endl;
    std::cout << "Is empty? " << (vec.empty() ? "Yes" : "No") << std::endl;
    vec.clear();
    std::cout << "Size after clear: " << vec.size() << std::endl;
    vec.push_back(10);
    vec.push_back(30);
    vec.insert(vec.begin() + 1, 20); // Insert 20 at index 1
    vec.erase(vec.begin()); // Removes the first element (10)
    vec.push_back(40);
    vec.push_back(50);
    vec.erase(vec.begin(), vec.begin() + 2); // Removes first two elements
    vec.reserve(10); // Reserves capacity for 10 elements (does not change size)
    vec.shrink_to_fit();
    return 0;
}
```

Topic

The set Classes

Sets

- A *set* is an associative container that is similar to a mathematical set.
- You can use the STL `set` class to create a set container.
- All the elements in a `set` must be **unique**.
 - No two elements can have the same value.
- The elements in a set are automatically **sorted** in ascending order.
- The `set` class is declared in the `<set>` header file.

set Class Constructors

Default Constructor

`set<dataType> name;`

Creates an empty **set**.

Range Constructor

`set<dataType> name(iterator1, iterator2);`

Creates a **set** that is initialized with a range of values.

iterator1 marks the beginning of the range and

iterator2 marks the end.

Copy Constructor

`set<dataType> name(set2);`

Creates a **set** that is a copy of **set2**.

The **set** Class

- Example: defining a **set** container to hold integers:

```
set<int> numbers;
```

- Example: defining and initializing a **set** container to hold integers:

```
set<int> numbers = {1, 2, 3, 4, 5};
```

The set Class

- A set cannot contain duplicate items.
- If the same value appears more than once in an initialization list, it will be added to the set only one time.
- For example, the following set will contain the values 1, 2, 3, 4, and 5:

```
set<int> numbers = {1, 1, 2, 2, 3, 4, 5, 5, 5};
```

Adding New Elements to a set

- The `insert()` member function adds a new element to a `set`:

```
set<int> numbers;  
numbers.insert(10);  
numbers.insert(20);  
numbers.insert(30);
```

Stepping Through a set With the Range-Based for Loop

```
// Create a set containing names.  
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};  
  
// Display each element.  
for (string element : names)  
{  
    cout << element << endl;  
}
```

Using an Iterator With a `set`

- The `begin()` and `end()` member functions return a bidirectional iterator of the `iterator` type
- The `cbegin()` and `cend()` member functions return a bidirectional iterator of the `const_iterator` type
- The `rbegin()` and `rend()` member functions return a reverse bidirectional iterator of the `reverse_iterator` type
- The `crbegin()` and `crend()` member functions return a reverse bidirectional iterator of the `const_reverse_iterator` type

Using an Iterator With a set

```
// Create a set containing names.  
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};  
  
// Create an iterator.  
set<string>::iterator iter;  
  
// Use the iterator to display each element in the set.  
for (iter = names.begin(); iter != names.end(); iter++)  
{  
    cout << *iter << endl;  
}
```

Determining Whether an Element Exists

- The `set` class's `count()` member function accepts a value as its argument, and returns 1 if that value exists in the set. The function returns 0 otherwise.

```
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};
if (names.count("Lisa"))
    cout << "Lisa was found in the set.\n";
else
    cout << "Lisa was not found.\n";
```

Retrieving an Element

- The **set** class has a **find()** member function that searches for an element with a specified value.
- The **find()** function returns an iterator to the element matching it.
- If the element is not found, the **find()** function returns an iterator to the end of the **set**.

Retrieving an Element

```
// Create a set containing names.  
set<string> names = {"Joe", "Karen", "Lisa", "Jackie"};  
  
// Create an iterator.  
set<string>::iterator iter;  
  
// Find "Karen".  
iter = names.find("Karen");  
  
// Display the result.  
if (iter != names.end())  
{  
    cout << *iter << " was found.\n";  
}  
else  
{  
    cout << "Karen was not found.\n";  
}
```

The **unordered_set** Class

- The **unordered_set** class is similar to the **set** class, except in two regards:
 - The values in an **unordered_set** are not sorted
 - The **unordered_set** class has better performance
- You should use the **unordered_set** class instead of the **set** class if:
 - You will be making a lot of searches on a large number of elements
 - You are not concerned with retrieving them in ascending order
- The **unordered_set** class is declared in the **<unordered_set>** header file

The **multiset** Class

- The **multiset** class is a set that allows **duplicate** items.
- The **multiset** class has the same member functions as the **set** class.
- The **multiset** class is declared in the **<set>** header file.

The unordered_multiset Class

- The unordered_multiset class is similar to the multiset class, except in two regards:
 - The values in an unordered_multiset are not sorted
 - The unordered_multiset class has better performance
- You should use the unordered_multiset class instead of the multiset class if:
 - You will be making a lot of searches on a large number of elements
 - You are not concerned with retrieving them in ascending order
- The unordered_multiset class is declared in the `<unordered_set>` header file

2800ICT

Object Oriented Programming

STL Container – Map class

STL - Algorithms

Review Week2

- Introduction to STL
- STL Container Fundamentals
- STL Iterator Fundamentals
- The **array** Class
- The **vector** Class
- The **set** Classes
- The **map** Classes <next week>

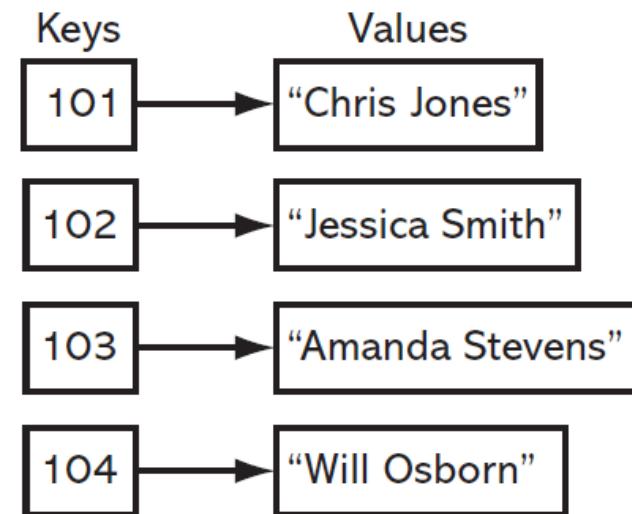
```
#include <array>
#include <iostream>
using namespace std;
int main()
{
    array<int, 5> myArray {1, 2, 3, 4, 5};
    for (auto it = myArray.begin(); it != myArray.end(); ++it)
    {
        cout << *it << " ";
    }
    cout << endl;
    return 0;
}
```

Topic

STL Container - map Classes

Maps – General Concepts

- A *map* is an associative container.
 - Nonsequential memory
- Each element that is stored in a map has two parts
 - **key**: **unique** – no duplicates.
 - **value**



- To retrieve a specific value from a map, you use the key that is associated with that value.

The map Class

- You can use the STL `map` class to store key-value pairs.
- The `map` class is declared in the `<map>` header file.

map Class Constructors

Default Constructor

```
map<keyDataType, valueDataType> name;  
Creates an empty map.
```

Range Constructor

```
map<keyDataType, valueDataType>  
name(iterator1, iterator2);  
Creates a map that is initialized with a range of values from  
another map. iterator1 marks the beginning of the range  
and iterator2 marks the end.
```

Copy Constructor

```
map<keyDataType, valueDataType> name(map2);  
Creates a map that is a copy of map2.
```

The map Class

- Example: defining a `map` container to hold employee ID numbers (as `ints`) and their corresponding employee names (as `strings`):

```
map<int, string> employees;
```

Key data
type

Value data
type

Initializing a map

```
map<int, string> employees =  
{  
    {101, "Chris Jones"}, {102, "Jessica Smith"},  
    {103, "Amanda Stevens"}, {104, "Will Osborn"}  
};
```

- In the first element, the key is 101 and the value is "Chris Jones".
- In the second element, the key is 102 and the value is "Jessica Smith".
- In the third element, the key is 103 and the value is "Amanda Stevens".
- In the fourth element, the key is 104 and the value is "Will Osborn".

Retrieving Elements with the at() Member Function

- You can use the `at()` member function to retrieve a `map` element by its key:

```
// Create a map containing employee IDs and names.  
map<int, string> employees =  
{  
    {101, "Chris Jones"}, {102, "Jessica Smith"},  
    {103, "Amanda Stevens"}, {104, "Will Osborn"}  
};  
  
// Retrieve a value from the map.  
cout << employees.at(103) << endl;
```

Displays "Amanda Stevens"

Retrieving Elements with the `at()` Member Function

- To prevent the `at()` member function from throwing an exception (if the specified key does not exist), use the `count()` member function to determine whether it exists:

```
// Create a map containing employee IDs and names.  
map<int, string> employees =  
{  
    {101, "Chris Jones"}, {102, "Jessica Smith"},  
    {103, "Amanda Stevens"}, {104, "Will Osborn"}  
};
```

```
// Retrieve a value from the map.  
if (employees.count(103))  
    cout << employees.at(103) << endl;  
else  
    cout << "Employee not found.\n";
```

The `count()` member function returns 1 if the specified key exists, or 0 otherwise.

The Overloaded [] Operator

- You can use the [] operator to add new elements to a map.
- General format:

mapName[key] = value;

- This adds the key-value pair to the map.
- If the key already exists in the map, **it's associated value will be changed to *value*.**

The Overloaded [] Operator

```
map<int, string> employees;  
employees[110] = "Beth Young";  
employees[111] = "Jake Brown";  
employees[112] = "Emily Davis";
```

- After this code executes, the `employees` map will contain the following elements:
 - Key = 110, Value = "Beth Young"
 - Key = 111, Value = "Jake Brown"
 - Key = 112, Value = "Emily Davis"

The pair Type

- Internally, the elements of a `map` are stored as instances of the `pair` type.
- `pair` is a struct that has two member variables: `first` and `second`.
- The element's key is stored in `first`, and the element's value is stored in `second`.
- The `pair` struct is declared in the `<utility>` header file.
 - When you `#include` the `<map>` header file, `<utility>` is automatically included as well.

Inserting Elements with the `insert()` Member Function

- The `map` class provides an `insert()` member function that adds a `pair` object as an element to the `map`.
- You can use the STL function template `make_pair` to construct a `pair` object.
- The `make_pair` function template is declared in the `<utility>` header file.

Inserting Elements with the `insert()` Member Function

```
map<int, string> employees;
employees.insert(make_pair(110, "Beth Young"));
employees.insert(make_pair(111, "Jake Brown"));
employees.insert(make_pair(112, "Emily Davis"));
```

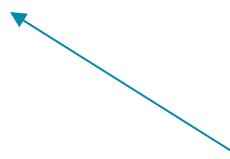
- After this code executes, the `employees` map will contain the following elements:
 - Key = 110, Value = "Beth Young"
 - Key = 111, Value = "Jake Brown"
 - Key = 112, Value = "Emily Davis"

Note: If the element that you are inserting with the `insert()` member function has the same key as an existing element, **the function will not insert the new element.**

Deleting Elements

- You can use the `erase()` member function to delete a map element by its key:

```
// Create a map containing employee IDs and names.  
map<int, string> employees =  
{  
    {101, "Chris Jones"}, {102, "Jessica Smith"},  
    {103, "Amanda Stevens"}, {104, "Will Osborn"}  
};  
  
// Delete the employee with ID 102.  
employees.erase(102);
```



Deletes Jessica Smith from the map

Stepping Through a map with the Range-Based for Loop

```
// Create a map containing employee IDs and names.  
map<int, string> employees =  
{  
    {101, "Chris Jones"}, {102, "Jessica Smith"},  
    {103, "Amanda Stevens"}, {104, "Will Osborn"}  
};  
  
// Display each element.  
for (pair<int, string> element : employees)  
{  
    cout << "ID: " << element.first << "\tName: "  
        << element.second << endl;  
}
```

Remember, each element is a pair.

Stepping Through a map with the Range-Based for Loop

```
// Create a map containing employee IDs and names.  
map<int, string> employees =  
{  
    {101, "Chris Jones"}, {102, "Jessica Smith"},  
    {103, "Amanda Stevens"}, {104, "Will Osborn"}  
};  
  
// Display each element.  
for (auto element : employees)  
{  
    cout << "ID: " << element.first << "\tName: "  
        << element.second << endl;  
}
```

Using an Iterator With a `map`

- The `begin()` and `end()` member functions return a bidirectional iterator of the `iterator` type
- The `cbegin()` and `cend()` member functions return a bidirectional iterator of the `const_iterator` type
- The `rbegin()` and `rend()` member functions return a reverse bidirectional iterator of the `reverse_iterator` type
- The `crbegin()` and `crend()` member functions return a reverse bidirectional iterator of the `const_reverse_iterator` type

Using an Iterator With a `map`

- When an iterator points to a `map` element, it points to an instance of the `pair` type.
- The element has two member variables: `first` and `second`.
- The element's key is stored in `first`, and the element's value is stored in `second`.

```
// map_iter.cpp
#include <iostream>
#include <map>
#include <string>

int main()
{
    // Creating a map to represent employee records: employee ID to name
    std::map<int, std::string> employeeRecords;

    // Inserting some employee records into the map
    employeeRecords.insert(std::make_pair(101, "John Doe"));
    employeeRecords.insert(std::make_pair(102, "Jane Smith"));
    employeeRecords.insert(std::make_pair(103, "Alice Johnson"));

    // Declare an iterator for a map
    // std::map<int, std::string>::iterator it

    // Using an iterator to iterate over the map and print each employee's ID and name
    std::cout << "Employee Records:\n";
    for (auto it = employeeRecords.begin(); it != employeeRecords.end(); ++it)
    {
        std::cout << "ID: " << it->first << ", Name: " << it->second << std::endl;
        // it->first is equivalent to (*it).first
        std::cout << "ID: " << (*it).first << ", Name: " << (*it).second << std::endl;
    }
    return 0;
}
```

The `unordered_map` Class

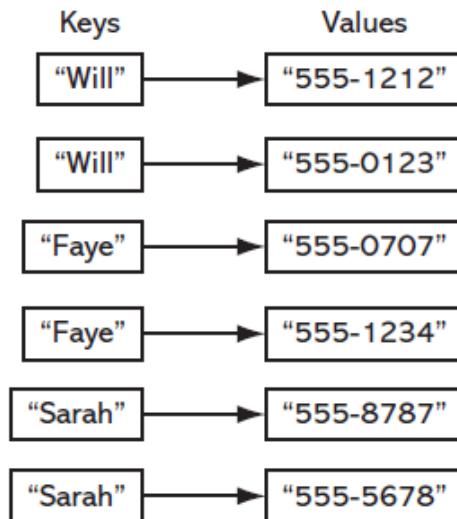
- The `unordered_map` class is similar to the `map` class, except in two regards:
 - The keys in an `unordered_map` are not sorted
 - The `unordered_map` class has better performance
- You should use the `unordered_map` class instead of the `map` class if:
 - You will be making a lot of searches on a large number of elements
 - You are not concerned with retrieving them in key order
- The `unordered_map` class is declared in the `<unordered_map>` header file

The `multimap` Class

- The `multimap` class is a map that allows **duplicate keys**
- The `multimap` class has most of the same member functions as the `map` class
- The `multimap` class is declared in the `<map>` header file

The `multimap` Class

- Consider a phonebook application where the key is a person's name and the value is that person's phone number.
- A `multimap` container would allow each person to have multiple phone numbers



Adding Elements to a `multimap`

- The `multimap` class **does not** overload the `[]` operator.
 - So, you cannot use an assignment statement to add a new element to a `multimap`.
- Instead, you will use the `insert()` member functions.

Adding Elements to a multimap

```
multimap<string, string> phonebook;
phonebook.insert(make_pair("Will", "555-1212"));
phonebook.insert(make_pair("Will", "555-0123"));
phonebook.insert(make_pair("Faye", "555-0707"));
phonebook.insert(make_pair("Faye", "555-1234"));
phonebook.insert(make_pair("Sarah", "555-8787"));
phonebook.insert(make_pair("Sarah", "555-5678"));
```

Retrieving Elements with a Specified Key

- The `multimap` class has a `find()` member function that searches for an element with a specified key.
- The `find()` function returns **an iterator** to the first element matching it.
- If the element is not found, the `find()` function returns an iterator to the end of the `multimap`.

Deleting Elements with a Specified Key

- To delete all elements matching a specified key, use the `erase()` member function.

```
// Define a phonebook multimap.  
multimap<string, string> phonebook =  
    { {"Will", "555-1212"}, {"Will", "555-0123"},  
      {"Faye", "555-0707"}, {"Faye", "555-1234"},  
      {"Sarah", "555-8787"}, {"Sarah", "555-5678"} };  
  
// Delete Will's phone numbers from the multimap.  
phonebook.erase("Will");
```

The `unordered_multimap` Class

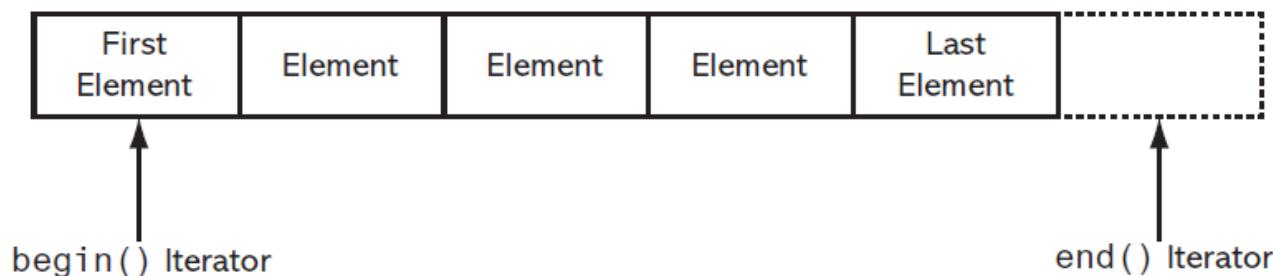
- The `unordered_multimap` class is similar to the `multimap` class, except:
 - The keys in an `unordered_multimap` are **not sorted**
 - The `unordered_multimap` class has better performance
- You should use the `unordered_multimap` class instead of the `multimap` class if:
 - You will be making a lot of searches on a large number of elements
 - You are not concerned with retrieving them in key order
- The `unordered_multimap` class is declared in the `<unordered_multimap>` header file

Topic

STL Algorithms

STL Algorithms

- The STL provides a number of algorithms, implemented as function templates, in the `<algorithm>` header file.
- These functions perform various operations on ranges of elements.



Categories of Algorithms in the STL

- **Min/max** algorithms
- **Sorting** algorithms
- **Search** algorithms
- Read-only sequence algorithms
- **Copying and moving** algorithms
- Swapping algorithms
- Replacement algorithms
- **Removal** algorithms
- Reversal algorithms
- **Fill** algorithms
- Rotation algorithms
- **Shuffling** algorithms
- **Set** algorithms
- Transformation algorithm
- Partition algorithms
- **Merge** algorithms
- Permutation algorithms
- Heap algorithms
- Lexicographical comparison algorithm

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

int main()
{
    std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6};

    auto minIt = std::min_element(v.begin(), v.end());
    auto maxIt = std::max_element(v.begin(), v.end());

    std::cout << "Minimum element in vector: " << *minIt << std::endl;
    std::cout << "Maximum element in vector: " << *maxIt << std::endl;

    // For both min and max
    auto result = std::minmax_element(v.begin(), v.end());
    std::cout << "Min and Max elements are: "
           << *result.first << " and " << *result.second << std::endl;

    return 0;
}
```

Plugging Your Own Functions into an Algorithm

- Many of the function templates in the STL are designed to accept function pointers as arguments.
- This allows you to “plug” one of your own functions into the algorithm.
- For example:

```
for_each(iterator1, iterator2, function)
```

- *iterator1* and *iterator2* mark the beginning and end of a range of elements.
- *function* is the name of a function that accepts an element as its argument.
- The `for_each()` function iterates over the range of elements, passing **each element** as an argument to *function*.

for_each

```
// for_each.cpp
#include <algorithm> // for std::for_each
#include <vector>    // for std::vector
#include <iostream>   // for std::cout

// Function to double a number
void doubleNumber(int &n)
{
    n *= 2;
}

int main()
{
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Use std::for_each to apply doubleNumber to each element in the vector
    std::for_each(numbers.begin(), numbers.end(), doubleNumber);

    // Print the modified vector
    for (int number : numbers)
    {
        std::cout << number << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Plugging Your Own Functions into an Algorithm

- Another example:

`count_if(iterator1, iterator2, function)`

- *iterator1* and *iterator2* mark the beginning and end of a range of elements.
- *function* is the name of a function that accepts **an element** as its argument, and **returns either true or false**.
- The `count_if()` function iterates over the range of elements, passing each element as an argument to *function*.
- The `count_if` function returns the number of elements for which *function* returns **true**.

count_if

```
#include <algorithm> // for std::count_if
#include <vector>    // for std::vector
#include <iostream>   // for std::cout

// Predicate function that checks if an integer is even
bool isEven(int number)
{
    return number % 2 == 0;
}

int main()
{
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8};

    // Use std::count_if with the isEven predicate function
    int evenCount = std::count_if(numbers.begin(), numbers.end(), isEven);

    // Print the count of even numbers
    std::cout << "There are " << evenCount << " even numbers in the vector." << std::endl;

    return 0;
}
```

Sorting

- The `sort` function:

```
sort(iterator1, iterator2);
```

iterator1 and *iterator2* mark the beginning and end of a range of elements.

- The function sorts the range of elements in **ascending** order.
- Also have to allow user to define the meaning of < :

```
sort(iterator1, iterator2, function);
```

```

// sort_example.cpp
#include <algorithm>
#include <array>
#include <iostream>
void myPrint(int n){
    std::cout << n << " ";
}
// Custom comparator function (Descending Order)
bool customCompare(int a, int b) {
    return a > b; // Reverse comparison
}
int main()
{
    std::array<int, 10> s{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
    std::cout << "sorted with the default operator <: std::less\n";
    std::sort(s.begin(), s.end());
    std::for_each(s.begin(), s.end(), myPrint);
    std::cout << std::endl;
    std::cout << "sorted with the standard library compare function: std::greater\n";
    std::sort(s.begin(), s.end(), std::greater<int>());
    // std::sort(s.begin(), s.end(), customCompare);
    std::for_each(s.begin(), s.end(), myPrint);
    std::cout << std::endl;
}

```

sorted with the default operator <: std::less

0 1 2 3 4 5 6 7 8 9

sorted with the standard library compare function: std::greater

9 8 7 6 5 4 3 2 1 0

Sorting

Searching

- The `binary_search` function:

```
binary_search(iterator1, iterator2, value);
```

iterator1 and *iterator2* mark the beginning and end of a range of elements **that are sorted in ascending order**. *value* is the value to search for. The function returns **true** if *value* is found in the range, or **false** otherwise.

Searching

```
#include <algorithm> // For std::sort and std::binary_search
#include <vector>    // For std::vector
#include <iostream>   // For std::cout

int main()
{
    std::cout<<std::boolalpha;

    // Initialize a vector with unsorted integers
    std::vector<int> numbers = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    // Sort the vector in ascending order
    std::sort(numbers.begin(), numbers.end());

    // Output the sorted vector
    std::cout << "Sorted vector: ";
    for (int num : numbers)
    {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Search for a value in the sorted vector
    int valueToFind = 6;
    bool found = std::binary_search(numbers.begin(), numbers.end(), valueToFind);

    // Output the result of the search
    std::cout << found << std::endl;
    return 0;
}
```

Search algorithms

- **iterator find(iterator1, iterator2, value)**
 - returns an iterator that points to first occurrence of value

```
#include <iostream>
#include <vector>
#include <algorithm> // For std::find

int main()
{
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Searching for the value 3 in the vector
    auto it = std::find(vec.begin(), vec.end(), 3);

    if (it != vec.end())
    {
        std::cout << "Found value " << *it << " at position " << (it - vec.begin()) << std::endl;
    }
    else
    {
        std::cout << "Value not found." << std::endl;
    }

    return 0;
}
```

Search algorithms

- **iterator `find_if(iterator1, iterator2, function)`**
 - returns an iterator that points to the first element for which *function* returns true.

```
#include <iostream>
#include <vector>
#include <algorithm> // For std::find_if
bool isEven(int value)
{
    return value % 2 == 0;
}
int main()
{
    std::vector<int> vec = {1, 3, 5, 4, 2};

    // Searching for the first even number in the vector
    auto it = std::find_if(vec.begin(), vec.end(), isEven);

    if (it != vec.end())
    {
        std::cout << "Found even value " << *it << " at position " << (it - vec.begin()) << std::endl;
    }
    else
    {
        std::cout << "No even values found." << std::endl;
    }
    return 0;
}
```

Fill and Generate

- **fill(iterator1, iterator2, value);**
 - fills the values of the elements between **iterator1** and **iterator2** with *value*
- **fill_n(iterator1, n, value);**
 - changes specified number of elements starting at **iterator1** to *value*
- **generate(iterator1, iterator2, function);**
 - similar to fill except that it calls a **function** to return value
- **generate_n(iterator1, n, function)**
 - same as **fill_n** except that it calls a function to return value

Comparing sequences of values

- `bool equal(iterator1, iterator2, iterator3);`
 - compares sequence from `iterator1` to `iterator2` with the sequence beginning at `iterator3`
 - return true if they are equal, false otherwise

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

int main()
{
    std::vector<int> vec1 = {1, 2, 3, 4, 5};
    std::vector<int> vec2 = {1, 2, 3, 4, 5};
    std::vector<int> vec3 = {1, 2, 3, 4, 6};

    bool isEqual12 = std::equal(vec1.begin(), vec1.end(), vec2.begin());
    bool isEqual13 = std::equal(vec1.begin(), vec1.end(), vec3.begin());

    std::cout << "vec1 and vec2 are " << (isEqual12 ? "equal." : "not equal.") << std::endl;
    std::cout << "vec1 and vec3 are " << (isEqual13 ? "equal." : "not equal.") << std::endl;

    return 0;
}

vec1 and vec2 are equal.
vec1 and vec3 are not equal.
```

Comparing sequences of values

- **pair mismatch(iterator1, iterator2, iterator3);**
 - compares sequence from **iterator1** to **iterator2** with the sequence starting at **iterator3**
 - returns a **pair** object with iterators pointing to where first mismatch occurred

```
#include <iostream>
#include <algorithm>
#include <vector>
int main()
{
    std::vector<int> vec1 = {1, 2, 3, 4, 5};
    std::vector<int> vec2 = {1, 2, 3, 7, 5};

    auto result = std::mismatch(vec1.begin(), vec1.end(), vec2.begin());

    if (result.first != vec1.end())
    {
        std::cout << "First mismatching pair: (" 
              << *result.first << ", " << *result.second << ")" << std::endl;
    }
    else
    {
        std::cout << "The sequences are equal." << std::endl;
    }
    return 0;
}
```

Removing elements from containers

- `iterator remove(iterator1, iterator2, value);`
 - shifts elements in the range `iterator1` and `iterator2` that are not equal to `value` towards the beginning of the range, keeping the remaining elements in their relative order
 - returns an iterator to **the new logical end of the range.**

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 3, 5, 3};

    auto newEnd = std::remove(vec.begin(), vec.end(), 3); // Remove all '3's
    for (int val : vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    vec.erase(newEnd, vec.end()); // Erase the "removed" elements
    for (int val : vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Tips: to physically remove elements from a container, use `remove` together with `erase`.

```
1 2 4 5 3 5 3
1 2 4 5
```

Removing elements from containers

- **iterator `remove_if(iterator1, iterator2, function);`**
 - removes elements from a container that satisfy a specific condition.
 - returns an iterator to **the new logical end of the range**.
 - does not erase elements from the container but rearranges the elements so that unwanted ones are moved to the end.

```
#include <iostream>
#include <vector>
#include <algorithm>
bool isOdd(int x) {
    return x % 2 != 0;
}
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto newEnd = std::remove_if(numbers.begin(), numbers.end(), isOdd);
    // Remove odd numbers from the vector
    numbers.erase(newEnd, numbers.end());

    // Print the vector after removing odd numbers
    std::cout << "Vector after removing odd numbers:" << std::endl;
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

return 0;
}
```

Tips: to physically remove elements from a container, use `remove_if` together with `erase`.

Unique and Reverse order

- **iterator unique(iterator1, iterator2)**
 - removes (logically) duplicate elements from a sorted list
 - returns iterator to the new end of sequence

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

int main() {
    std::vector<int> numbers = {1, 2, 2, 3, 3, 4, 5, 5, 5, 6};

    // Removing consecutive duplicates
    auto newEnd = std::unique(numbers.begin(), numbers.end());

    // Erasing the elements beyond the new end
    numbers.erase(newEnd, numbers.end());

    // Printing the vector after removing duplicates
    std::cout << "Vector after removing consecutive duplicates:" << std::endl;
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Tips: to physically remove elements from a container, use **unique** together with **erase**.

Replacing elements

- `replace(iterator1, iterator2, value, newvalue);`
 - replaces *value* with *newvalue* for the elements located in the range iterator1 to iterator2
- `replace_if(iterator1, iterator2, function, newvalue);`
 - replaces all elements in the range iterator1 - iterator2 for which *function* returns true with *newvalue*

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

int main()
{
    std::vector<int> vec = {1, 2, 3, 4, 2, 6, 2};

    // Replace all instances of '2' with '5'
    std::replace(vec.begin(), vec.end(), 2, 5);

    for (int val : vec)
    {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Replacing elements

- `replace_copy(iterator1, iterator2, iterator3, value, newvalue);`
 - replaces and copies elements of the range `iterator1 - iterator2` to `iterator3`
- `replace_copy_if(iterator1, iterator2, iterator3, function, newvalue);`
 - replaces and copies elements for which `function` returns true to `iterator3`

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

int main()
{
    std::vector<int> vec = {1, 2, 3, 4, 2, 6, 2};
    std::vector<int> result(vec.size());

    // Copy vec to result, replacing all '2's with '5's
    std::replace_copy(vec.begin(), vec.end(), result.begin(), 2, 5);

    for (int val : result)
    {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Copy and Merge

- `copy(iterator1, iterator2, iterator3)`
 - copies the range of elements from `iterator1` to `iterator2` into `iterator3`
- `copy_backward(iterator1, iterator2, iterator3)`
 - copies in reverse order the range of elements from `iterator1` to `iterator2` into `iterator3`
- `merge(iter1, iter2, iter3, iter4, iter5)`
 - ranges `iter1 - iter2` and `iter3 - iter4` must be **sorted** and copies both lists into `iter5` in ascending order

Utility algorithms

- `random_shuffle(iterator1, iterator2)`
 - randomly mixes elements in the range `iterator1 - iterator2`
- `int count(iterator1, iterator2, value)`
 - returns number of instances of `value` in the range
- `int count_if(iterator1, iterator2, function)`
 - counts number of instances for which `function` returns true

Utility algorithms

- **iterator `min_element(iterator1, iterator2)`**
 - returns iterator to smallest element
 - `min_element(it1, it2)` – `it1` gives the index of the minimum element
- **iterator `max_element(iterator1, iterator2)`**
 - returns iterator to largest element
 - `max_element(it1, it2)` – `it1` gives the index of the maximum element
- **`accumulate(iterator1, iterator2, initVal)`**
 - returns the sum of the elements in the range

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

int main()
{
    std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6};

    auto minIt = std::min_element(v.begin(), v.end());
    auto maxIt = std::max_element(v.begin(), v.end());

    std::cout << "Minimum element in vector: " << *minIt << std::endl;
    std::cout << "Maximum element in vector: " << *maxIt << std::endl;

    // For both min and max
    auto result = std::minmax_element(v.begin(), v.end());
    std::cout << "Min and Max elements are: "
           << *result.first << " and " << *result.second << std::endl;

    return 0;
}
```

Utility algorithms

- `for_each(iterator1, iterator2, function)`
 - calls function on every element in range
- `transform(iterator1, iterator2, iterator3, function)`
 - calls *function* for all elements in range iterator1 - iterator2, and copies result to iterator3

Algorithms on sets

- Must be sorted sequences
- `includes(iter1, iter2, iter3, iter4)`
 - returns true if `iter1 - iter2` contains `iter3 - iter4`.
- `set_difference(iter1, iter2, iter3, iter4, iter5)`
 - copies elements in range `iter1 - iter2` that do not exist in second range `iter3 - iter4` into `iter5`
- `set_intersection(iter1, iter2, iter3, iter4, iter5)`
 - copies common elements from the two ranges `iter1 - iter2` and `iter3 - iter4` into `iter5`

Algorithms on sets

- `set_symmetric_difference(iter1, iter2, iter3, iter4, iter5)`
 - copies elements in range `iter1 - iter2` that are not in range `iter3 - iter4` and vice versa, into `iter5`. Both sets must be sorted
- `set_union(iter1, iter2, iter3, iter4, iter5)`
 - copies elements in both ranges to `iter5`. Both sets must be sorted

```

#include <algorithm>
#include <iostream>
#include <vector>
void print_result(std::vector<int>& result, std::string flag)
{
    std::cout << flag;
    for (int n : result)
        std::cout << n << ' ';
    std::cout << '\n';
    result.clear();
}
int main()
{
    std::vector<int> v1 = {1, 2, 4, 5, 6};
    std::vector<int> v2 = {2, 3, 5, 7};
    std::vector<int> result;
    // Set Union
    std::set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(result));
    print_result(result, "Union: ");

    // Set Intersection
    std::set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(result));
    print_result(result, "Intersection: ");

    // Set Difference
    std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(result));
    print_result(result, "Difference (v1 - v2): ");

    // Set Symmetric Difference
    std::set_symmetric_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(result));
    print_result(result, "Symmetric Difference: ");
    return 0;
}

```

```

Union: 1 2 3 4 5 6 7
Intersection: 2 5
Difference (v1 - v2): 1 4 6
Symmetric Difference: 1 3 4 6 7

```

2800ICT

Object Oriented Programming

Classes

Review Week3

STL Container – Map class STL - Algorithms

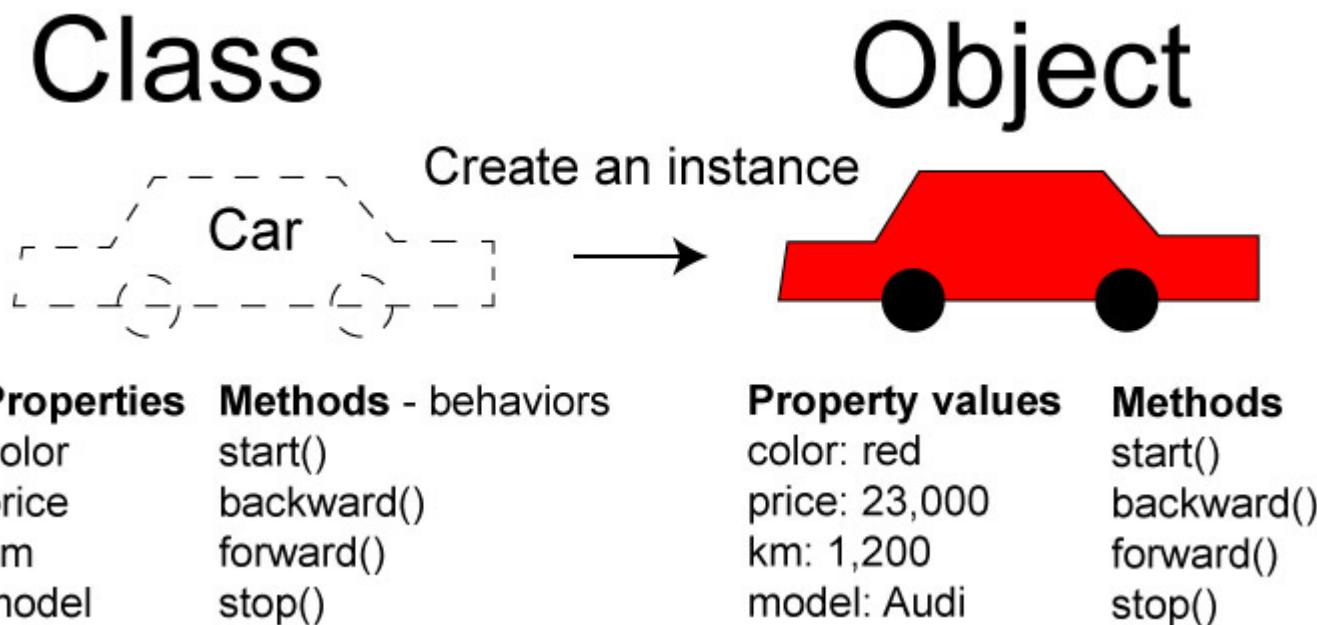
- **Min/max** algorithms
- **Sorting** algorithms
- **Search** algorithms
- Read-only sequence algorithms
- **Copying and moving** algorithms
- Swapping algorithms
- Replacement algorithms
- **Removal** algorithms
- Reversal algorithms
- **Fill** algorithms
- Rotation algorithms
- **Shuffling** algorithms
- **Set** algorithms
- Transformation algorithm
- Partition algorithms
- **Merge** algorithms
- Permutation algorithms
- Heap algorithms
- Lexicographical comparison algorithm

Topics

- Introduction to Classes
- Defining a Class Instance
- Constructors and Destructors
- The `this` Pointer
- Operator Overloading
- Class Templates

Classes and Objects

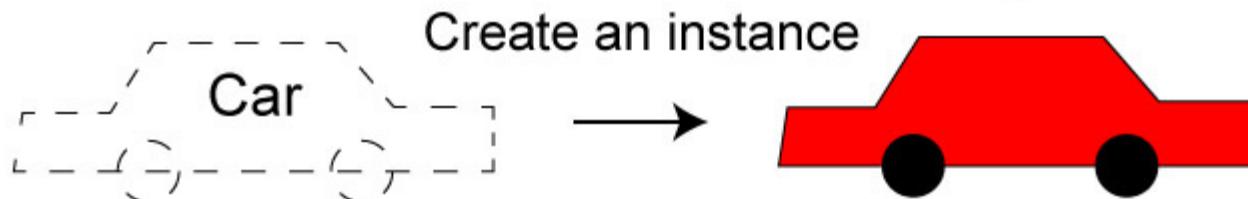
- A class is a template or prototype that describes what an object will be
- An object is an instance of a class.



Class Terminology

- Attributes
 - member variables of a class
- Methods or behaviors
 - member functions of a class

Class Object



Properties	Methods - behaviors
color	start()
price	backward()
km	forward()
model	stop()

Property values	Methods
color: red	start()
price: 23,000	backward()
km: 1,200	forward()
model: Audi	stop()

Introduction to Classes

- Objects are created from a class
- Format:

```
class ClassName
{
private:
    // Data members (variables)
public:
    // Member functions (methods)
};
```

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

class Rectangle
{
private:
    double width;
    double length;
public:
    void setWidth(double);
    void setLength(double);
    double getWidth() const;
    double getLength() const;
    double getArea() const;
};

int main()
{
    Rectangle r1;
    r1.setWidth(5.2);
    r1.setLength(4);
    cout<<r1.getArea()<<endl;
    return 0;
}
```

Access Specifiers

- Used to control access to members of the class
- **public**
 - can be accessed by functions outside of the class
- **private**
 - can only be called by or accessed by functions that are members of the class

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

class Rectangle
{
private:
    double width;
    double length;
public:
    void setWidth(double);
    void setLength(double);
    double getWidth() const;
    double getLength() const;
    double getArea() const;
};
```

More on Access Specifiers

- Can be listed in **any order** in a class
- Can appear **multiple** times in a class
- If not specified, the **default** is **private**

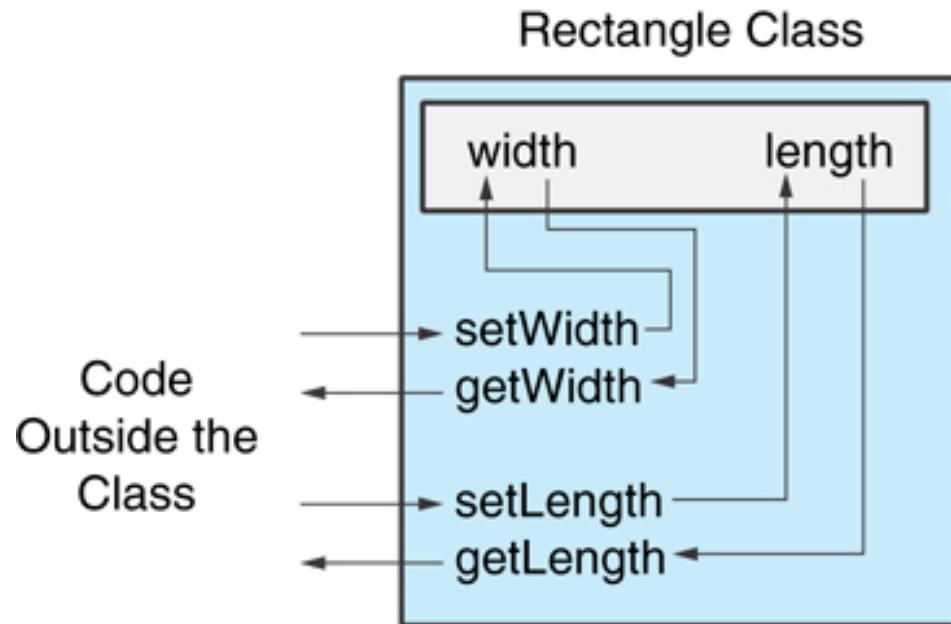
Why Have Private Members?

- Making **data members** **private** provides data protection
- Private data can be accessed only through **public** functions
- **Public** functions define the class's **public** interface

Using Private Member Functions

- A **private** member function can only be called by another member function
- It is used for internal processing by the class, not for use outside of the class

Code outside the class must use the class's public member functions to interact with the object.



Using const With Member Functions

- `const` appearing after the parentheses in a member function declaration specifies that **the function will not change any data in the called object.**

```
double getWidth( ) const;  
double getLength( ) const;  
double getArea( ) const;
```

Defining a Member Function

- When defining a member function:
 - Put prototype in class declaration
 - Define function using class name and **scope resolution operator** (::)

```
int Rectangle::setWidth(double w)
{
    width = w;
}
```

Accessors and Mutators

- Mutator (Setter): a member function that **stores** a value in a private member variable, or **changes** its value in some way
- Accessor (Getter): function that **retrieves** a value from a private member variable. Accessors **do not change** an object's data, so they should be marked `const`.

Separating Specification from Implementation

- *ClassName.hpp*
 - A header file, containing class specification.
 - for example, *Rectangle.hpp*
- *ClassName.cpp*
 - contain member function implementation
 - Then, *Rectangle.cpp* file should `#include` the class specification file

example3_1

> build

include

 ↳ Rectangle.hpp

 ↳ Rectangle.cpp

 ↳ run_rectangle.cpp

```
//Rectangle.hpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
private:
    double width;
    double length;
public:
    void setWidth(double);
    void setLength(double);
    double getWidth() const;
    double getLength() const;
    double getArea() const;
};

#endif // RECTANGLE_H
```

```
// Rectangle.cpp
#include <iostream>
#include "./include/Rectangle.hpp"
using namespace std;
void Rectangle::setWidth(double w) {
    if (w > 0) {
        width = w;
    }
    else {
        cout << "Invalid width\n";
        exit(EXIT_FAILURE);
    }
}
void Rectangle::setLength(double len) {
    if (len > 0) {
        length = len;
    }
    else {
        cout << "Invalid length\n";
        exit(EXIT_FAILURE);
    }
}
double Rectangle::getWidth() const { return
width; }
double Rectangle::getLength() const { return
length; }
double Rectangle::getArea() const { return
width * length; }
```

Separating Specification from Implementation

- Programs that use the class must `#include` the class specification file
 - compiler will link the member function implementations

```
// run_rectangle.cpp
#include <iostream>
#include "./include/Rectangle.hpp"

using namespace std;

int main()
{
    Rectangle r1;
    r1.setWidth(5.2);
    cout<<r1.getWidth()<<endl;

    r1.setLength(4);
    cout<<r1.getArea()<<endl;

    return 0;
}
```

Topics

- Introduction to Classes
- Defining a Class Instance/Object
- Constructors and Destructors
- The `this` Pointer
- Operator Overloading
- Class Templates

Defining an Instance/Object of a Class

- An object is an instance of a class
- Define instance/object

```
Rectangle r1, r2;
```

- Access members using **dot** operator:

```
r.setWidth(5.2);  
cout << r.getWidth();
```

- Do r1 and r2 share data?

```
#include <iostream>  
using namespace std;  
  
class Rectangle  
{  
private:  
    double width {0};  
    double length {0};  
public:  
    void setWidth(double);  
    void setLength(double);  
    double getWidth() const;  
    double getLength() const;  
    double getArea() const;  
};  
  
int main()  
{  
    Rectangle r1,r2;  
    r1.setWidth(5.2);  
    r1.setLength(4);  
    cout<<r1.getArea()<<endl;  
    return 0;  
}
```

Defining an Instance of a Class

- Compiler error if attempt to access private member using **dot** operator

Inline Member Functions

- Member functions can be also defined
 - `inline`: in class declaration
- Inline appropriate for short function bodies:

```
int getWidth() const  
{ return width; }
```

Rectangle Class with Inline Member Functions

```
// example3_2/Rectangle.hpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
private:
    double width;
    double length;
public:
    void setWidth(double);
    void setLength(double);
    double getWidth() const {return width;};
    double getLength() const {return length;};
    double getArea() const {return width * length;};
};

#endif // RECTANGLE_H
```

Tradeoffs – Inline vs. Regular Member Functions

- Regular functions
 - when called, compiler stores return address of call, allocates memory for local variables, etc.
- Inline functions
 - Code for an inline function is copied into program in place of call
 - larger executable program, but no function call overhead, hence faster execution

Member Function Overloading

- Member functions can be overloaded:

```
void setCost(double);
```

```
void setCost(char *);
```

Topics

- Introduction to Classes
- Defining a Class Instance
- Constructors and Destructors
- The `this` Pointer
- Operator Overloading
- Class Templates

Constructor

- A member function that is automatically called
when an object is created
- **Purpose is to initialize an object**
- Constructor has **the same name** as the class
- Can be overloaded
- Has no return type

```
// example3_3/Rectangle.hpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
private:
    double width{};
    double length{};
public:
    Rectangle();
    Rectangle(double, double);
    ~Rectangle();

    void setWidth(double);
    void setLength(double);
    double getWidth() const {return width;};
    double getLength() const {return length;};
    double getArea() const {return width * length;};
};

#endif // RECTANGLE_H
```

```
// example3_3/Rectangle.cpp
#include <iostream>
#include "Rectangle.hpp"
using namespace std;

Rectangle::Rectangle()
{
    width = 0.0;
    length = 0.0;
}

Rectangle::Rectangle(double w, double len)
{
    width = w;
    length = len;
    // this->length = len;
}
```

In-Place Initialization

- In-place initialization
 - If you are using C++11 or later, you can initialize a member variable in its declaration statement, just as you can with a regular variable.
- Here is an example:

```
class Rectangle
{
private:
    double width {0.0};
    double length {0.0};
public:
    Public member functions appear here...
};
```

Default Constructors

- A **default constructor** is a constructor that takes **no arguments**.
- If you write a class with **no constructor at all**, C++ compiler will create a default constructor for you, one that does nothing.
- A simple definition of an object (with no arguments) will call the default constructor:

```
Rectangle r;
```

Passing Arguments to Constructors

- To create a constructor that takes arguments:
 - indicate parameters in prototype:

```
Rectangle(double, double);
```

- Use parameters in the definition:

```
Rectangle::Rectangle(double w, double  
len)  
{  
    width = w;  
    length = len;  
}
```

Passing Arguments to Constructors

- You can pass arguments to the constructor when you create an object:

```
Rectangle r(10, 5);
```

More About Default Constructors

- If all of a constructor's **parameters** have **default arguments**, then it is a **default constructor**.
 - For example:

```
Rectangle(double = 0, double = 0);
```

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```

Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has **NO** default constructor.
- When this is the case, you must pass the required arguments to the constructor when creating an object.

Destructors

- A member function automatically called when an object is destroyed
- Destructor name is `~classname`
 - *e.g.*, `~Rectangle`
- Has no return type; takes no arguments
- **Only one destructor per class,**
 - *i.e.*, it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it

```
Rectangle::~Rectangle()
{
    cout<<"~Rectangle"<<endl;
}
```

Overloading Constructors

- A class can have more than one constructor
- Overloaded constructors in a class must have different parameter lists:

```
Rectangle();
```

```
Rectangle(double);
```

```
Rectangle(double, double);
```

Constructor Delegation

- In C++ , it is possible for **one constructor** to call **another constructor** in the same class.
- This is known as *constructor delegation*.

```
// 3_2.cpp
#include <iostream>
using namespace std;
class Rectangle {
private:
    int width, height;
public:
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    Rectangle() : Rectangle(1, 1) {
        cout << "Delegating Constructor Called!" << endl;
    }
    void showArea() {
        cout << "Rectangle Area: " << width * height << endl;
    }
};
int main() {
    Rectangle r1;           // Calls Default Constructor → Delegates to Parameterized
                           // Constructor
    Rectangle r2(5, 3);    // Calls Parameterized Constructor directly
    r1.showArea();
    r2.showArea();
    return 0;
}
```

Constructor Initialization List

- Syntax

```
ClassName(value1, value2) : member1(value1), member2(value2), ... { }
```

```
// 3_2.cpp
#include <iostream>
using namespace std;

class Rectangle {
private:
    int width, height;

public:
    // Parameterized Constructor (Single Point of Initialization)
    // Rectangle(int w, int h) {
    //     width = w;
    //     height = h;
    //     cout << "Parameterized Constructor Called!" << endl;
    // }
    Rectangle(int w, int h) : width(w), height(h) {}

    // Default Constructor Delegating to Parameterized Constructor
    Rectangle() : Rectangle(1, 1) {
        cout << "Delegating Constructor Called!" << endl;
    }
}
```

Only One Default Constructor and One Destructor

- Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters

```
Square () ;  
Square (int = 0) ; // will not compile
```

- Since a destructor takes no arguments, there can only be one destructor for a class

Correct?

```
// example3_3/Rectangle.hpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
private:
    double width{};
    double length{};
public:
    Rectangle();
    Rectangle(double, double);
    Rectangle(double w=0 , double len=0);

    ~Rectangle();

    void setWidth(double);
    void setLength(double);
    double getWidth() const {return width;};
    double getLength() const {return length;};
    double getArea() const {return width * length;};
};

#endif // RECTANGLE_H
```

Topics

- Introduction to Classes
- Defining a Class Instance
- Constructors and Destructors
- **The `this` Pointer**
- Operator Overloading
- Class Templates

The `this` Pointer

- `this`
 - an **implicit pointer** available inside all **non-static member** functions of a class
- Features
 - **Automatically available** in all non-static member functions.
 - **Points to the calling object's memory address.**
 - Can be used to **access members explicitly** inside a class.

Using this to Access Data Members

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

class Rectangle {
private:
    int width, height;

public:
    Rectangle(int width, int height) {
        // Using 'this' to differentiate between local and member variables
        this->width = width;
        this->height = height;
    }
    void display() {
        cout << "Width: " << this->width << ", Height: " << this->height << endl;
    }
};

int main() {
    Rectangle r1(10, 5);
    r1.display();
    return 0;
}
```

Topics

- Introduction to Classes
- Defining a Class Instance
- Constructors and Destructors
- The `this` Pointer
- Operator Overloading
- Class Templates

Operator Overloading

- Operators such as `=`, `+`, and others can be redefined when used with objects of a class
- The name of the function for the overloaded operator is `operator` followed by the operator symbol, e.g.,

`operator+` to overload the `+` operator, and
`operator=` to overload the `=` operator

```
ReturnType operatorSymbol(Parameters) {  
    // Overloaded operator implementation  
}
```

```

// 3_5_operatorOverloading.cpp
#include <iostream>
using namespace std;

class MyComplex
{
private:
    double px, py;
public:
    MyComplex(const double x, const double y) : px(x), py(y) {}
    MyComplex operator-(const MyComplex &p1) const;
    friend MyComplex operator+(const MyComplex &p1, const MyComplex &p2);
    friend ostream& operator<<(ostream &out, const MyComplex &p1);

    friend bool operator==(const MyComplex &p1, const MyComplex &p2);
};

int main()
{
    MyComplex c1(1, 2), c2(1, -4);
    cout << c1 << endl;
    cout << c2 << endl;
    MyComplex c3 = c1 + c2;
    cout << c3 << endl;
    MyComplex c4 = c1 - c2;
    cout << c4 << endl;
    return 0;
}

```

- Invoking an Overloaded Operator
- Return an object

```
MyComplex MyComplex::operator-(const MyComplex &p1) const
{
    return MyComplex(px - p1.px, py - p1.py);
}
MyComplex operator+(const MyComplex &p1, const MyComplex &p2)
{
    return MyComplex(p1.px + p2.px, p1.py + p2.py);
}
ostream& operator<<(ostream &out, const MyComplex &p1)
{
    out << "("<<p1.px<<","<<p1.py<<")";
    return out;
}
bool operator==(const MyComplex &p1, const MyComplex &p2)
{
    return (p1.px == p2.px && p1.py == p2.py);
}
```

Overloading Types of Operators

- Overloaded stream operators `>>`, `<<`
 - must return reference to `istream`, `ostream` objects and
 - take `istream`, `ostream` objects as parameters

```
ostream& operator<<(ostream &out, const MyComplex &p1)
{
    out << "("<<p1.px<<","<<p1.py<<")";
    return out;
}
```

Overloading Types of Operators

- Overloaded relational operators should return a `bool` value

```
bool operator==(const MyComplex &p1, const MyComplex &p2)
{
    return (p1.px == p2.px && p1.py == p2.py);
}
```

Overloaded [] Operator

- Can create classes that behave like arrays, provide bounds-checking on subscripts
- Must consider constructor, destructor
- operator[] should return a reference (&) if the elements need to be modified.

```
// 3_6.cpp
#include <iostream>
using namespace std;

class Rectangle {
private:
    int width, height;

public:
    Rectangle(int w, int h) : width(w), height(h) {}
    // Overload [] operator for read and write access
    int& operator[](int index) {
        if (index == 0) return width;
        if (index == 1) return height;
        cout << "Index out of bounds!" << endl;
        exit(1);
    }
};

int main() {
    Rectangle rect(10, 5);
    cout << "Original Width: " << rect[0] << ", Height: " << rect[1] << endl;
    // Modifying width and height
    rect[0] = 20;
    rect[1] = 10;
    cout << "Modified Width: " << rect[0] << ", Height: " << rect[1] << endl;
    return 0;
}
```

Notes on Overloaded Operators

- Can change meaning of an operator
- Cannot change **the number of operands** of the operator
- Only certain operators can be overloaded.
Cannot overload the following operators:

? : . . * :: sizeof

Topics

- Introduction to Classes
- Defining a Class Instance
- Constructors and Destructors
- The `this` Pointer
- Operator Overloading
- Class Templates

Class Templates

- A class template in C++ allows us to create **generic classes** that can work with **any data type**.
- Unlike functions, classes are instantiated by supplying the type name (int, double, string, etc.) at object definition

```
// class_template.cpp
#include <iostream>
using namespace std;

template <typename T1=int, typename T2=int>
class Rectangle {
private:
    T1 width;
    T2 height;

public:
    Rectangle(T1 w, T2 h) : width(w), height(h) {}

    void display() {
        cout << "Width: " << width << ", Height: " << height << endl;
    }
};

int main() {
    Rectangle<int, float> r(10, 5.5); // Width as int, Height as float
    r.display();
    return 0;
}
```

2800ICT

Object Oriented Programming

Object Oriented Design - I

Review Week 4

- Introduction to Classes
- Defining a Class Instance
- Constructors and Destructors
- The `this` Pointer
- Operator Overloading
- Class Templates

Topics

- Classes
 - Static Members and Friend Functions
- Copying Objects
 - Copy Constructor/Assignment
 - Lvalue & Rvalue
 - Move Constructor/Assignment
- Unified Modelling Language (UML)
- Class Aggregation

Topic

Classes – Static Members and Friend Functions

Static Members

- Instance variable
 - a member variable in a class.
 - Each object has its own copy.
- static variable
 - one variable shared among all objects of a class
- static member function
 - can be called before any objects are defined
 - can be used to **only** access static member variable;

static member variable

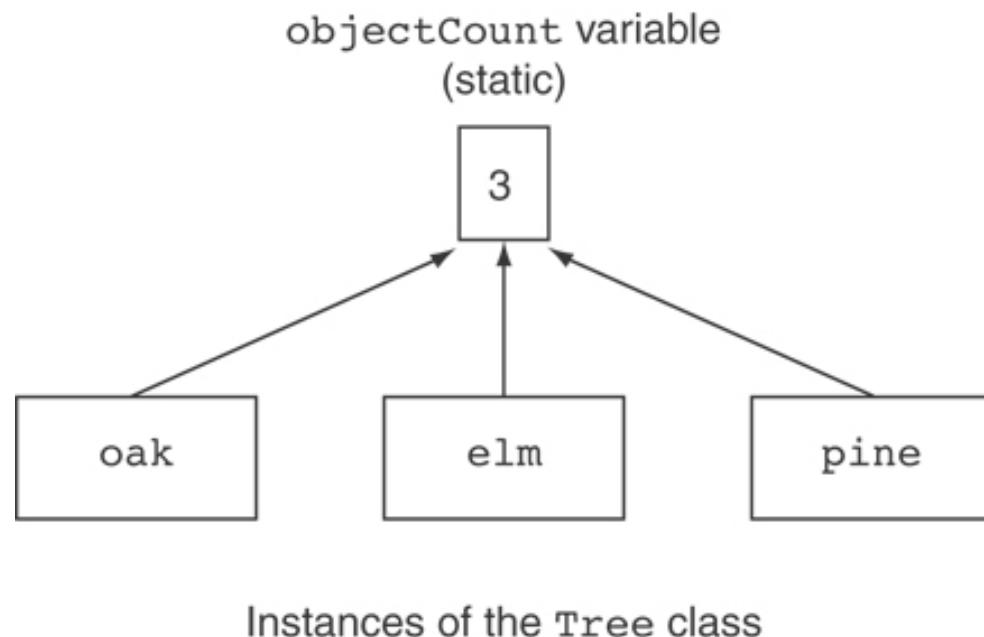
```
class Tree
{
private:
    static int objectCount; // declare Static member variable.
public:
    // Constructor
    Tree()
    {
        objectCount++;
    }

    // Accessor function for objectCount
    int getObjectCount() const
    {
        return objectCount;
    }
};

// Definition of the static member variable
int Tree::objectCount = 0;
```

```
// This program demonstrates a static member variable.  
#include <iostream>  
#include "Tree.h"  
using namespace std;  
  
int main()  
{  
    // Define three Tree objects.  
    Tree oak;  
    Tree elm;  
    Tree pine;  
  
    // Display the number of Tree objects we have.  
    cout << "We have " << pine.getObjectCount()  
        << " trees in our program!\n";  
    return 0;  
}
```

Three Instances of the Tree Class, But Only One objectCount Variable



static member function

- Declared with static before return type:

```
static int getObjectCount() const  
{ return objectCount; }
```

- Static member functions **can only access** static member data
- Can be called independent of objects:

```
int num = Tree::getObjectCount();
```

```
#include <iostream>

// Tree class
class Tree
{
private:
    static int objectCount;      // Static member variable.
public:
    // Constructor
    Tree()
        { objectCount++; }

    // Accessor function for objectCount
    static int getObjectCount()
        { return objectCount; }
};

// Definition of the static member variable, written
// outside the class.
int Tree::objectCount = 0;

int main()
{
    std::cout << "There are " << Tree::getObjectCount() << " objects.\n";
}
```

Friends of Classes

- Friend
 - a function or class that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with `friend` keyword in the function prototype

friend Function Declarations

- Stand-alone function:

```
friend void setAVal(int& intVal, int);  
// declares setAVal function to be  
// a friend of this class
```

- Member function of another class:

```
friend void SomeClass::setNum(int num)  
// setNum function from SomeClass  
// class is a friend of this class
```

friend Class Declarations

- Class as a friend of a class:

```
class FriendClass
{
    ...
};

class NewClass
{
public:
    friend class FriendClass;
    // declares entire class FriendClass as a friend
    // of this class
    ...
};
```

Topic

Copy & Move Objects

- std::shared_ptr Smart Pointer
- Copy Constructor
- Copy Assignment
- Lvalue & Rvalue
- Move Constructor
- Move Assignment

`std::shared_ptr` Smart Pointer

- What is `std::shared_ptr`
 - A smart pointer that manages shared ownership of a dynamically allocated object.
 - When the last `shared_ptr` to an object is destroyed, the object is automatically deleted.
- Why use it
 - **Automatic** memory management
 - No need to call `delete`
 - **Prevents memory leaks**
 - **Safer than raw pointers**

std::shared_ptr Smart Pointer

- Include the header

```
#include <memory>
```

- Create a shared_ptr object and allocate memory

```
std::shared_ptr<int> p1 = std::make_shared<int>(100);
```

- Share ownership

```
std::shared_ptr<int> p2 = p1;  
// Now both p1 and p2 share ownership
```

- Check use count

```
std::cout << p1.use_count();  
// Shows how many shared_ptr instances point to the same  
object
```

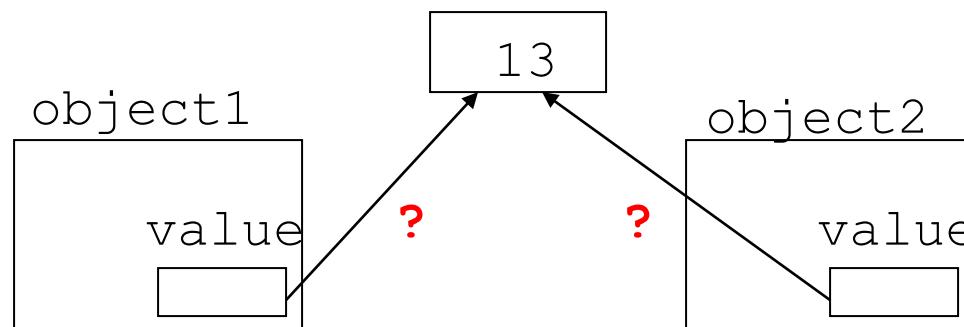
Object Memberwise Assignment

- Assigning one obj to another using `=` invokes **default memberwise assignment if no custom operator** is defined.
 - *e.g.*, `Rectangle r2 = r1;`
 - *or*, `r2 = r1;`
 - copy all member values from `r1` and assign to the corresponding member variables of `r2`
- For user-defined types, you can override this behavior by defining your own
 - copy/move constructor
 - copy/move assignment.

Case Study

- What happens when a class has a **std::shared_ptr member** and the **default constructor** is used.

```
int main()
{
    SomeClass object1(5);
    SomeClass object2 = object1;
    object2.setVal(13);
    cout << object1.getVal(); // print 13
}
```



Copy Constructor

- **Special constructor** used when a newly created object is initialized to the data of another object of same class
- Called when
 - Create a new object from an existing one

```
SomeClass object2 = object1;
```
- Default copy constructor copies field-to-field
 - works fine in many cases
 - All members are value types (e.g., int, double, std::string)
 - All members are STL containers (e.g., vector or std::array)
 - Don't work in the Case Study
 - Need deep copies for resource

Copy Constructor

- Syntax

```
ClassName(const ClassName& other);
```

- Example

```
#include <iostream>
#include <memory>
using namespace std;
class SomeClass
{
private:
    shared_ptr<int> value;

public:
    SomeClass(int val = 0) : value(make_shared<int>(val)) {}

    // Copy Constructor (Deep Copy)
    SomeClass(const SomeClass &other)
    {
        value = make_shared<int>(*other.value); // deep copy of value
    }

    int getVal() const { return *value; }
    void setVal(int val) { *value = val; }
};
```

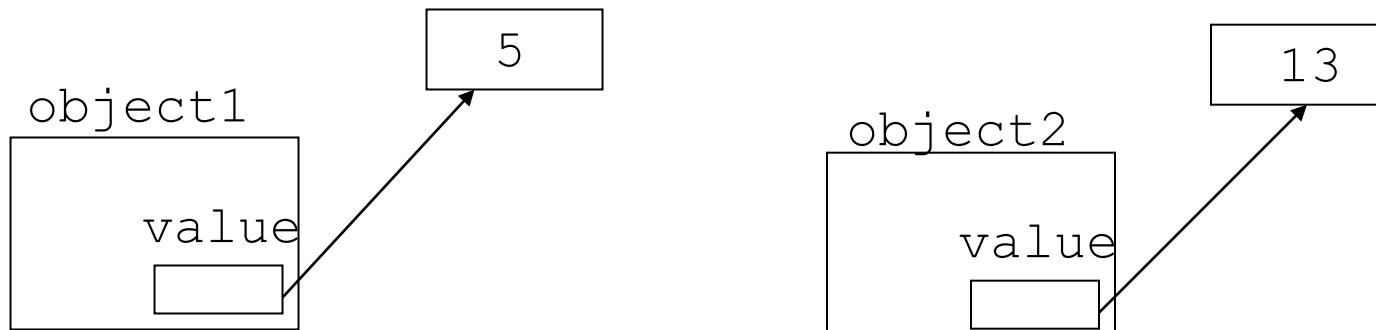
Copy Constructor

- Each object now points to separate dynamic memory:

```
int main() {
    SomeClass object1(5);
    SomeClass object2 = object1; // deep copy

    object2.setVal(13);

    cout << "object1.getVal(): " << object1.getVal() << endl; // prints 5
    cout << "object2.getVal(): " << object2.getVal() << endl; // prints 13
}
```



Copy Assignment

- Defines how an existing object is assigned the values of another object of the same type.
- Called when
 - Assign an existing object to a new one

```
object2 = object1;
```
- If not explicitly defined, C++ provides a **default version** that performs **memberwise copy**.
 - works fine in many cases
 - All members are value types (e.g., int, double, std::string)
 - All members are STL containers (e.g., vector or std::array)
 - Don't work in cases
 - Need deep copies for resource

Copy Assignment

- Syntax

```
ClassName& operator=(const ClassName& other);
```

- Example

```
#include <iostream>
#include <memory>
using namespace std;
class SomeClass
{
private:
    shared_ptr<int> value;

public:
    SomeClass(int val = 0) : value(make_shared<int>(val)) {}
    SomeClass(const SomeClass &other)
    {
        value = make_shared<int>(*other.value);
    }
    SomeClass &operator=(const SomeClass &other)
    {
        if (this != &other)
        {
            value = make_shared<int>(*other.value);
        }
        return *this;
    }
    int getVal() const { return *value; }
    void setVal(int val) { *value = val; }
};
```

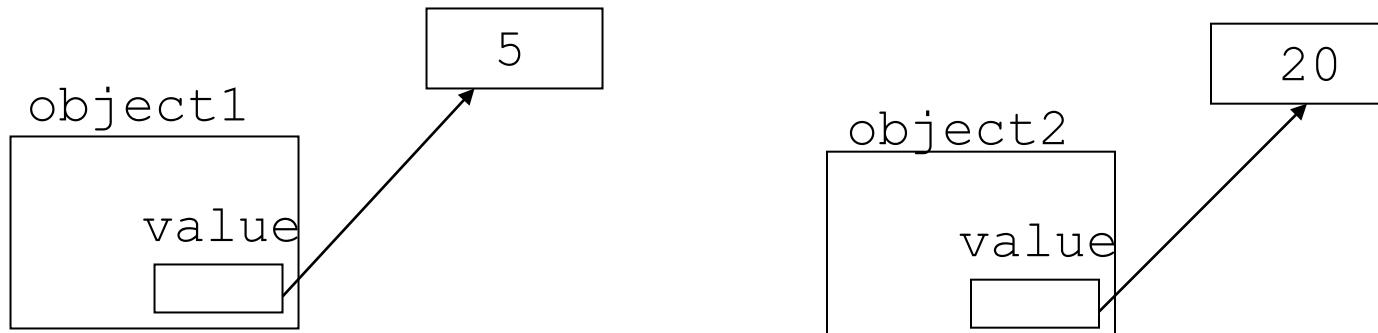
Copy Assignment

- Each object now points to separate dynamic memory:

```
int main()
{
    SomeClass object1(5);
    SomeClass object2(10);

    object2 = object1; // Copy assignment (deep copy)
    object2.setVal(20); // Modify object2 only

    cout << "object1: " << object1.getVal() << endl; // prints 5
    cout << "object2: " << object2.getVal() << endl; // prints 20
}
```



Lvalues and Rvalues

- Two types of **values** stored in memory during the execution of a program:
 - ***Lvalues***
 - ***Rvalue***
- ***Lvalue***
 - Values that **persist** beyond the statement that created them,
 - and **have names** that make them accessible to other statements in the program.
- ***Rvalue***
 - Values that are **temporary**,
 - and **cannot be accessed beyond the statement** that created them.

Lvalues and Rvalues

Which are *Lvalues* and *Rvalues*?

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

int main() {
    int x = 10;
    int y = 20;
    int z = x + y;
    z = x+20;
    cout << "The value of z is: " << z << endl;
    return 0;
}
```

Lvalue & Rvalue References

- ***Lvalue Reference* ($T\&$)**

- Binds to Lvalues (named variables with a memory address).
- Used in normal variable references, function parameters, etc.

```
int x = 10;
int& ref = x; // ref is an Lvalue reference
ref = 20;      // modifies x
```

Lvalue & Rvalue References

- **Rvalue Reference (T&&)**

- Binds to Rvalues (temporary objects or literals).
- Enables **move semantics** and efficient resource transfer.

```
int&& rref = 30;    // OK: binds to rvalue

// int&& w_r = x;
// int&& w_r = x+20;

void process(int& x) { cout << "Lvalue\n"; }
void process(int&& x) { cout << "Rvalue\n"; }

int a = 5;
process(a);          // call which function?
process(10);         // call which function?
```

Move Constructor

- **Special constructor** that transfers ownership of resources from a temporary (rvalue) object to a new object, without copying.
- Called when an object is constructed from a temporary (rvalue)
 - `ClassName b = std::move(a);`
 - `return obj;` (may invoke move constructor when returning by value)

Move Constructor

- Default one works fine in many cases
 - All members have move constructors
 - The class has no user-defined:
 - Copy constructor
 - Copy assignment operator
 - Destructor
 - (If **any are user-defined**, move constructor is not generated by default)
- Default one doesn't work in
 - Need deep transfer resource
 - want **custom behavior**

Move Constructor

- Syntax

```
ClassName(ClassName&& other);
```

- Example

```
#include <iostream>
#include <memory>
using namespace std;
class SomeClass
{
private:
    shared_ptr<int> value;
public:
    SomeClass(int val = 0);
    SomeClass(const SomeClass &other);

    //  Move Constructor
    SomeClass(SomeClass &&other);
    int getVal() const;
    void setVal(int val);
};
```

```
SomeClass::SomeClass(SomeClass &&other)
{
    value = move(other.value);
    // Transfer ownership
    // other.value becomes nullptr
}
```

```
// Main function to demonstrate move constructor
int main()
{
    SomeClass original(42);

    // Move constructor is called here
    SomeClass moved = move(original);

    cout << "original: " << original.getVal() << endl;
    // Output: -1 (moved-from)
    cout << "moved: " << moved.getVal() << endl;
}
```

Move Assignment

- transfers resources from an existing temporary (rvalue) object to another already-existing object
- Called when
 - Assign an rvalue to an existing object

```
MyClass a, b;  
a = std::move(b); // calls move assignment
```

Move Assignment

- Default one works fine in many cases
 - All members have move assignment operators
 - e.g., std::string, shared_ptr, STL containers
 - The class has no user-defined:
 - Copy constructor
 - Copy assignment operator
 - Destructor
 - (If **any are user-defined**, move constructor is not generated by default)
- Default one doesn't work in
 - Need deep transfer resource
 - want **custom behavior**

Move Assignment

- Syntax

```
ClassName& operator=(ClassName&& other);
```

- Example

```
#include <iostream>
#include <memory>
using namespace std;
class SomeClass
{
private:
    shared_ptr<int> value;
public:
    SomeClass(int val = 0);
    SomeClass(const SomeClass &other);
    SomeClass(SomeClass &&other);

    //  Move Assignment Operator
    SomeClass &operator=(SomeClass &&other);

    int getVal() const;
    void setVal(int val);
};
```

```
SomeClass
&SomeClass::operator=(SomeClass
&&other)
{
    if (this != &other)
    {
        value = move(other.value); // Transfer ownership
    }
    return *this;
}
```

```
int main()
{
    SomeClass a(100);
    SomeClass b(200);

    cout << "Before move assignment:\n";
    cout << "a = " << a.getVal() << ", b = " << b.getVal() << endl;

    b = move(a); // Calls move assignment operator

    cout << "After move assignment:\n";
    cout << "a = " << a.getVal() << " (moved-from)" << endl;
    cout << "b = " << b.getVal() << endl;
}
```

Best Practice

- If your class contains smart pointers,
- but you want each object to have its own independent copy of the underlying data (instead of sharing ownership),
- you should define a custom copy/move constructor/assignment to ensure deep copying behavior.

Topic

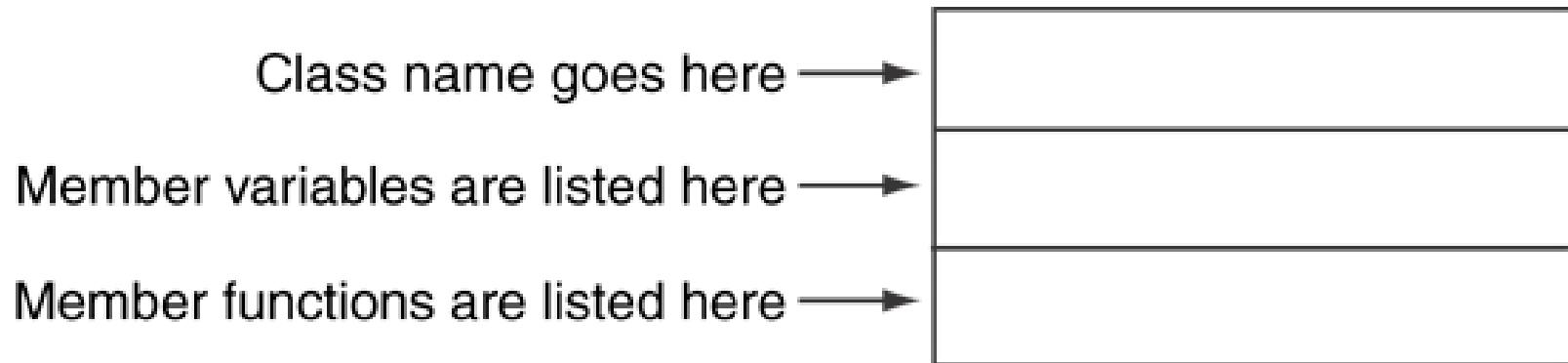
Unified Modelling Language

The Unified Modelling Language

- *UML* stands for *Unified Modelling Language*.
- The UML provides a set of standard diagrams for graphically depicting object-oriented systems

UML Class Diagram

- A UML diagram for a class has three main sections.



UML Access Specification Notation

- In UML you indicate
 - a private member with a minus (-)
 - a public member with a plus(+).

Rectangle
- width: double - length: double
+ Rectangle(x: double=0, y:double=0): + ~Rectangle(): + setWidth(width: double): void + setLength(length: double): void + getWidth() const: double + getLength() const: double + getArea() const: double

UML Data Type Notation

- To indicate the **data type** of a member variable
 - place a colon followed by the name of the data type after the name of the variable.
 - `width : double`
 - `length : double`

Rectangle
- <code>width: double</code>
- <code>length: double</code>
+ <code>Rectangle(x: double=0, y:double=0)</code> :
+ <code>~Rectangle()</code> :
+ <code>setWidth(width: double): void</code>
+ <code>setLength(length: double): void</code>
+ <code>getWidth() const: double</code>
+ <code>getLength() const: double</code>
+ <code>getArea() const: double</code>

UML Parameter Type Notation

- To indicate the data type of a function's parameter variable
 - place a colon followed by the name of the data type after the name of the variable.

+ `setWidth(w : double)`

Rectangle
- width: double
- length: double
+ Rectangle(x: double=0, y:double=0): void
+ ~Rectangle(): void
+ setWidth(width: double): void
+ setLength(length: double): void
+ getWidth() const: double
+ getLength() const: double
+ getArea() const: double

UML Function Return Type Notation

- To indicate the data type of a function's return value
 - place a colon followed by the name of the data type after the function's parameter list.
- + `setWidth(w : double) : void`

Rectangle
- width: double
- length: double
+ Rectangle(x: double=0, y:double=0): void
+ ~Rectangle(): void
+ setWidth(width: double): void
+ setLength(length: double): void
+ getWidth() const: double
+ getLength() const: double
+ getArea() const: double

```
class Rectangle
{
private:
    double width;
    double length;
public:
    Rectangle(double width=0, double length=0);
    ~Rectangle();
    bool setWidth(double);
    bool setLength(double);
    double getWidth() const;
    double getLength() const;
    double getArea() const;
    friend ostream& operator<<(ostream& os, const Rectangle& rect);
};
```

Rectangle
- width: double
- length: double
+ Rectangle(x: double=0, y:double=0):
+ ~Rectangle():
+ setWidth(width: double): void
+ setLegnht(length: double): void
+ getWidth() const: double
+ getLegnht() const: double
+ getArea() const: double
<<friend>> operator<<(os: ostream&, reect: const Rectangle&):ostream&

Topic

Class Aggregation

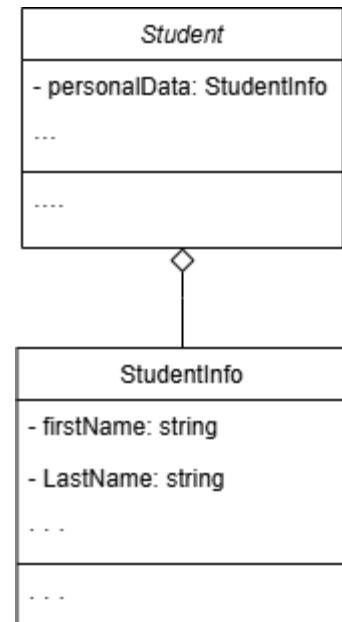
Aggregation

- Aggregation
 - a class is a member of a class
- Supports the modeling of ‘has a’ relationship between classes
 - enclosing class ‘has a’ enclosed class
- Same notation as for structures within structures

Aggregation

```
class StudentInfo
{
private:
    string firstName, LastName;
    string address, city, state, zip;
    ...
};

class Student
{
private:
    StudentInfo personalData;
    ...
};
```



2800ICT

Object Oriented Programming

Object Oriented Design 2

Review Week 5

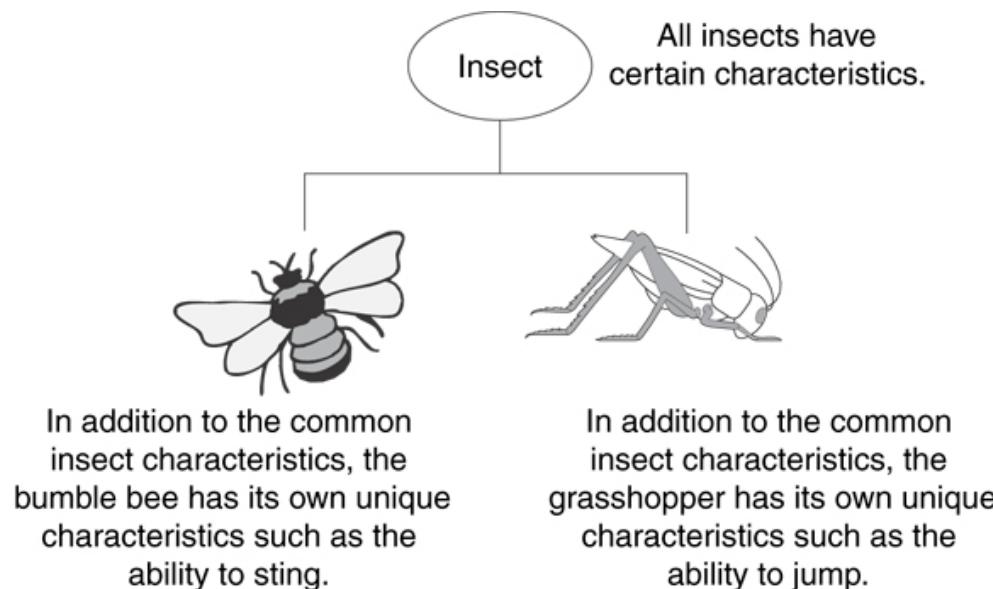
- Classes
 - Static Members and Friend Functions
- Copying Objects
 - Copy Constructor/Assignment
 - Lvalue & Rvalue
 - Move Constructor/Assignment
- Unified Modelling Language (UML)
- Class Aggregation

Topics

- Inheritance
 - Class Inheritance
 - Protected Members and Class Access
 - Constructors and Destructors in Base and Derived Classes
 - Redefining Base Class Functions
- Polymorphism
 - Virtual Member Functions
- Abstract Base Class
 - Pure Virtual Functions
 - Multiple Inheritance

What Is Inheritance?

- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class



The "is a" Relationship

- Inheritance establishes an "is a" relationship between classes.
 - A poodle is a dog
 - A car is a vehicle
 - A flower is a plant
 - A football player is an athlete

Inheritance – Terminology and Notation

- Base class (or parent) – inherited from
- Derived class (or child) – inherits from the base class
- Notation:

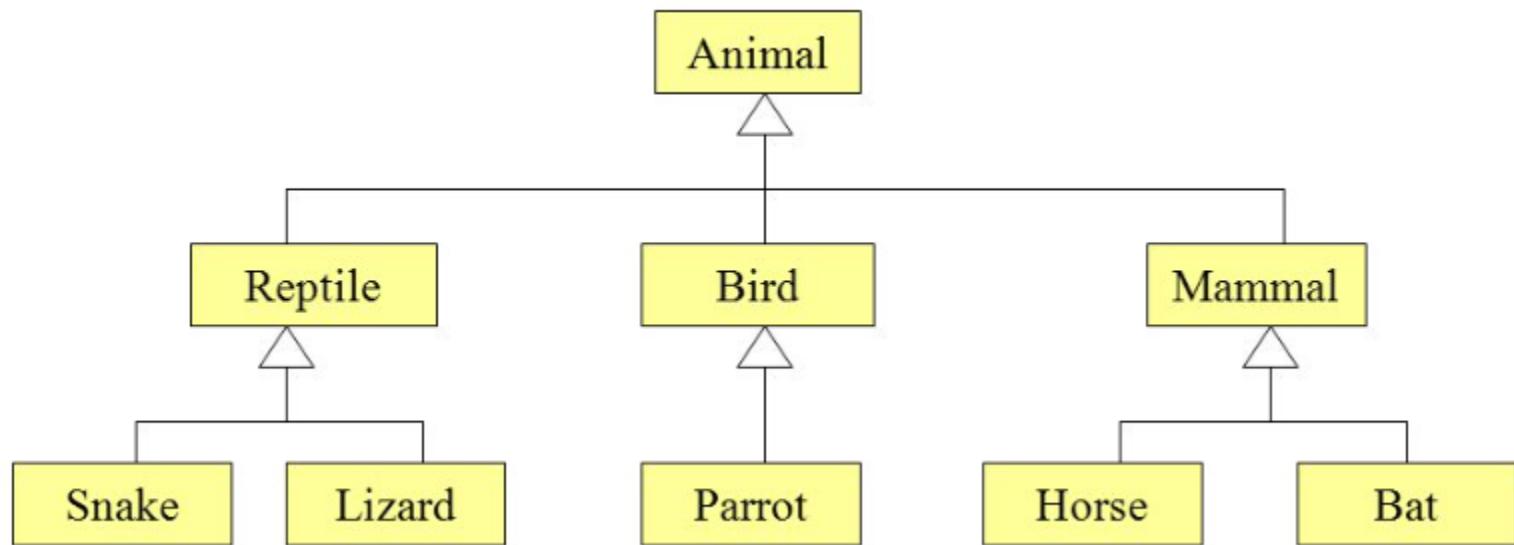
```
class Student          // base class
{
    ...
};

class UnderGrad : public Student
{
    ...                      // derived class
};

};
```

UML: Class Hierarchies

- A base class can be derived from another base class.



Back to the ‘is a’ Relationship

- An object of a derived class 'is a(n)' object of the base class
- Example:
 - an UnderGrad **is a** Student
 - a Mammal **is an** Animal
- A derived object has all of the characteristics of the base class

What Does a Child Have?

An object of the derived class **has**:

- all members defined in child class
- all members declared in parent class

An object of the derived class can **use**:

- all `public` members defined in child class
- all `public` members defined in parent class

Topics

- Inheritance
 - Class Inheritance
 - Protected Members and Class Access
 - Constructors and Destructors in Base and Derived Classes
 - Redefining Base Class Functions
- Polymorphism
 - Virtual Member Functions
- Abstract Base Class
 - Pure Virtual Functions
 - Multiple Inheritance

Protected Members and Class Access

- Member access specification
 - protected like private, but accessible by objects of derived class
- Class access specification
 - determines how private, protected, and public members of base class are inherited by the derived class

Inheritance vs. Access: public

```
class Grade
```

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```

```
class Test : public Grade
```

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

When Test class inherits
from Grade class using
public class access, it
looks like this: 

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);  
void setScore(float);  
float getScore();  
float getLetter();
```

Inheritance vs. Access: **protected**

```
class Grade
```

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```

```
class Test : protected Grade
```

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

When Test class inherits
from Grade class using
protected class access, it
looks like this: —————→

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

protected members:

```
void setScore(float);  
float getScore();  
float getLetter();
```

Inheritance vs. Access: **private**

```
class Grade
```

private members:

```
char letter;  
float score;  
void calcGrade();
```

public members:

```
void setScore(float);  
float getScore();  
char getLetter();
```

```
class Test : private Grade
```

private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;
```

public members:

```
Test(int, int);
```

When Test class inherits
from Grade class using
private class access, it
looks like this:



private members:

```
int numQuestions;  
float pointsEach;  
int numMissed;  
void setScore(float);  
float getScore();  
float getLetter();
```

public members:

```
Test(int, int);
```

Summary: Inheritance vs. Access

Base class members

```
private: x  
protected: y  
public: z
```

private
base class

How inherited base class
members
appear in derived class

```
x is inaccessible  
private: y  
private: z
```

```
private: x  
protected: y  
public: z
```

protected
base class

```
x is inaccessible  
protected: y  
protected: z
```

```
private: x  
protected: y  
public: z
```

public
base class

```
x is inaccessible  
protected: y  
public: z
```

Topics

- Inheritance
 - Class Inheritance
 - Protected Members and Class Access
 - Constructors and Destructors in Base and Derived Classes
 - Redefining Base Class Functions
- Polymorphism
 - Virtual Member Functions
- Abstract Base Class
 - Pure Virtual Functions
 - Multiple Inheritance

Constructors & Destructors in Base & Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created,
 - **the base class's constructor is executed first,**
 - followed by the derived class's constructor
- When an object of a derived class is destroyed,
 - its destructor is called first,
 - then **that of the base class**

```
BaseClass constructor.  
DerivedClass constructor.  
The program is now going to end.  
DerivedClass destructor.  
BaseClass destructor.
```

```
// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
#include <iostream>
using namespace std;

//*****
// BaseClass declaration      *
//*****
class BaseClass
{
public:
    BaseClass() // Constructor
    {
        cout << "BaseClass constructor.\n";
    }

    ~BaseClass() // Destructor
    {
        cout << "BaseClass destructor.\n";
    }
};
```

```
//*****
// DerivedClass declaration      *
//*****  
class DerivedClass : public BaseClass  
{  
public:  
    DerivedClass() // Constructor  
    {  
        cout << "    DerivedClass constructor.\n";  
    }  
  
    ~DerivedClass() // Destructor  
    {  
        cout << "    DerivedClass destructor.\n";  
    }  
};  
  
int main()  
{  
    // cout << "We will now define a DerivedClass object.\n";  
  
    DerivedClass object;  
  
    cout << "        The program is now going to end.\n";  
    return 0;  
}  
  
                                BaseClass constructor.  
                                DerivedClass constructor.  
                                The program is now going to end.  
                                DerivedClass destructor.  
                                BaseClass destructor.
```

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Must be done if base class has no default constructor
 - If no base constructor is specified, **the default base constructor** will be used.
- Specify arguments to base constructor on derived constructor heading:

```
DerivedClass::DerivedClass(int side) : BaseClass(side, side) {}
```

Passing Arguments to Base Class Constructor

```
derived class constructor           base class constructor  
                                {  
DerivedClass::DerivedClass(int side) : BaseClass(side, side) {}  
                                }  
                                derived constructor      base constructor  
                                parameter             parameters
```

```

#include <iostream>
using namespace std;
class Base
{
public:
    Base(int x); // Constructor declaration
    void display(); // Member function declaration

private:
    int baseValue;
};

class Derived : public Base
{
public:
    Derived(int x = 0, int y = 0);
    void display();

private:
    int derivedValue;
};

int main()
{
    Derived derivedObj(10, 20);
    derivedObj.display();
    return 0;
}

```

```

Base::Base(int x) : baseValue(x)
{
    cout << "Base constructor" << endl;
}

void Base::display()
{
    cout << "Base value: " << baseValue << endl;
}

Derived::Derived(int x, int y) : Base(x), derivedValue(y)
{
    cout << "Derived constructor" << endl;
}

void Derived::display()
{
    Base::display(); // Call base class display
    cout << "Derived value: " << derivedValue << endl;
}

```

Topics

- Inheritance
 - Class Inheritance
 - Protected Members and Class Access
 - Constructors and Destructors in Base and Derived Classes
 - Redefining Base Class Functions
- Polymorphism
 - Virtual Member Functions
- Abstract Base Class
 - Pure Virtual Functions
 - Multiple Inheritance

Redefining Base Class Functions

- Redefining function
 - function in a derived class that has the *same name and parameter list* as a function in the base class
- Typically used to replace a function in base class with different actions in derived class

Redefining Base Class Functions

- Not the same as overloading
 - with overloading, parameter lists must be different
- Objects of base class use base class version of function;
- Objects of derived class use derived class version of function

Problem with Redefining

- Consider this situation:
 - Class BaseClass defines functions `x()` and `y()`. `x()` calls `y()`.
 - Class DerivedClass inherits from BaseClass and redefines function `y()`.
 - An object D of class DerivedClass is created and function `x()` is called.
 - When `x()` is called, which `y()` is used, the one defined in BaseClass or the the redefined one in DerivedClass?

BaseClass

```
void X(){Y();}  
void Y();
```

DerivedClass

```
void Y();
```

```
DerivedClass D;  
D.X();
```

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

class BaseClass
{
public:
    void x() { y(); }
    void y() { cout << "BaseClass's y()" << endl; } // Not virtual
};

class DerivedClass : public BaseClass
{
public:
    void y() { cout << "DerivedClass's y()" << endl; }
};

int main()
{
    DerivedClass D;
    D.x(); // Calls x() from BaseClass, which in turn calls y()
    return 0;
}
```

Topics

- Inheritance
 - Class Inheritance
 - Protected Members and Class Access
 - Constructors and Destructors in Base and Derived Classes
 - Redefining Base Class Functions
- Polymorphism
 - Virtual Member Functions
- Abstract Base Class
 - Pure Virtual Functions
 - Multiple Inheritance

Polymorphism

- Definition
 - Greek: meaning "many shapes" or "many forms."
 - allows objects of different classes to be treated as objects of a common superclass.
- Types of Polymorphism
 - Compile-Time Polymorphism
 - Achieved through function/operator **overloading** and **redefining**.
 - Run-Time Polymorphism
 - Achieved through **virtual** functions (**overriding**), allowing function behavior to be determined at runtime.

Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class

- Key to run-time polymorphism.
- Declared with the **virtual** keyword in a base class.
- Can be overridden in derived classes for specific behavior.

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

class BaseClass
{
public:
    void x() { y(); }
    virtual void y()
    { cout << "BaseClass's y()" << endl; }

class DerivedClass : public BaseClass
{
public:
    void y() override
    { cout << "DerivedClass's y()" << endl; }

int main()
{
    DerivedClass D;
    D.x(); // DerivedClass's y()
    return 0;
}
```

Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class
 - Key to run-time polymorphism.
 - Declared with the **virtual** keyword in a base class.
 - Can be overridden in derived classes for specific behavior.
- Supports dynamic binding: functions bound at run time to function that they call

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

class BaseClass
{
public:
    void x() { y(); }
    virtual void y()
    { cout << "BaseClass's y()" << endl; }

class DerivedClass : public BaseClass
{
public:
    void y() override
    { cout << "DerivedClass's y()" << endl; }

int main()
{
    DerivedClass D;
    D.x(); // DerivedClass's y()
    return 0;
}
```

Polymorphism and Virtual Member Functions

- **Virtual member function:** function in base class that expects to be redefined in derived class
 - Key to run-time polymorphism.
 - Declared with the **virtual** keyword in a base class.
 - Can be overridden in derived classes for specific behavior.
- Supports **dynamic binding:** functions bound at run time to function that they call
- Without virtual member functions, C++ uses **static binding** (compile time)

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

class BaseClass
{
public:
    void x() { y(); }
    virtual void y()
    { cout << "BaseClass's y()" << endl; }

class DerivedClass : public BaseClass
{
public:
    void y() override
    { cout << "DerivedClass's y()" << endl; }

int main()
{
    DerivedClass D;
    D.x(); // DerivedClass's y()
    return 0;
}
```

Reference Variable to Base Class

- Can define a reference variable to a *base* class object

```
#include <iostream>
using namespace std;
class BaseClass
{
public:
    void x() { y(); }
    virtual void y()
    { cout << "BaseClass's y()" << endl; }
};

class DerivedClass : public BaseClass
{
public:
    void y() override
    { cout << "DerivedClass's y()" << endl; }
};
```

```
int main()
{
    BaseClass b;
    DerivedClass d;
    BaseClass& b1 = d;
    b1.y();

    return 0;
}
```

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.
- So, a virtual function is overridden, and a non-virtual function is redefined.

Concept	Compile-Time Polymorphism	Run-Time Polymorphism
Redefining (no virtual)	<input checked="" type="checkbox"/> Yes, compile-time binding	<input type="checkbox"/> No runtime dispatch
Overriding (with virtual)	<input type="checkbox"/> No, not compile-time	<input checked="" type="checkbox"/> Yes, resolved at runtime

override and final Key Words

- The `override` key word tells the compiler that the function is supposed to override a function in the base class.
- When used with a class, the `final` prevents the class from being used as a base class.
- When used with a member function, the `final` ensures that no derived class can override this particular function.

Topics

- Inheritance
 - Class Inheritance
 - Protected Members and Class Access
 - Constructors and Destructors in Base and Derived Classes
 - Redefining Base Class Functions
- Polymorphism
 - Virtual Member Functions
- Abstract Base Class
 - Pure Virtual Functions
 - Multiple Inheritance

Abstract Base Classes (ABCs)

- Definition
 - Abstract Base Classes (ABCs) serve as foundational blueprints for other classes.
 - **Cannot be instantiated directly due to the presence of at least one pure virtual function.**
- Purpose
 - Define interfaces for a related group of classes.
 - Encourage consistent implementation of shared functionality in derived classes.
 - Facilitate polymorphism, allowing objects of different derived classes to be treated uniformly.
- Characteristics
 - Contains **at least one pure virtual functions.**

Pure Virtual Functions

- Pure virtual function: a **virtual** member function that **must** be overridden in a derived class that has objects
- Syntax

```
virtual void func_name() = 0;  
– The = 0 indicates a pure virtual function.
```

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

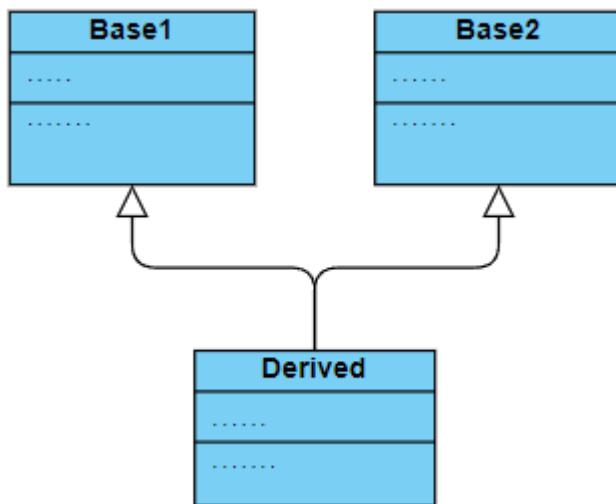
class BaseClass
{
public:
    virtual void y() = 0;
};

class DerivedClass : public BaseClass
{
public:
    void y() override { cout << "DerivedClass's y()" << endl; }
};

int main()
{
    // BaseClass b;
    DerivedClass d;
    d.y();
    return 0;
}
```

Multiple Inheritance

- A derived class can have more than one base class
- Each base class can have its own access specification in derived class's definition:



```
// Derived class inherits from both Base1 and Base2
class Derived : public Base1, public Base2
{
public:
    void displayDerived()
    {
        std::cout << "Derived display" << std::endl;
    }
};
```

Multiple Inheritance

- Arguments can be passed to both base classes' constructors:

```
cube::cube(int side) : square(side), rectSolid(side, side, side);
```

- Base class constructors are called in order given in class declaration, not in order used in class constructor

```

#include <iostream>

// Base class 1
class Base1
{
public:
    void display1()
    {
        std::cout << "Base1 display" << std::endl;
    }
};

// Base class 2
class Base2
{
public:
    void display2()
    {
        std::cout << "Base2 display" << std::endl;
    }
};

// Derived class inherits from both Base1 and Base2
class Derived : public Base1, public Base2
{
public:
    void displayDerived()
    {
        std::cout << "Derived display" << std::endl;
    }
};

```

```

int main()
{
    Derived obj;
    obj.display1();
    obj.display2();
    obj.displayDerived();
    return 0;
}

```

Multiple Inheritance

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
 - Derived class redefines the multiply-defined function
 - Derived class invokes member function in a particular base class using scope resolution operator `::`
- Compiler errors occur if derived class uses base class function without one of these solutions

```
#include <iostream>

// Base class 1
class Base1 {
public:
    void display() {
        std::cout << "Base1 display" << std::endl;
    }
};

// Base class 2
class Base2 {
public:
    void display() {
        std::cout << "Base2 display" << std::endl;
    }
};

// Derived class inherits from both Base1 and Base2
class Derived : public Base1, public Base2 {
public:
    // You can use the scope resolution operator to specify which display() function to call
    void displayDerived() {
        Base1::display(); // Call display() from Base1
        Base2::display(); // Call display() from Base2
    }
};

int main() {
    Derived obj;
    obj.displayDerived(); // Call displayDerived() function of Derived class
    return 0;
}
```

2800ICT

Object Oriented Programming

Exceptions

Review Week6

- Inheritance
 - Class Inheritance
 - Protected Members and Class Access
 - Constructors and Destructors in Base and Derived Classes
 - Redefining Base Class Functions
- Polymorphism
 - Virtual Member Functions
- Abstract Base Class
 - Pure Virtual Functions
 - Multiple Inheritance

Topics

- Exception Handling Basics
- Handling Multiple Exceptions
- Stack Unwinding
- Using Standard Exceptions
- Custom Exception Class

Introduction

- Sometimes the best outcome can be when nothing unusual happens
 - however, exceptional things might happen
 - dividing a number by zero
 - out of range
 - memory error
- C++ exception handling facilities are used when the invocation of a method may cause something exceptional to occur

Exceptions - Terminology

- Exception
 - object or value that signals an error
- Throw an exception
 - send a signal that an error has occurred
- Catch/Handle an exception
 - process the exception; interpret the signal

Exceptions – Flow of Control

- throw
 - followed by an argument, is used to throw an exception
- try
 - followed by a block {}, is used to invoke code that throws an exception
- catch
 - followed by a block {}, is used to detect and process exceptions thrown in preceding try block.
Takes a parameter that matches the type thrown.

```
try
{
    statements1;
    function1();
    statements2;
}
catch (const char* m)
{
    handler-statements;
}
statements3;
```

```
function1()
{
    statements4;
    if (cond)
        throw "Error";
    statements5;
}
```

```
// basic_exception_handling.cpp
#include <iostream>
#include <stdexcept> // for standard exceptions
using namespace std;

// A function that may throw an exception
double divide(int a, int b) {
    if (b == 0) {
        throw runtime_error("Division by zero!");
    }
    return static_cast<double>(a) / b;
}

int main() {
    int x, y;
    cout << "Enter two numbers: ";
    cin >> x >> y;

    try {
        double result = divide(x, y); // Call the function
        cout << "Result: " << result << endl;
    }
    catch (const runtime_error& e) {
        cout << "Exception caught: " << e.what() << endl;
    }

    cout << "Program continues..." << endl;
    return 0;
}
```

Handling Multiple Exceptions

- Context
 - A single try block can throw different types of exceptions.
- Importance: Handling each exception type specifically can lead to more resilient and understandable code.
- Syntax Example:

```
try {  
    // Code that may throw multiple types of exceptions  
} catch (const SpecificExceptionType1& e) {  
    // Handle SpecificExceptionType1  
} catch (const SpecificExceptionType2& e) {  
    // Handle SpecificExceptionType2  
}
```

```

// handling_multipleException.cpp
#include <iostream>
#include <stdexcept> // for standard exceptions
using namespace std;

void testFunction(int a)
{
    if (a == 0)
    {
        throw runtime_error("Zero value error");
    }
    else if (a == 1)
    {
        throw invalid_argument("Invalid value 1");
    }
    else if (a == 2)
    {
        throw logic_error("Logic error occurred");
    }
}

```

```

int main()
{
    try
    {
        testFunction(2);
    }
    catch (const runtime_error &e)
    {
        cout << "Runtime error: "<<e.what()<<endl;
    }
    catch (const invalid_argument &e)
    {
        cout << "Invalid argument: "<<e.what()<<endl;
    }
    catch (const exception &e)
    {
        cout<<"Standard exception: "<<e.what()<<endl;
    }
    catch (...)
    {
        cout << "Unknown exception occurred."<<endl;
    }
}

return 0;
}

```

```
// handling_multipleException2.cpp
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std; // <- Added for simplicity
void processInput(const string &input){
    int number = 0;
    try {
        number = stoi(input);
        if (number < 0 || number > 100){
            throw out_of_range("Number is out of acceptable range (0-100)");
        }
        cout << "Processed number: " << number << endl;
    }
    catch (const invalid_argument &e){
        cerr << "Exception: Invalid argument. Please enter a valid number." << endl;
    }
    catch (const out_of_range &e){
        cerr << "Exception: " << e.what() << endl;
    }
    catch (const exception &e){
        cerr << "Unexpected error: " << e.what() << endl;
    }
}
int main(){
    string userInput = "105"; // Can change to "abc", "50", etc. to test different cases
    processInput(userInput);
    return 0;
}
```

Stack Unwinding

- What is Stack Unwinding?
 - Stack unwinding is C++'s cleanup process *when* something goes wrong (an exception is thrown), and the program **looks for someone to handle that exception.**

Example

```
// stack_unwinding.cpp
#include <iostream>
#include <stdexcept> // for std::runtime_error

void functionC() {
    std::cout << "Inside functionC" << std::endl;
    throw std::runtime_error("exception from functionC");
}

void functionB() {
    std::cout << "Inside functionB" << std::endl;
    functionC();
}

void functionA() {
    std::cout << "Inside functionA" << std::endl;
    try {
        functionB();
    } catch (const std::exception& e) {
        std::cout << "Caught exception: " << e.what() << std::endl;
    }
}

int main() {
    functionA();
    return 0;
}
```

Using Standard Exceptions

- The C++ Standard Library provides a hierarchy of exception classes designed to represent a broad range of error conditions
 - Root
 - `std::exception`
 - Derived
 - `std::runtime_error`, `std::logic_error`, `std::out_of_range`, `std::invalid_argument`
- Why Use Standard Exceptions?
 - Consistency: more understandable to other developers familiar with C++.
 - Specificity: represent specific error conditions accurately.
 - **Compatibility:** Ensures your code is compatible with other C++ libraries that might catch these exceptions.

Example

```
#include <iostream>
#include <stdexcept> // Include for standard exceptions

void processInput(int input) {
    if (input < 0) {
        throw std::invalid_argument("Input must be positive");
    } else if (input == 0) {
        throw std::logic_error("Zero is not a valid input");
    }

    // Process the positive input
    std::cout << "Processing input: " << input << std::endl;
}

int main() {
    try {
        processInput(-1); // Example input
    } catch (const std::invalid_argument& e) {
        std::cerr << "Invalid argument: " << e.what() << std::endl;
    } catch (const std::logic_error& e) {
        std::cerr << "Logic error: " << e.what() << std::endl;
    } catch (...) {
        std::cerr << "An unexpected error occurred." << std::endl;
    }
    return 0;
}
```

Custom Exception Class

- Why define custom exceptions?
 - Enhance clarity by signaling specific error types
 - Improve maintainability and error handling structure
 - Allow attachment of contextual information

Custom Exception Class

- Implementing a Custom Exception Class
 - Deriving from **std::exception**
 - **#include <stdexcept>**
 - A **message** variable
 - Details/info about the exception
 - **Constructor**
 - To initialize the exception with error-specific information.
 - **what()** Method
 - Overridden to return a message describing the error.
 - **const char* what() const noexcept override;**
 - what() returns a C-style string: **const char***
 - Why C-style?
 - noexcept ensures no exceptions are thrown from what()

Example

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

class MyCustomException : public std::exception {
private:
    std::string message;
public:
    MyCustomException(const std::string& msg) : message(msg) {}

    virtual const char* what() const override noexcept{
        return message.c_str();
    }
};

void someFunctionThatMightFail() {
    // Some condition that leads to an error
    throw MyCustomException("Something went wrong in the application");
}

int main() {
    try {
        someFunctionThatMightFail();
    } catch (const MyCustomException& e) {
        std::cerr << "Caught an error: " << e.what() << std::endl;
    }
    return 0;
}
```

Example: used in a class

```
#include <iostream>
#include <stdexcept>
#include <string>
// Custom exception class
class InsufficientFundsException : public std::exception {
private:
    std::string message;
public:
    InsufficientFundsException(const std::string& msg) : message(msg) {}
    // Override what() to return the custom message
    virtual const char* what() const noexcept override {
        return message.c_str();
    }
};

class BankAccount {
private:
    double balance;
public:
    BankAccount(double initialBalance) : balance(initialBalance) {}

    void withdraw(double amount) {
        if (amount > balance) {
            throw InsufficientFundsException("Withdrawal amount exceeds balance.");
        }
        balance -= amount;
        std::cout << "Withdrawal successful. New balance: $" << balance << std::endl;
    }
};
```

Example

```
int main() {
    BankAccount account(100.00); // Initialize account with $100

    try {
        std::cout << "Attempting to withdraw $150..." << std::endl;
        account.withdraw(150.00); // Attempt to withdraw $150
    } catch (const InsufficientFundsException& e) {
        std::cerr << "An error occurred: " << e.what() << std::endl;
    }

    return 0;
}
```

cin vs getline

- cin
 - cin.fail(): Checks if input failed
 - cin.clear(): Resets the stream to a usable state
 - cin.ignore(n, char): Flushes n invalid characters
- getline

```
std::getline(std::istream& is, std::string& str);
```

Feature	cin >>	getline(std::cin, ...)
Reads spaces?	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> Yes
Stops at?	space, tab, or newline	Newline \n
Leaves \n in buffer?	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No (it consumes \n)
Common use case	Numbers, single word	Full lines, sentences

```
// string2.cpp
#include <iostream>
#include <string>
#include <limits>
#include <sstream>
using namespace std;
int main()
{
    int value;
    string message;
    cout << "Enter text: ";

    cin >> message;
    // cin.clear();
    // cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cout << "The text entered is: " << message << endl;

    cout << "Enter a number: ";
    cin >> value;
    // cin.clear();
    // cin.ignore(numeric_limits<streamsize>::max(), '\n');
    cout << "The number entered is: " << value << endl;

    cout << "Enter text: ";
    getline(cin, message);

    cout << "The text entered is:" << message << endl;
    cout << int(message.length()) << endl;
    return 0;
}
```

Rethink main function

```
#include <iostream>
#include <cstdlib> // for std::atoi
using namespace std;

int main(int argc, char* argv[]) {
    if (argc < 3) {
        cout << "Usage: " << argv[0] << " <a> <b>" << endl;
        return 1;
    }

    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int sum = a + b;

    cout << "Sum: " << sum << endl;
    return 0;
}

string input = argv[1];
double d = stod(input); // convert to double
float f = stof(input); // convert to float
```

2800ICT

Object Oriented Programming

Functors and Lambda Expressions

Topics

- Function Object/Functor
 - operator ()
 - Anonymous Function Objects
- Lambda Expressions

Function Objects - functor

- Definition
 - Functors (or function objects) are objects that can be called as if they are a function.

```
Counter c;  
cout << c() << endl;
```

- Key Point
 - Any class with the **operator()** overloaded becomes a functor.

Function Objects

- A function object acts like a function.
 - It can be called
 - It can accept arguments
 - It can return a value

Function Objects

- To create a function object, you write a class that overloads the operator () .

```
class Counter
{
private:
    int count;

public:
    Counter() : count(0) {}

    int operator()(int step = 1)
    {   count += step;
        return count;
    }
};

#include <iostream>
int main()
{
    Counter c;

    std::cout << "Called: " << c() << " times.\n";
    std::cout << "Called: " << c() << " times.\n";
    std::cout << "Called: " << c(5) << " times.\n";

    return 0;
}
```

Anonymous Function Objects

- Function objects can be called at the point of their creation, without being given a name. Consider this class:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class IsEven
{
public:
    void operator()(int &x) const
    {
        x = x % 2;
    }
};
int main()
{
    vector<int> vint{1, 2, 3, 4, 5};
    for_each(vint.begin(), vint.end(), IsEven());
    return 0;
}
```

Lambda Expressions

- Syntax:

[capture](parameter list) ->returnType { *body* }

- [] is known as the lambda introducer. It marks the beginning of a lambda expression.
- *parameter List* is a list of parameter
- ->returnType is optional
 - The compiler can determine it based on the code.
- *function body* is the code.

Lambda Expressions

- Lambda expressions generate **function objects**,
- You can **assign** a lambda expression to a variable and
- then **call** it through the variable's name:

```
auto sum = [](int a, int b) {return a + b;};
int x = 2;
int y = 5;
int z = sum(x, y);
```

Anonymous Function Objects

- Example: a lambda expression for a function object that determines whether an integer is even is:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
    int evenNums = count_if(v.cbegin(), v.cend(), [] (int x)
                           { return x % 2 == 0; });
    cout << "The vector contains " << evenNums << " even numbers.\n";
    return 0;
}
```

Lambda Expressions

```
[](int a, int b) { return a + b; }  
[](int a) { cout << a * a << " ";
```

Lambda Expressions

- When you call a lambda expression, you write a list of arguments, enclosed in parentheses, right after the expression.
- For example, the following code snippet displays 7, which is the sum of the variables `x` and `y`:

```
int x = 2;  
int y = 5;  
cout << [](int a, int b) {return a + b;}(x, y) << endl;
```

Lambda: Capture groups

- `[capture](parameter List) { function body }`
- Variables can be **captured** individually, by listing them. For example, the capture group `[x, &y, z]` captures x and z by value, and y by reference
- A **default capture mode** can also be specified, which means that you do not have to manually list every referenced variable, unless a referenced variable needs the opposite capture mode. For example:
 - `[=]` captures all variables used in the closure **by value**
 - `[&]` captures all variables used in the closure **by reference**
 - `[=, &y]` captures all variables by value, except y, which is captured by reference
 - `[&, y]` captures all values by reference, except y, which is captured by value

Lambda: Capturing by value

```
#include <iostream>

int main()
{
    int x = 5;
    int y = 3;
    auto add = [x, y]()
    {
        return x + y;
    };
    std::cout << "Result of addition: " << add() << std::endl;

    auto subtract = [&x, &y]()
    {
        return x - y;
    };

    std::cout << "Result of subtraction: " << subtract() << std::endl;

    x = 10;
    y = 2;
    std::cout << "Updated result of addition: " << add() << std::endl;
    std::cout << "Updated result of subtraction: " << subtract() << std::endl;

    return 0;
}
```

2800ICT

Object Oriented Programming

Smart Pointers

Review Week 7

- Exception Handling Basics
- Handling Multiple Exceptions
- Stack Unwinding
- Using Standard Exceptions
- Custom Exception Class
- Function Object/Functor
 - operator ()
 - Anonymous Function Objects
- Lambda Expressions

Topics

- **unique_ptr**
- **shared_ptr**
- **weak_ptr**

Why Smart Pointers?

- The Problem with Raw Pointers
 - No deletion → memory leaks
 - Early deletion → dangling pointers
 - Double-freeing
- Smart pointers help you
 - avoid manual new/delete,
 - reduce memory bugs, and
 - make your C++ code cleaner, safer, and easier to maintain.

Smart pointer classes to the rescue

- `#include <memory>`
- Smart Pointer Types
 - `unique_ptr`
 - `shared_ptr`
 - `weak_ptr`

Best Practice:

It is recommended to use smart pointers!

Smart Pointers

- Behave like built-in (raw) pointers
- Also manage dynamically created objects
 - Objects get deleted in smart pointer destructor
- Type of ownership:
 - unique
 - shared

std::unique_ptr

- It provides **exclusive** ownership
 - only **one** unique_ptr can own a given object at a time.
- It will automatically free the resource when going out of *scope*.
 - scope refers to the code block where a variable is alive.
- Syntax:

```
std::unique_ptr<Type> ptr = std::make_unique<Type>(constructor_args);
```

- Example

```
std::unique_ptr<int> ptr = std::make_unique<int>(10);
```

Std::make_unique

- C++14 comes with an additional function named std::make_unique().
- This templated function
 - **constructs** an object of the template type and
 - **initializes** it with the arguments passed into the function

```
#include <iostream>
#include <memory>
class Resource {
private:
    int flag = 0;
public:
    Resource(int flag = 1) : flag(flag) {
        std::cout << flag << " | Resource acquired\n";
    }
    ~Resource() {
        std::cout << flag << " | Resource destroyed\n";
    }
    friend std::ostream& operator<<(std::ostream& out, const Resource& res);
};
std::ostream& operator<<(std::ostream& out, const Resource& res) {
    out << "Resource(flag=" << res.flag << ")";
    return out;
}
void someFunction2() {
    auto ptr = std::make_unique<Resource>(2);

    std::cout << *ptr << std::endl;
    std::cout << "Address of ptr: " << &ptr << std::endl;
}
int main() {
    someFunction2();
    std::cout << "End of main\n";
    return 0;
}
```

Std::unique_ptr

- Cannot
 - Cannot be copied to another std::unique_ptr
 - Cannot be passed by value to a function
- Can
 - A unique_ptr can only be moved

```
// unique_ptr_1.cpp
#include <iostream>
#include <memory> // for unique_ptr

void demoUniquePtr()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(10);
    std::cout << "Value: " << *ptr << std::endl;

    std::unique_ptr<int> ptr1 = std::make_unique<int>(42);
    // std::unique_ptr<int> ptr2 = ptr1;
    std::unique_ptr<int> ptr2 = std::move(ptr1);
    if (ptr2)
        std::cout << "Transferred value: " << *ptr2 << std::endl;
}

int main()
{
    demoUniquePtr();
    return 0;
}
```

`std::shared_ptr`

- allows multiple `shared_ptr` instances to **share ownership of the same object**.
- `std::make_shared`

`std::shared_ptr`

- Each time a `shared_ptr` is assigned
 - a **use_count** is incremented (there is one more “owner” of the data)
- When a `shared_ptr` goes out of scope
 - the `reference_count` is decremented
 - if `reference_count = 0`, the object referenced by the pointer is freed.

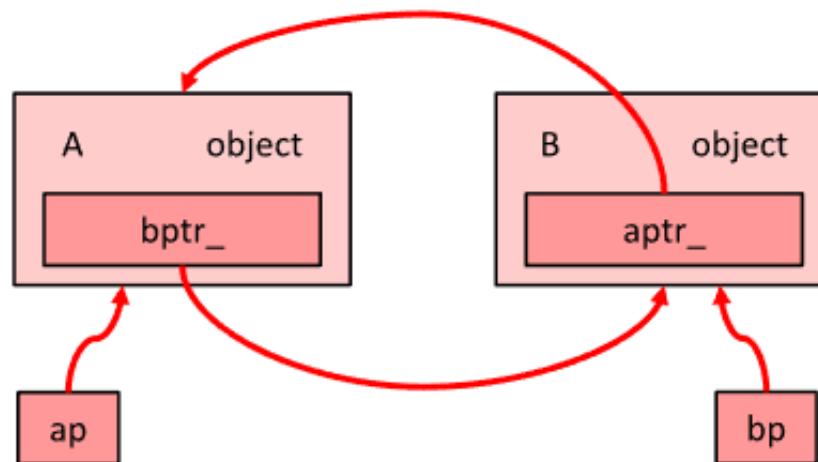
```
// shared_ptr.cpp
#include <iostream>
#include <memory>
using namespace std;
class MyClass {
public:
    MyClass() { cout << "Constructor\n"; }
    ~MyClass() { cout << "Destructor\n"; }
};
int main() {
    shared_ptr<MyClass> p1 = make_shared<MyClass>();
    cout << "p1 use count: " << p1.use_count() << endl;
    shared_ptr<MyClass> p2 = p1;
    cout << "p2 use count: " << p2.use_count() << endl;
    {
        shared_ptr<MyClass> p3 = p2;
        cout << "p3 use count: " << p3.use_count() << endl;
    }
    cout << "After p3 scope, p1 use count: " << p1.use_count() << endl;
    p2.reset();
    cout << "After p2.reset(), p1 use count: " << p1.use_count() << endl;
    p1.reset();
    cout << "After p1.reset(), p1 use count: " << p1.use_count() << endl;
    cout << "End of main\n";
    return 0;
}
```

`std::shared_ptr`

- Circular dependency issues with `std::shared_ptr`

class **A** and **B**, each holding a `std::shared_ptr` to the other:

- **ap** is a shared pointer to an **A** object
- **bp** is a shared pointer to a **B** object
- Inside the **A** object, there's a `shared_ptr` **bptr_**
- Inside the **B** object, there's a `shared_ptr<A>` **aptr_**



```
// share_ptr1.cpp
#include <iostream>
#include <memory> // for shared_ptr
#include <string>
using namespace std;
class Person
{
private:
    string m_name;
    shared_ptr<Person> m_partner; // initially created empty
public:
    Person(const string &name) : m_name(name) {
        cout << m_name << " created\n";
    }
    ~Person() {
        cout << m_name << " destroyed\n";
    }
    friend bool partnerUp(shared_ptr<Person> &p1, shared_ptr<Person> &p2) {
        if (!p1 || !p2)
        {
            return false;
        }
        p1->m_partner = p2;
        p2->m_partner = p1;

        cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';
        return true;
    }
};
```

```
void testSharedPtrCouple()
{
    auto lucy = make_shared<Person>("Lucy");
    auto ricky = make_shared<Person>("Ricky");

    partnerUp(lucy, ricky); // Mutual ownership

    cout << "Lucy use_count: " << lucy.use_count() << endl; // Likely 2
    cout << "Ricky use_count: " << ricky.use_count() << endl; // Likely 2
}

int main()
{
    testSharedPtrCouple();
    cout << "End of main\n";
    return 0;
}
```

`std::weak_ptr`

- `std::weak_ptr` was designed to solve the “cyclical ownership” problem described above.
- `std::weak_ptr` is **not** considered **an owner**.
 - when a `std::shared` pointer goes out of scope, the runtime only considers **whether other `std::shared_ptr` are co-owning the object**.
 - Therefore, `std::weak_ptr` **does not count!**

```
// weak_ptr1.cpp
#include <iostream>
#include <memory> // for shared_ptr
#include <string>
using namespace std;
class Person
{
private:
    string m_name;
    weak_ptr<Person> m_partner; // note: This is now a weak_ptr
public:
    Person(const string &name) : m_name(name) {
        cout << m_name << " created\n";
    }
    ~Person() {
        cout << m_name << " destroyed\n";
    }
    friend bool partnerUp(shared_ptr<Person> &p1, shared_ptr<Person> &p2) {
        if (!p1 || !p2) {
            return false;
        }
        p1->m_partner = p2;
        p2->m_partner = p1;

        cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';
        return true;
    }
};
```

```
void testSharedPtrCouple()
{
    auto lucy = make_shared<Person>("Lucy");
    auto ricky = make_shared<Person>("Ricky");

    partnerUp(lucy, ricky); // Mutual ownership

    cout << "Lucy use_count: " << lucy.use_count() << endl;
    cout << "Ricky use_count: " << ricky.use_count() << endl;
}

int main()
{
    testSharedPtrCouple();
    cout << "End of main\n";
    return 0;
}
```

std::weak_ptr

- Downside
 - std::weak_ptr is not directly usable
 - it has **no operator->**
- member function: lock()
 - use its **lock()** member function to **convert it into a std::shared_ptr**

```
// weak_to_share.cpp
#include <iostream>
#include <memory> // for shared_ptr and weak_ptr
#include <string>
using namespace std;
class Person
{
private:
    string m_name;
    weak_ptr<Person> m_partner; // note: This is now a weak_ptr
public:
    Person(const string &name) : m_name(name) {
        cout << m_name << " created\n";
    }
    ~Person() {
        cout << m_name << " destroyed\n";
    }
    friend bool partnerUp(shared_ptr<Person> &p1, shared_ptr<Person> &p2) {
        if (!p1 || !p2)
            return false;
        p1->m_partner = p2;
        p2->m_partner = p1;
        cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';
        return true;
    }
    // use lock() to convert weak_ptr to shared_ptr
    const shared_ptr<Person> getPartner() const { return m_partner.lock(); }
    const string &getName() const { return m_name; }
};
int main()
{
    auto lucy{make_shared<Person>("Lucy")};
    auto ricky{make_shared<Person>("Ricky")};
    partnerUp(lucy, ricky);
    auto partner = ricky->getPartner(); // get shared_ptr to Ricky's partner
    cout << ricky->getName() << "'s partner is: " << partner->getName() << '\n';
    return 0;
}
```

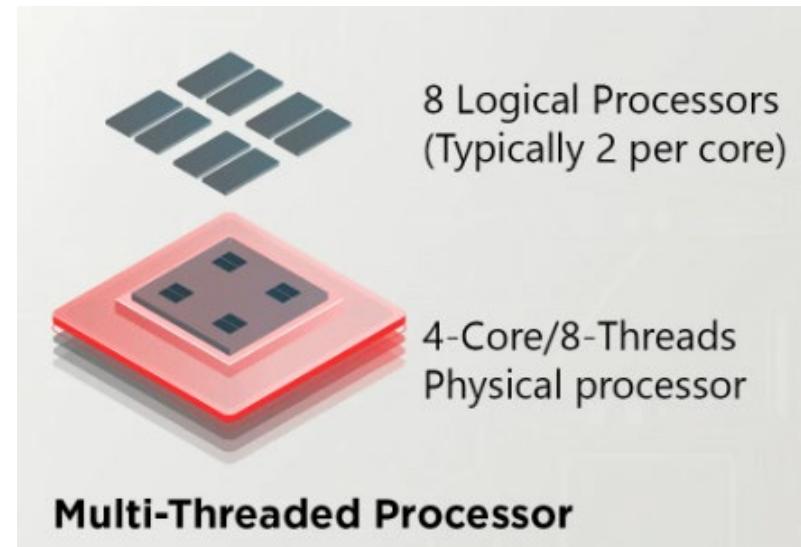
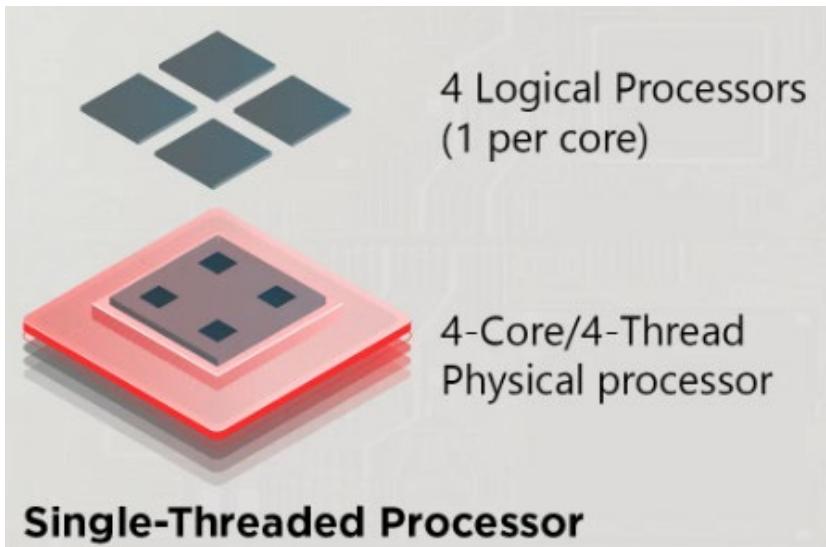
2800ICT

Object Oriented Programming

Multi-Threading in C++

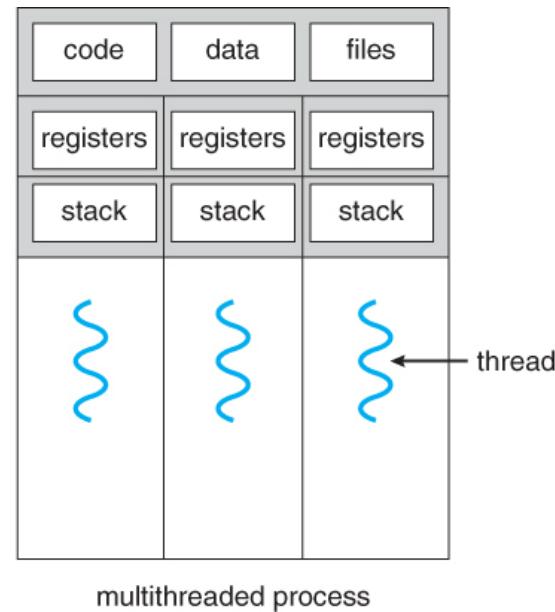
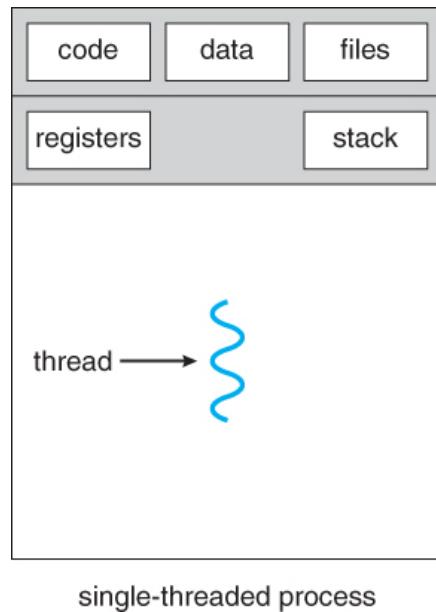
Introduction to Multi-Threading

- **Definition:** What is multi-threading?
 - Multi-Threading



Introduction to Multi-Threading

- **Definition:** What is multi-threading?
 - Multi-Threading
 - Thread vs. Process



Introduction to Multi-Threading

- **Definition:** What is multi-threading?
 - Multi-Threading
 - Thread vs. Process
- **Importance:** Why use multi-threading?
 - Performance Improvement
 - Responsiveness
 - Efficient Resource Utilization

C++ Thread Library

- Header File:
 - `#include <thread>`
- Main Components:
 - `std::thread`
 - Managing thread lifecycle
 - **creation**
 - **join()**
 - **join() is a member function of std::thread**
 - telling the main thread:
"Pause here until that thread is completely done."

Creating Threads in C++

- Syntax:
 - `std::thread threadObj(<function>, <args>...);`
 - The function **f** will be invoked in a **new thread** with the arguments **args**
 - The thread will terminate once **f** returns

```
// thread0.cpp
#include <iostream>
#include <thread>
#include <string>

void printMessage(const std::string &message) {
    std::cout << "Thread message: " << message << std::endl;
}

int main() {
    std::string hello = "Hello, Multi-threading World!";
    std::thread t(printMessage, hello);
    t.join();
    return 0;
}
```

std::ref/std::cref

- creates a (constant) reference wrapper for a variable
 - allowing it to be passed by reference
 - It essentially says:
"Here, take this reference, not a copy."

```
// thread_ref.cpp
#include<iostream>
#include<string>
#include<thread>
/*
For multithreading, with std::thread,
it's necessary to explicitly bind references using std::ref for parameter
passing;
otherwise, the reference declaration in the parameter is ineffective.
*/
void threadFunc(std::string &str, int &a)
{
    str = "change by threadFunc";
    a = 13;
}
int main()
{
    std::string str("main");
    int a = 9;
    threadFunc(str, a);
    std::cout<<"str = " << str << std::endl;
    std::cout<<"a = " << a << std::endl;
    std::thread th(threadFunc, std::ref(str), std::ref(a));
    th.join();
    std::cout<<"str = " << str << std::endl;
    std::cout<<"a = " << a << std::endl;
    return 0;
}
```

```

// thread_ref.cpp
#include<iostream>
#include<string>
#include<thread>
/*
For multithreading, with std::thread,
it's necessary to explicitly bind references using std::ref for parameter
passing;
otherwise, the reference declaration in the parameter is ineffective.
*/
void threadFunc(std::string &str, int &a)
{
    str = "change by threadFunc";
    a = 13;
}
int main()
{
    std::string str("main");
    int a = 9;
    threadFunc(str, a);
    std::cout<<"str = " << str << std::endl;
    std::cout<<"a = " << a << std::endl;
    std::thread th(threadFunc, std::ref(str), std::ref(a));
    th.join();
    std::cout<<"str = " << str << std::endl;
    std::cout<<"a = " << a << std::endl;
    return 0;
}

```

Q: If **a** is already a reference, do we still need **std::ref(a)** when passing it to a thread?

Thread Management

- **join()**
 - used to **wait for a thread to finish**
 - must be called before an std::thread object is destroyed
 - Ensuring threads complete execution.

```
// thread3.cpp
#include <iostream>
#include <thread>
using namespace std;

void foo(string flag, int n)
{
    for(int i=0;i<n; i++)
    {
        cout <<flag;
    }
}

int main()
{
    thread first(foo,"thread 1\n", 5);
    thread second(foo,"thread 2\n", 5);

    // Pauses until first finishes.
    // This operation must complete before it can be destroyed.
    first.join();

    // Pauses until second finishes.
    // The program can only continue after join is complete.
    second.join();
    std::cout << "Main thread\n";
    return 0;
}
```

Other Functions

- useful functions that are closely related to starting and stopping threads:
 - `std::this_thread::sleep_for()`: Stop the current thread for a given amount of time
 - `std::this_thread::sleep_until()`: Stop the current thread until a given point in time
 - `std::this_thread::yield()`: Let the operating system schedule another thread
 - `std::this_thread::get_id()`: Get the id of the current thread

Synchronization - atomic

- Need for Synchronization:
 - Data Races
 - Consequences
- std::atomic
 - `#include <atomic>`
 - a powerful feature
 - ensures that **writes** to a variable are done without interruption, even across multiple threads.

```
// thread_atomic.cpp
#include <iostream>
#include <thread>
#include <atomic>

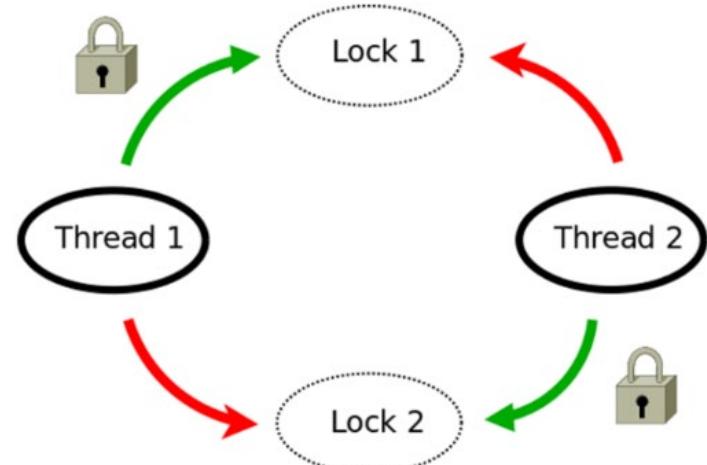
std::atomic<int> counter{0};

void increment() {
    for (int i = 0; i < 10000; ++i)
        ++counter;
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << "Counter: " << counter << std::endl;
}
```

Synchronization-Mutexes

- Mutexes: class std::mutex
 - short for mutual exclusion
 - `#include <mutex>`
 - allows only one thread at a time to access a critical section of code.
- Locking and Unlocking:
 - lock
 - unlock
 - `lock_guard<mutex>`



Synchronization and Mutexes

- Locking and Unlocking:
 - `lock_guard`
 - Locking upon creation
 - **Automatic** unlocking upon scope exit
 - No mid-scope unlocking
 - Non-copyable

```
// thread_mutex.cpp
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx; // create a mutex
int counter = 0;

void safeIncrement() {
    std::lock_guard<std::mutex> lock(mtx); // locks and unlocks automatically
    ++counter;
    std::cout << std::this_thread::get_id() << ":" << counter << '\n';
}

int main() {
    std::thread t1(safeIncrement);
    std::thread t2(safeIncrement);

    t1.join();
    t2.join();

    std::cout << "Counter: " << counter << std::endl;
}
```

push_back vs emplace_back

- two important ways to add elements to a std::vector (and other STL containers)

```
// push_emplace.cpp
int main()
{
    std::vector<Person> people;

    std::cout << "\n--- push_back with copy ---\n";
    Person p("Alice", 30);
    people.push_back(p); // Calls copy constructor

    std::cout << "\n--- push_back with move ---\n";
    people.push_back(std::move(p)); // Calls move constructor

    std::cout << "\n--- emplace_back ---\n";
    people.emplace_back("Bob", 25); // Constructs directly in-place, no copy/move

    return 0;
}
```

```
// push_emplace.cpp
#include <iostream>
#include <vector>
#include <string>

class Person
{
private:
    std::string name;
    int age;
public:
    Person(std::string n, int a) : name(std::move(n)), age(a)
    {
        std::cout << "Constructed: " << name << " (" << age << age << ")\n";
    }
    Person(const Person &other) : name(other.name), age(other.age)
    {
        std::cout << "Copied: " << name << "\n";
    }
    Person(Person &&other) noexcept : name(std::move(other.name)), age(other.age)
    {
        std::cout << "Moved: " << name << "\n";
    }
};
```

push_back vs emplace_back

Feature	push_back (obj)	emplace_back (args) / (obj)
Requires existing object	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No constructs in-place from arguments
Creates temporary	<input checked="" type="checkbox"/> Yes then copy/move into container	<input type="checkbox"/> No directly constructs in container memory
Constructor called	<input type="checkbox"/> Not necessarily	<input checked="" type="checkbox"/> Yes constructor called with forwarded arguments
Performance	<input type="checkbox"/> Slightly slower copy/move may occur	<input checked="" type="checkbox"/> Faster no copy/move when constructing new object
Code clarity	<input checked="" type="checkbox"/> Clear, but longer	<input checked="" type="checkbox"/> Shorter and more expressive for complex constructors

Suggestion: You can often replace `push_back()` with `emplace_back()`.

thread in containers: emplace_back

```
// thread5_1.cpp
#include <iostream>
#include <vector>
#include <thread>
#include <string>
#include <sstream> // <== include this
void task(int num) {
    std::ostringstream oss;
    oss << "thread id: " << std::this_thread::get_id()
        << " | task number: " << num << "\n";
    std::string str = oss.str();

    std::cout << str;
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i) {
        threads.emplace_back(task, i); // constructs thread in-place
    }
    for (auto& t : threads) {
        t.join();
    }
    return 0;
}
```

thread in containers: push_back

```
// thread5.cpp
#include <iostream>
#include <vector>
#include <thread>
#include <string>
#include <sstream> // <== include this
void task(int num) {
    std::ostringstream oss;
    oss << "thread id: " << std::this_thread::get_id()
        << " | task number: " << num << "\n";
    std::string str = oss.str();
    std::cout << str;
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i) {
        std::thread t(task, i);
        threads.push_back(std::move(t)); // must move, thread is non-copyable
    }
    for (auto& t : threads) {
        t.join();
    }
    return 0;
}
```

Example – multiple-threading

- Create a 2D array with 2 rows and 3 columns
- `array<array<int, 3>, 2> grid`
 - an array of arrays
 - `array<..., 2>` - 2 rows
 - `array<int, 3>` - 3 columns
- Access elements
 - `grid.at(row_index).(column_index)` for **safety**
 - `grid[row_index][column_index]` for **speed**
 - range-based for-loop
 - traditional index-based for-loop


```
// 2D-array.cpp
#include <iostream>
#include <array>
using namespace std;
int main()
{
    // Create and initialize a 2D array with 2 rows and 3 columns
    array<array<int, 3>, 2> grid{};
    // Modify a value
    grid[1][2] = 99;

    // Print the 2D array
    for (const auto &row : grid) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << "\n";
    }
    for (size_t i = 0; i < grid.size(); ++i) {
        for (size_t j = 0; j < grid.at(i).size(); ++j) {
            std::cout << grid.at(i).at(j) << " ";
            // std::cout << grid[i][j] << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

Example – multiple-threading

- Create a 2D vector with 2 rows and 3 columns
- `vector<vector<int>> grid(2, vector<int>(3))`
 - a vector of vectors
- Access elements
 - `grid.at(row_index).(column_index)` for **safety**
 - `grid[row_index][column_index]` for **speed**
 - range-based for-loop
 - traditional index-based for-loop


```
// 2D-vector.cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<vector<int>> grid(2, vector<int>(3));
    grid[1][2] = 99;
    for (const auto &row : grid)  {
        for (int val : row)  {
            cout << val << " ";
        }
        cout << "\n";
    }
    for (size_t i = 0; i < grid.size(); ++i)  {
        for (size_t j = 0; j < grid.at(i).size(); ++j)  {
            cout << grid.at(i).at(j) << " ";
            // cout << grid[i][j] << " "; // this works too, just no bounds check
        }
        cout << "\n";
    }
    return 0;
}
```

Example – Multi-threaded Matrix Addition

- multi-threaded program that calculates the element-wise sum of two 2D arrays of size 3x4
- $c[i][j] = a[i][j] + b[i][j]$
- Solutions?
 - By row
 - One thread takes care of one row
 - By col
 - One thread takes care of one column
 - ?

$$\begin{array}{|c|c|c|c|} \hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} + \begin{array}{|c|c|c|c|} \hline 12 & 11 & 10 & 9 \\ \hline 13 & 13 & 13 & 5 \\ \hline 13 & 13 & 13 & 1 \\ \hline \end{array}$$

```

// 2D-multi-threading.cpp
// Multi-threaded Matrix Addition
#include <iostream>
#include <vector>
#include <thread>

using namespace std;

// Thread function: add one row from A and B into C
void addRow(const vector<vector<int>>& A,
            const vector<vector<int>>& B,
            vector<vector<int>>& C,
            int row)
{
    for (size_t col = 0; col < A[0].size(); ++col)
    {
        C[row][col] = A[row][col] + B[row][col];
    }
}

```

1	2	3	4
5	6	7	8
9	10	11	12

+

12	11	10	9
13	13	13	5
13	13	13	1

```

int main()
{
    const int rows = 3, cols = 4;
    // Initialize matrices A and B with example values
    vector<vector<int>> A(rows, vector<int>(cols, 1)); // all elements = 1
    vector<vector<int>> B(rows, vector<int>(cols, 2)); // all elements = 2
    vector<vector<int>> C(rows, vector<int>(cols));      // result matrix
    // Create a thread for each row
    vector<thread> threads;
    for (int i = 0; i < rows; ++i){
        threads.emplace_back(addRow, cref(A), cref(B), ref(C), i);
    }
    // Join all threads
    for (auto &t : threads){
        t.join();
    }
    cout << "Result matrix C = A + B:\n";
    for (const auto &row : C) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << "\n";
    }
    return 0;
}

```

Review Week8

- Smart pointers
 - unique_ptr
 - shared_ptr
 - weak_ptr
- Multi-Threading
 - std::thread threadObj(<function>, <args>...);
 - threadObj.join()
 - push_back vs emplace_back

2800ICT
Object Oriented Programming

C++ Data Structure
- skip list

Topics

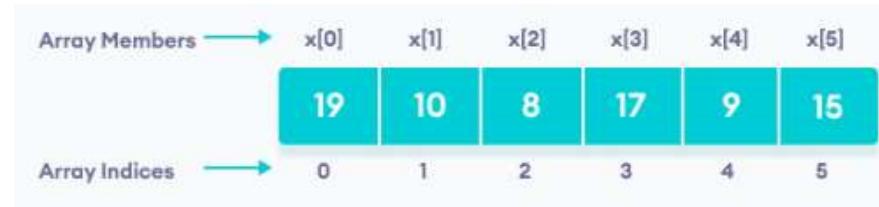
- STL Container
 - List
 - double linked list
- A New Data Structure
 - skip_list

Revisiting: Array, Vector

- **Array**

- Features:

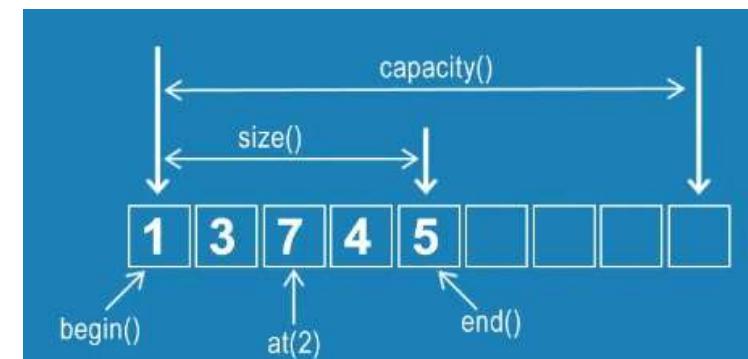
- fixed-size
 - contiguous memory



- **Vector**

- Features:

- contiguous memory
 - dynamic
 - can change in size



Array vs Vector

Capacity

[size](#)

[max_size](#)

[empty](#)

Element access

[operator\[\]](#)

[at](#)

[front](#)

[back](#)

[data](#)

Modifiers

[fill](#)

[swap](#)

Capacity:

[size](#)

[max_size](#)

[resize](#)

[capacity](#)

[empty](#)

[reserve](#)

[shrink_to_fit](#)

Element access:

[operator\[\]](#)

[at](#)

[front](#)

[back](#)

[data](#)

Modifiers:

[assign](#)

[push_back](#)

[pop_back](#)

[insert](#)

[erase](#)

[swap](#)

[clear](#)

[emplace](#)

[emplace_back](#)

Revisiting: Array, Vector

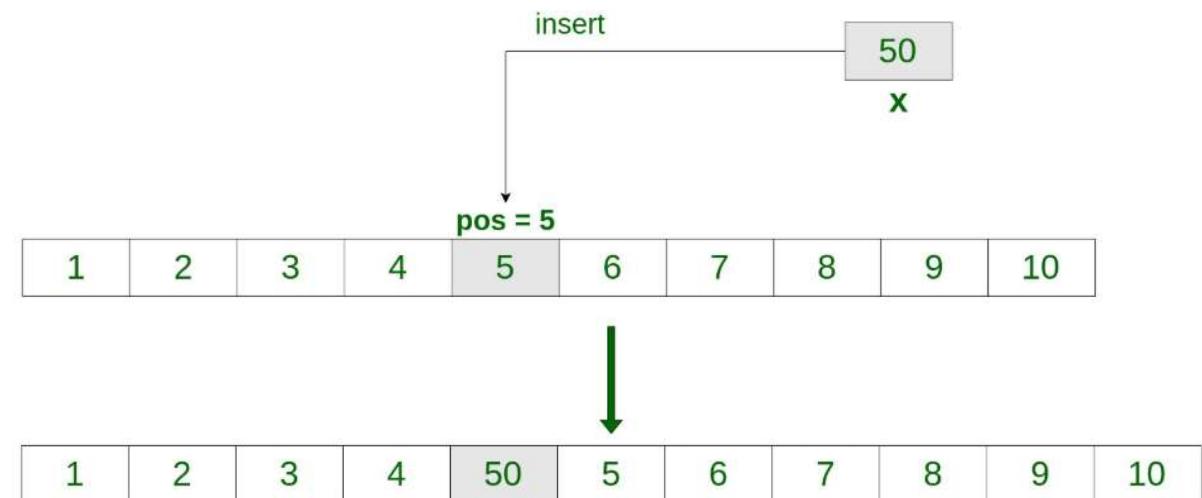
- What about inserting/deleting an element

- Array

- ?

- Vector

- ?



Revisiting: Array, Vector

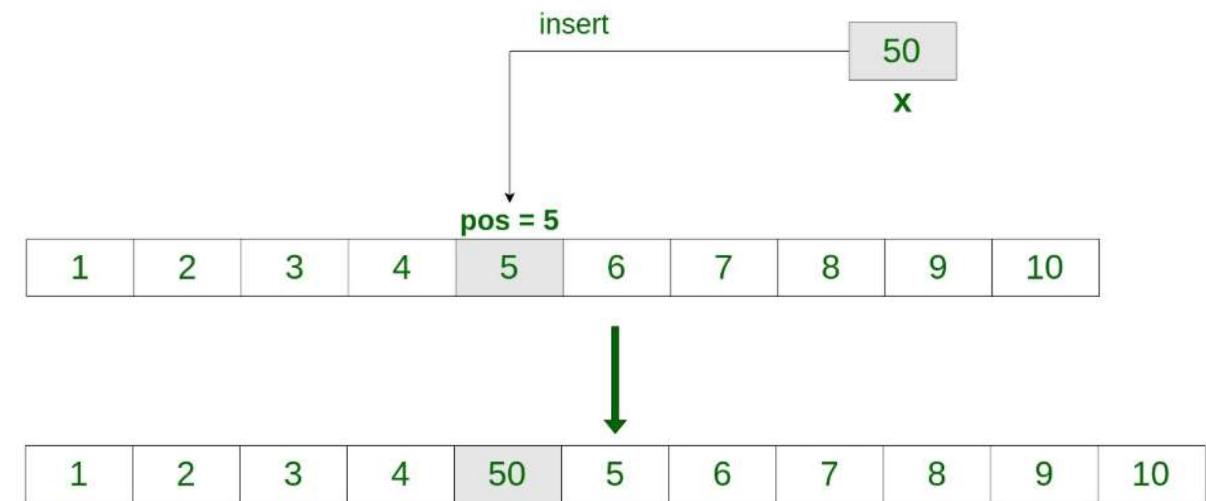
- What about inserting/deleting an element

- Array

- ?

- Vector

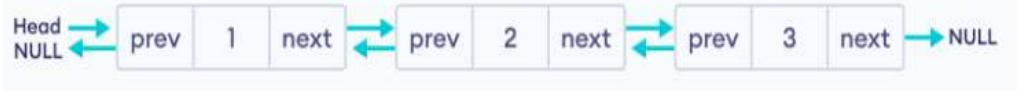
- ?



Vector:

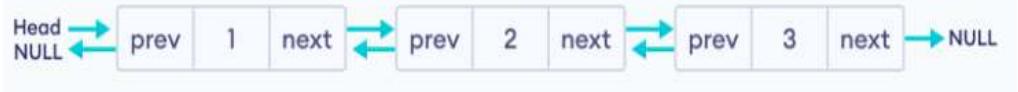
Inserting or deleting elements (except at the end) requires shifting later elements, which is inefficient.

STL: list



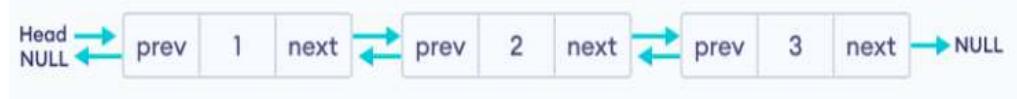
- is implemented as doubly-linked lists
- node: 
 - *prev – pointer/address of the previous node
 - data - data item
 - *next – pointer/address of next node

STL: list



- is implemented as doubly-linked lists
- allows constant time insert and erase operations anywhere within the sequence
- Compared to array and vector,
 - Benefit
 - lists perform generally **better** in inserting and moving elements.
 - Drawback
 - lists lack direct access to the elements by their position.

STL: list



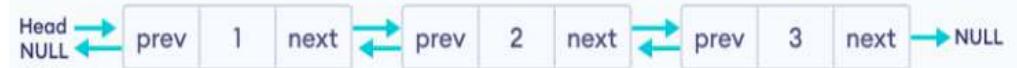
Iterators:

<u>begin</u>	Return iterator to beginning (public member function)
<u>end</u>	Return iterator to end (public member function)
<u>rbegin</u>	Return reverse iterator to reverse beginning (public member function)
<u>rend</u>	Return reverse iterator to reverse end (public member function)
<u>cbegin</u>	Return const_iterator to beginning (public member function)
<u>cend</u>	Return const_iterator to end (public member function)
<u>crbegin</u>	Return const_reverse_iterator to reverse beginning (public member function)
<u>crend</u>	Return const_reverse_iterator to reverse end (public member function)

Capacity:

<u>empty</u>	Test whether container is empty (public member function)
<u>size</u>	Return size (public member function)
<u>max_size</u>	Return maximum size (public member function)

STL: list



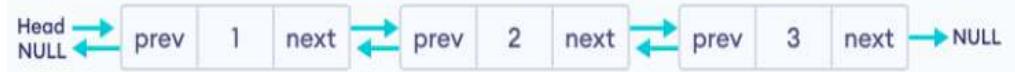
Element access:

front	Access first element <small>(public member function)</small>
back	Access last element <small>(public member function)</small>

Modifiers:

assign	Assign new content to container <small>(public member function)</small>
emplace_front	Construct and insert element at beginning <small>(public member function)</small>
push_front	Insert element at beginning <small>(public member function)</small>
pop_front	Delete first element <small>(public member function)</small>
emplace_back	Construct and insert element at the end <small>(public member function)</small>
push_back	Add element at the end <small>(public member function)</small>
pop_back	Delete last element <small>(public member function)</small>
emplace	Construct and insert element <small>(public member function)</small>
insert	Insert elements <small>(public member function)</small>
erase	Erase elements <small>(public member function)</small>
swap	Swap content <small>(public member function)</small>
resize	Change size <small>(public member function)</small>
clear	Clear content <small>(public member function)</small>

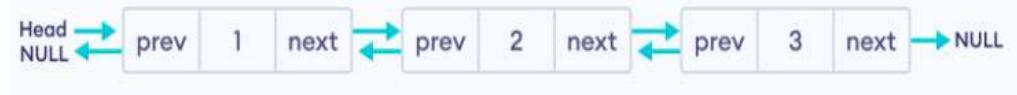
STL: list



Operations:

<u>splice</u>	Transfer elements from list to list (public member function)
<u>remove</u>	Remove elements with specific value (public member function)
<u>remove_if</u>	Remove elements fulfilling condition (public member function template)
<u>unique</u>	Remove duplicate values (public member function)
<u>merge</u>	Merge sorted lists (public member function)
<u>sort</u>	Sort elements in container (public member function)
<u>reverse</u>	Reverse the order of elements (public member function)

STL: list



```
// advance example
#include <iostream> // std::cout
#include <iterator> // std::advance
#include <list>      // std::list

int main()
{
    std::list<int> mylist;
    for (int i = 0; i < 10; i++)
    {
        mylist.push_back(i * 10);
    }

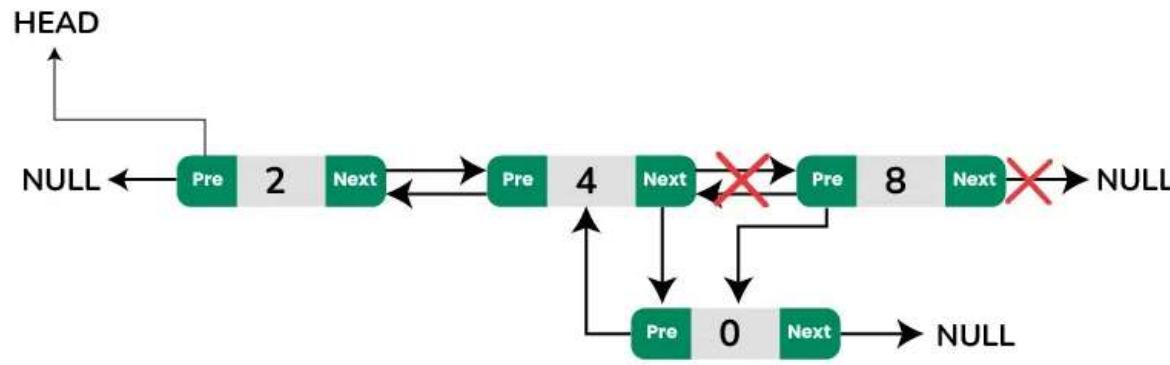
    std::list<int>::iterator it = mylist.begin();

    std::advance(it, 5);

    std::cout << "The sixth element in mylist is: " << *it << '\n';

    return 0;
}
```

STL: list - insert



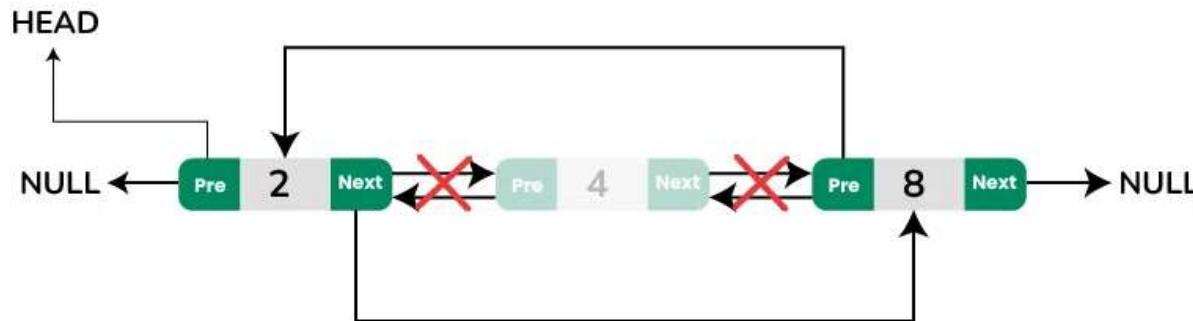
```
#include <iostream>
#include <list>
#include <vector>

int main ()
{
    std::list<int> mylist;
    for (int i=0; i<=4; ++i) mylist.push_back(i); // 0 1 2 3 4

    auto it = mylist.begin();
    std::advance(it, 3); // it points now to number 3 ^

    mylist.insert(it, 10); // 0 1 2 10 3 4
    for (it=mylist.begin(); it!=mylist.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    return 0;
}
```

STL: list - erase



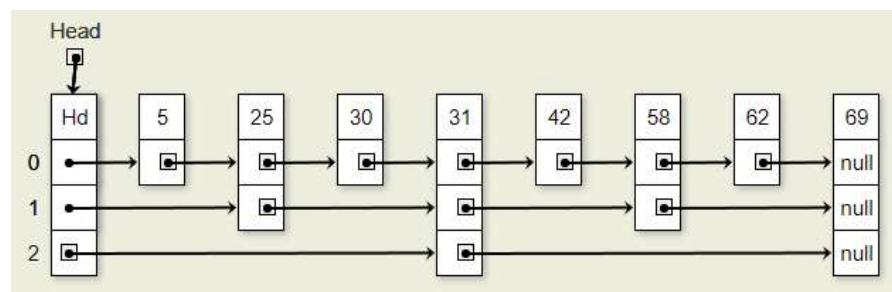
```
// erasing from list
#include <iostream>
#include <list>

int main ()
{
    std::list<int> mylist;

    // set some values:
    for (int i=1; i<10; ++i) mylist.push_back(i*10);
                                // 10 20 30 40 50 60 70 80 90
    auto it = mylist.begin();    // ^
    advance (it,6);            // ^
    it1 = mylist.erase(it);     // 10 20 30 40 50 60 80 90
                                // ^
    std::cout << "mylist contains:";
    for (it=mylist.begin(); it!=mylist.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    return 0;
}
```

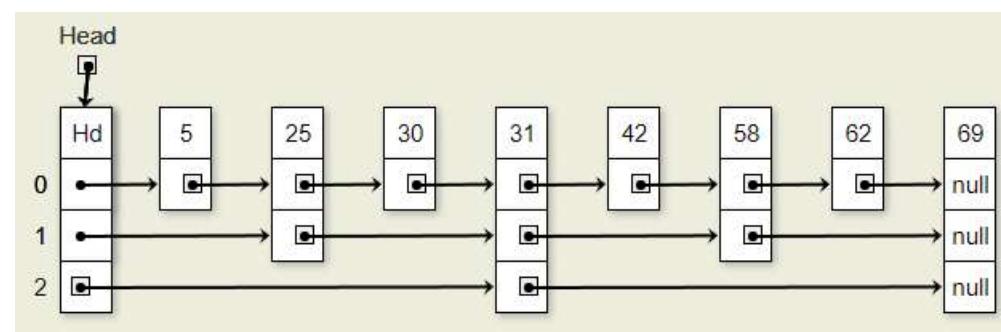
New Data Structure: Skip List

- limitation of linked lists
 - either search or update operations require linear time $O(n)$.
- skip lists are designed to overcome the limitation.
 - built upon the general idea of a linked list.
 - allow fast search, insertion and deletion $\Theta(\log n)$
 - values in a skip list are maintained in order at all times.

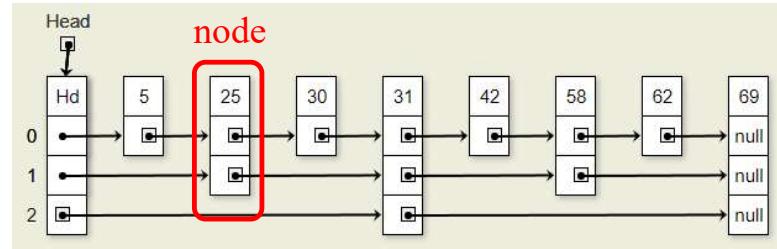
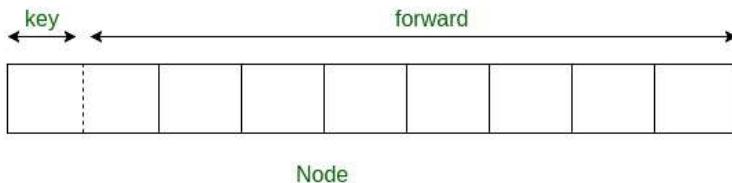


Skip List

- Keys in sorted order.
- Each higher level contains 1/2 the elements of the level below it -> $O(\log n)$ levels
- the lower layer works as a “normal lane” that connects every station.
- Called *skip lists* because higher level lists let you skip over many items

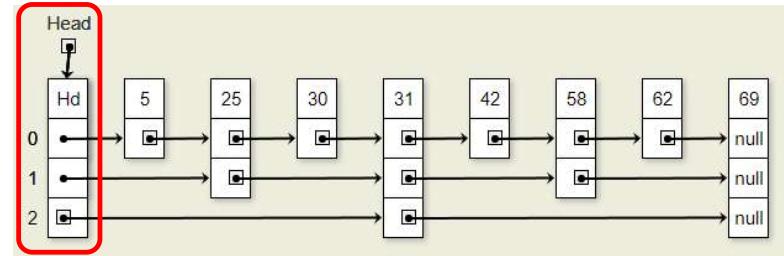


Skip List: Node



```
class Node {  
private:  
    int key;  
  
public:  
    vector<shared_ptr<Node>> forward;  
  
    Node(int k, int level) : key(k), forward(level + 1, nullptr) {}  
  
    int getKey() const { return key; }  
  
    ~Node() {  
        cout << "key " << key << " destroyed.\n";  
    }  
};
```

Skip List Structure

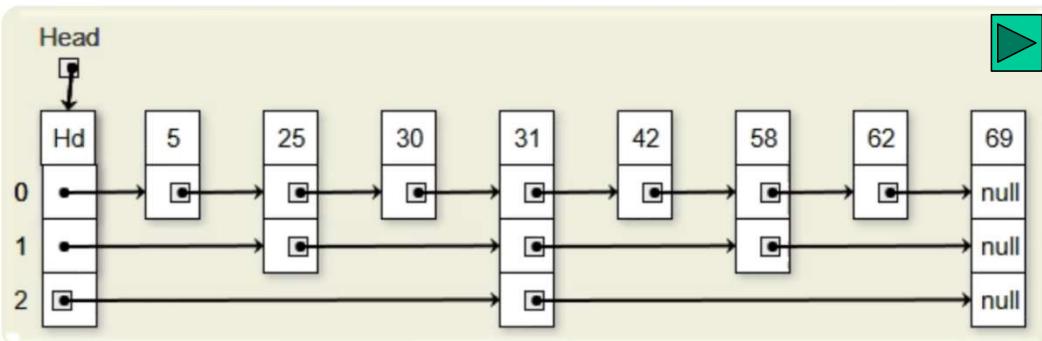


```
class SkipList {  
private:  
    int MAXLVL; //Maximum level of the skip list  
    float P; //Probability of node promotion to higher levels (typically 0.5).  
  
    int level;  
    shared_ptr<Node> header;  
public:  
    SkipList(int, float);  
    bool searchElement(int);  
    void insertElement(int);  
    void deleteElement(int);  
    void displayList();  
    int randomLevel();  
    ~SkipList() {  
        cout << "skip list destroyed.\n";  
    }  
};
```

- Search
- Insert
- Delete

Skip List – Search operation

demo: let's find node with value 62.



```
1 bool SkipList::searchElement(int key)
2 {
3     auto current = header;
4     cout << "Search Path:" << endl;
5     for (int i = level; i >= 0; i--)
6     {
7         while (current->forward[i] && current->forward[i]->getKey() < key)
8         {
9             current = current->forward[i];
10        }
11    }
12    current = current->forward[0];
13    return current && current->getKey() == key;
14 }
```

When search for k:

- If $k = \text{key}$, done!
- If $k < \text{next key}$, go down a level
- If $k \geq \text{next key}$, go right

```
class Node {
private:
    int key;

public:
    vector<shared_ptr<Node>> forward;
...
};
```

```
class SkipList {
private:
    int MAXLVL;
    float P;
    int level;
    shared_ptr<Node> header;
public:
    ...
};
```

Skip List – Insert operation

- Insert & delete might need to rearrange the entire list
- Perfect Skip Lists are too structured to support efficient updates.
- Idea:
 - Relax the requirement that each level have exactly half the items of the previous level
 - Instead: design structure so that we **expect** 1/2 the items to be carried up to the next level
 - Expect 1/2 the nodes at level 1
 - Expect 1/4 the nodes at level 2
 -

Skip List – Insert operation

- Idea:
 - Instead: design structure so that we **expect** 1/2 the items to be carried up to the next level
 - **Expect** 1/2 the nodes at level 1
 - **Expect** 1/4 the nodes at level 2
 -
 - Use probability to build subsequent layers
 - Therefore, skip lists are a randomized data structure

Skip List – randomLevel

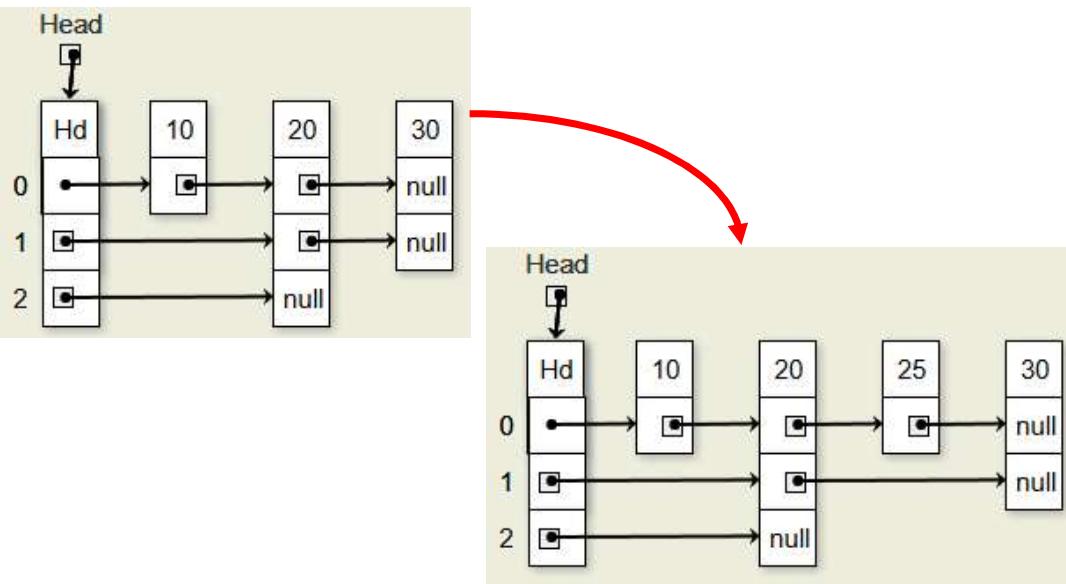
```
int SkipList::randomLevel()
{
// rand() generates an integer in the range [0, RAND_MAX]
    float r = (float)rand() / RAND_MAX;
    int lvl = 0;
    while (r < P && lvl < MAXLVL)
    {
        lvl++;
        r = (float)rand() / RAND_MAX;
    }
    return lvl;
}
```

- $P = 1/2$
- **Expect** $1/2$ the nodes at level 1
 - $1/2 (P)$ probability of returning $lvl = 1$
- **Expect** $1/4$ the nodes at level 2
 - $1/4 (P \cdot P)$ probability of returning $lvl = 2$
- **Expect** $1/8$ the nodes at level 3
 - $1/8 (P \cdot P \cdot P)$ probability of returning $lvl = 3$
- ...

Skip List – Insert operation

```
SkipList::SkipList(int MAXLVL, float P)
: MAXLVL(MAXLVL), P(P), level(0)
{
    header = make_shared<Node>(INT_MIN, MAXLVL);
}
```

demo: Insert value 25,
assuming randomLevel returns 0.

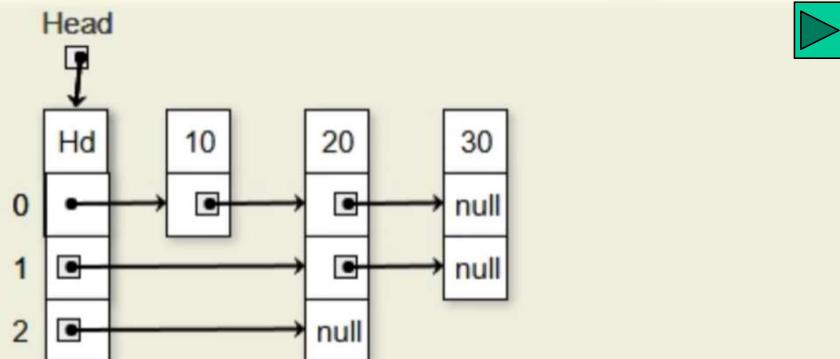


```
1 void SkipList::insertElement(int key)
2 {
3     auto current = header;
4     vector<shared_ptr<Node>> update(MAXLVL + 1, nullptr);
5     for (int i = level; i >= 0; i--)
6     {
7         while (current->forward[i] && current->forward[i]->getKey() < key)
8         {
9             current = current->forward[i];
10        }
11        update[i] = current;
12    }
13    current = current->forward[0];
14    if (!current || current->getKey() != key)
15    {
16        int rlevel = randomLevel();
17        if (rlevel > level)
18        {
19            for (int i = level + 1; i <= rlevel; i++)
20            {
21                update[i] = header;
22            }
23            level = rlevel;
24        }
25        auto n = make_shared<Node>(key, rlevel);
26        for (int i = 0; i <= rlevel; i++)
27        {
28            n->forward[i] = update[i]->forward[i];
29            update[i]->forward[i] = n;
30        }
31    }
32 }
```

Skip List – Insert operation

```
SkipList::SkipList(int MAXLVL, float P)
: MAXLVL(MAXLVL), P(P), level(0)
{
    header = make_shared<Node>(INT_MIN, MAXLVL);
}
```

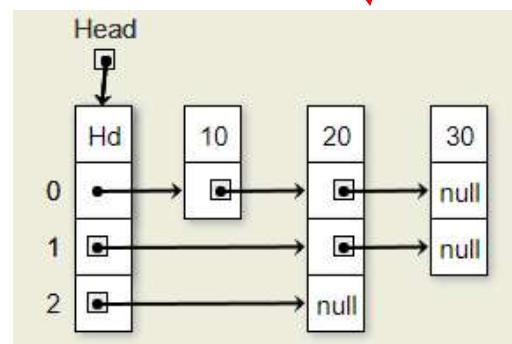
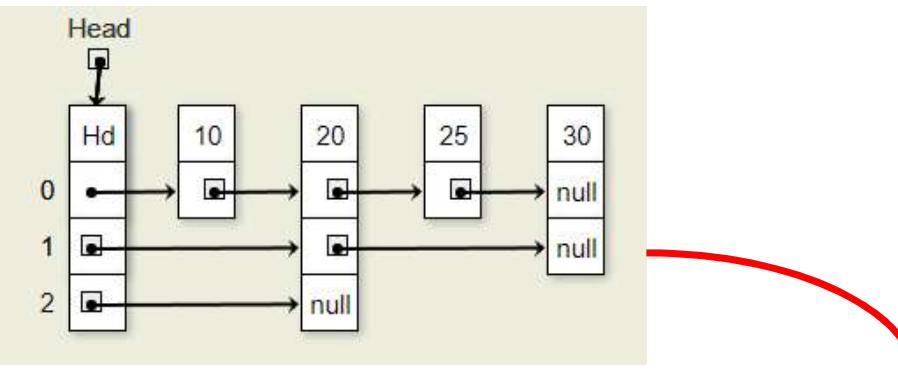
demo: Insert value 25,
assuming randomLevel returns 0.



```
1 void SkipList::insertElement(int key)
2 {
3     auto current = header;
4     vector<shared_ptr<Node>> update(MAXLVL + 1, nullptr);
5     for (int i = level; i >= 0; i--)
6     {
7         while (current->forward[i] && current->forward[i]->getKey() < key)
8         {
9             current = current->forward[i];
10        }
11        update[i] = current;
12    }
13    current = current->forward[0];
14    if (!current || current->getKey() != key)
15    {
16        int rlevel = randomLevel();
17        if (rlevel > level)
18        {
19            for (int i = level + 1; i <= rlevel; i++)
20            {
21                update[i] = header;
22            }
23            level = rlevel;
24        }
25        auto n = make_shared<Node>(key, rlevel);
26        for (int i = 0; i <= rlevel; i++)
27        {
28            n->forward[i] = update[i]->forward[i];
29            update[i]->forward[i] = n;
30        }
31    }
32 }
```

Skip List – Delete operation

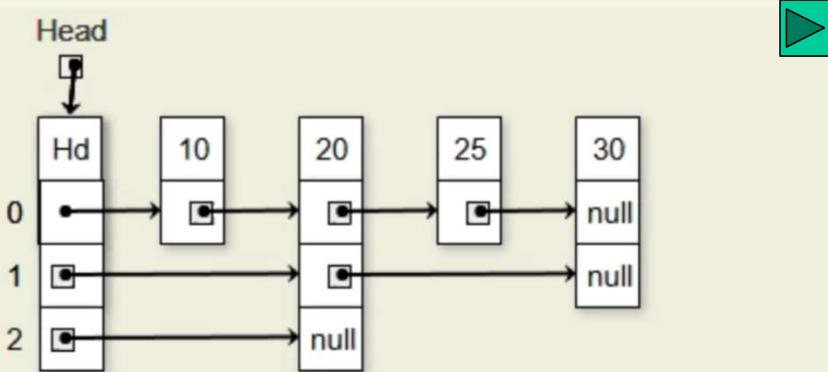
demo: remove the key 25.



```
1 void SkipList::deleteElement(int key)
2 {
3     auto current = header;
4     vector<shared_ptr<Node>> update(MAXLVL + 1, nullptr);
5     for (int i = level; i >= 0; i--)
6     {
7         while (current->forward[i] && current->forward[i]->getKey() < key)
8         {
9             current = current->forward[i];
10        }
11        update[i] = current;
12    }
13    current = current->forward[0];
14    if (current && current->getKey() == key)
15    {
16        for (int i = 0; i <= level; i++)
17        {
18            if (update[i]->forward[i] != current)
19                break;
20            update[i]->forward[i] = current->forward[i];
21        }
22        while (level > 0 && !header->forward[level])
23        {
24            level--;
25        }
26    }
27    cout << "Successfully deleted key " << key << "\n";
28 }
```

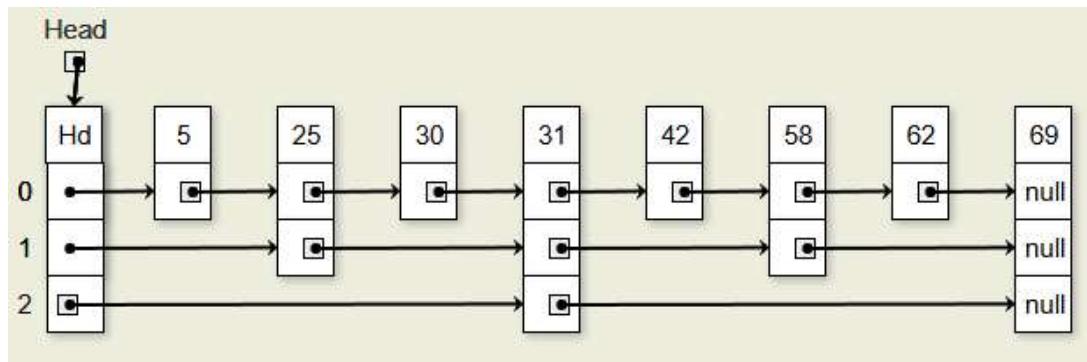
Skip List – Delete operation

demo: remove the key 25.



```
1 void SkipList::deleteElement(int key)
2 {
3     auto current = header;
4     vector<shared_ptr<Node>> update(MAXLVL + 1, nullptr);
5     for (int i = level; i >= 0; i--)
6     {
7         while (current->forward[i] && current->forward[i]->getKey() < key)
8         {
9             current = current->forward[i];
10        }
11        update[i] = current;
12    }
13    current = current->forward[0];
14    if (current && current->getKey() == key)
15    {
16        for (int i = 0; i <= level; i++)
17        {
18            if (update[i]->forward[i] != current)
19                break;
20            update[i]->forward[i] = current->forward[i];
21        }
22        while (level > 0 && !header->forward[level])
23        {
24            level--;
25        }
26        cout << "Successfully deleted key " << key << "\n";
27    }
28 }
29 }
```

Skip List – display



*****Skip List*****
Level 0: 5 25 30 31 42 58 62 69
Level 1: 25 31 58 69
Level 2: 31 69

```
void SkipList::displayList()
{
    cout << "\n*****Skip List*****\n";
    for (int i = 0; i <= level; i++)
    {
        auto node = header->forward[i];
        cout << "Level " << i << ": ";
        while (node)
        {
            cout << node->getKey() << " ";
            node = node->forward[i];
        }
        cout << "\n";
    }
}
```

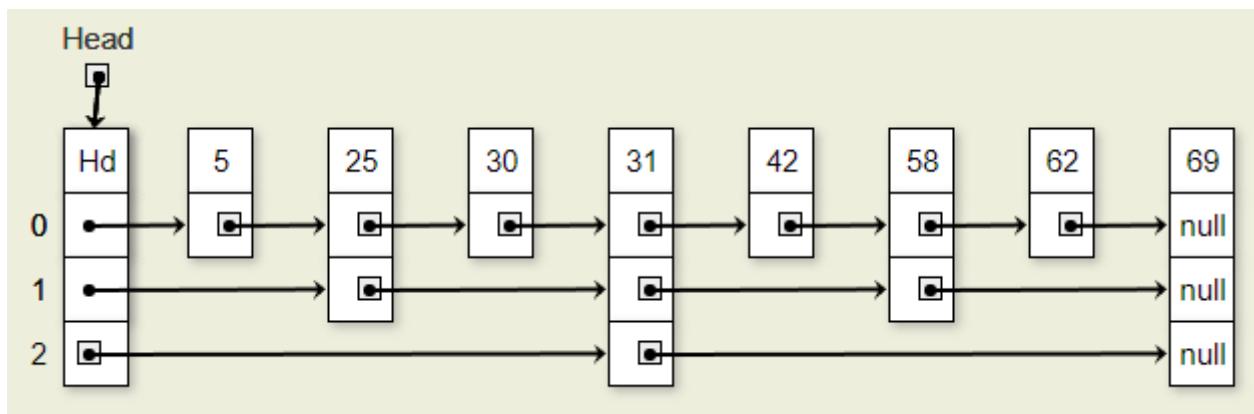
Skip List – Summary

- Time Complexity
 - $O(\log n)$: Search, Deletion, and Insertion
 - these bounds are expected or average-case bounds
- Space Complexity
 - $O(n)$
 - Suppose we have the total number of positions in our skip list equal to

$$n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots\right) = n \cdot 2$$

Review Week9

- STL Container
 - List
 - double linked list
- A New Data Structure
 - skip_list



2800ICT

Object Oriented Programming

C++ Data Structure

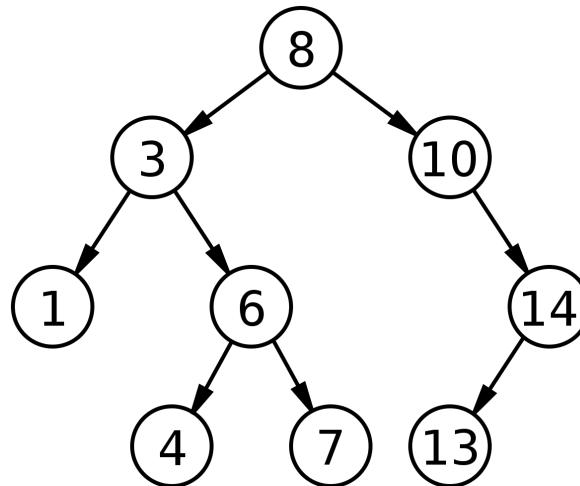
- treap

Topics

- Concepts:
 - Binary search tree
 - Binary heap
- Treap

Binary Search Tree

- Binary search tree is a tree data structure
- Each node can have at most two children
 - Left child has a lower or equal value
 - Right child has a higher value

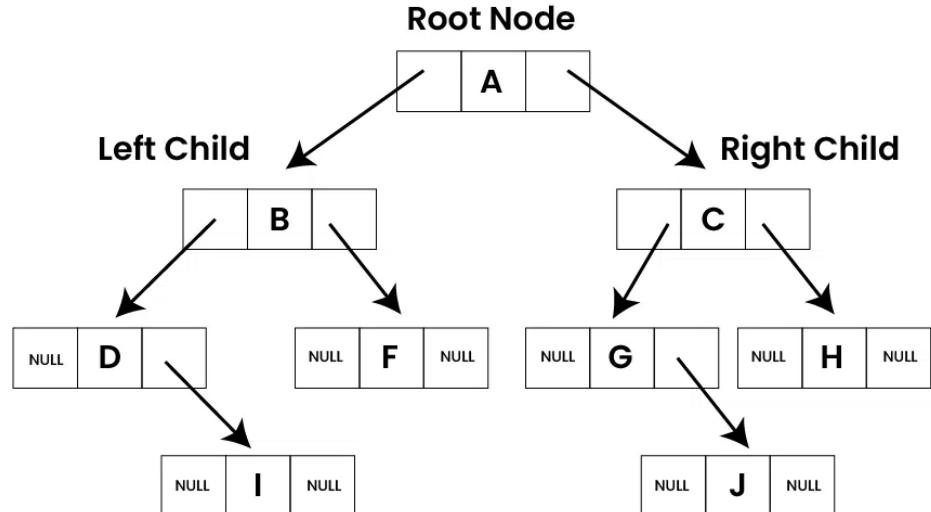


Binary Search Tree

- Each node consists of three parts
 - Data
 - Pointer to the left child
 - Pointer to the right child

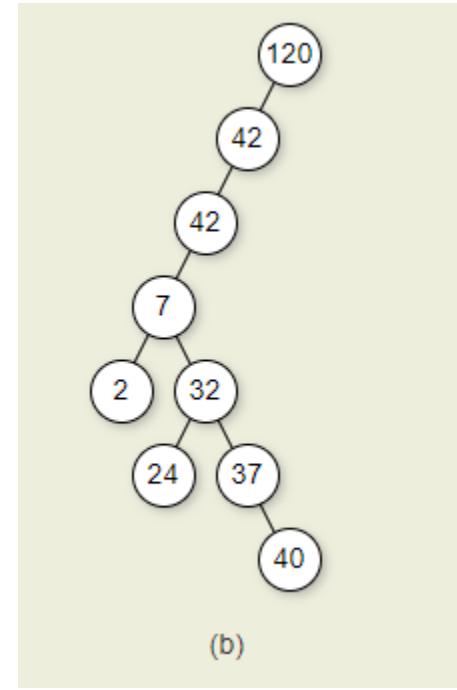
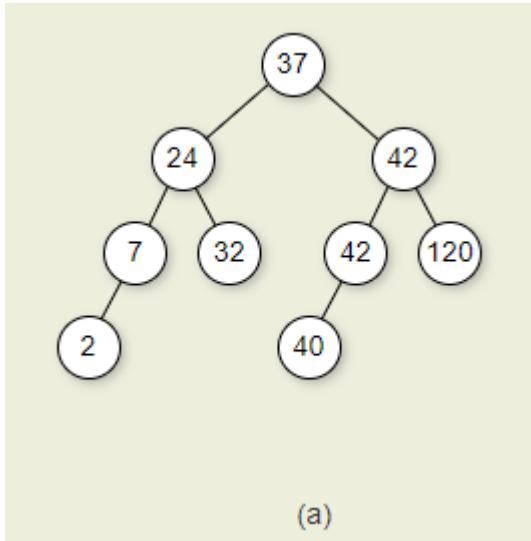
```
class Node
{
private:
    int data;
public:
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;

    // Constructor
    Node(int value) : data(value),
left(nullptr), right(nullptr) {}
};
```



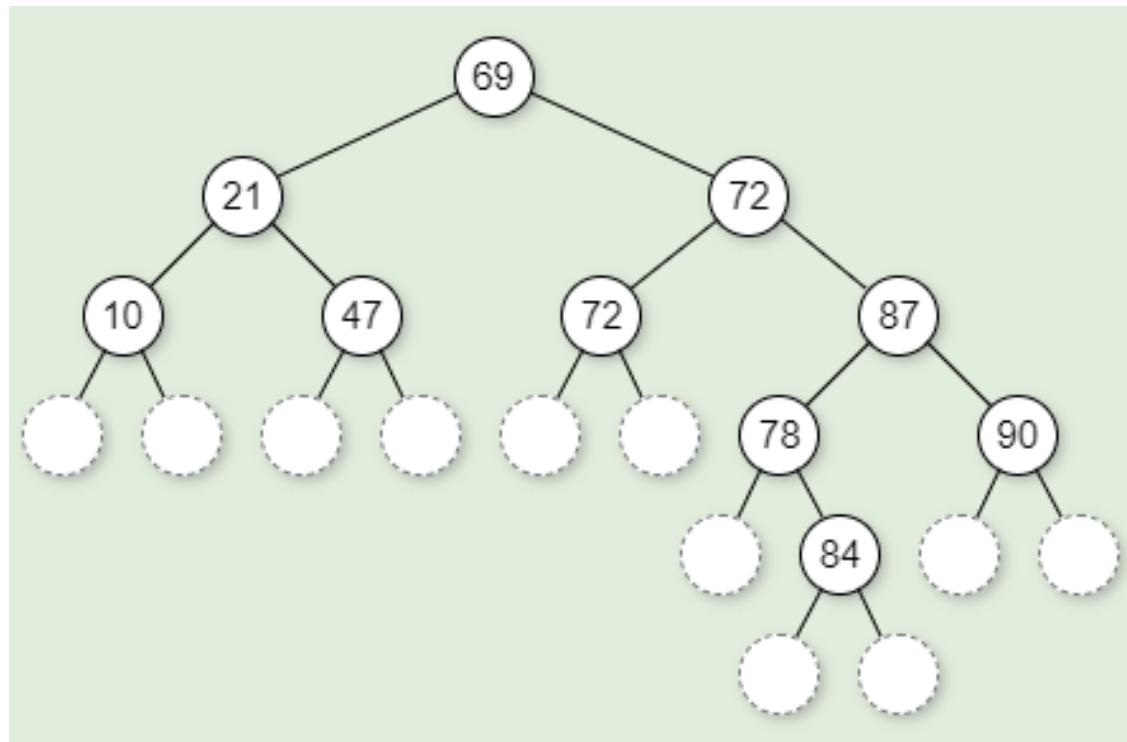
Binary Search Tree

- Binary tree or not?



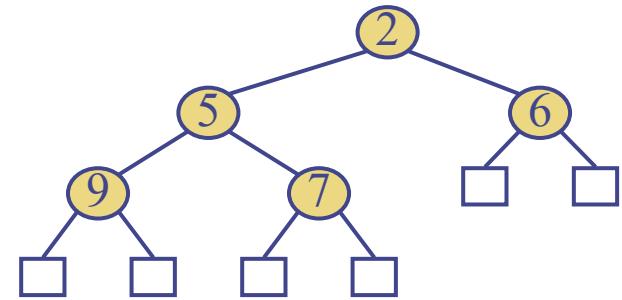
Binary Search Tree

- Demo: let's build a binary tree



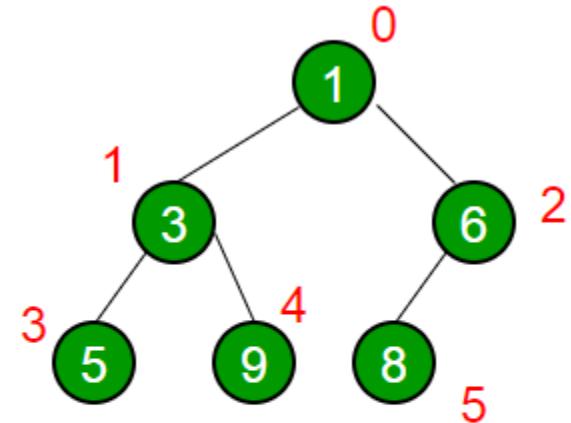
Binary Heap

- a **complete binary tree-based** data structure
 - **complete binary tree**
 - all levels are fully filled except possibly the last, which is filled from left to right.
 - Each node can have at most two children
 - Binary heap satisfies the heap property
 - Min binary heap
 - $n.priority \geq n.parent.priority$
 - Max binary heap
 - $n.priority \leq n.parent.priority$
- The primary use is to implement a priority queue.



Binary Heap

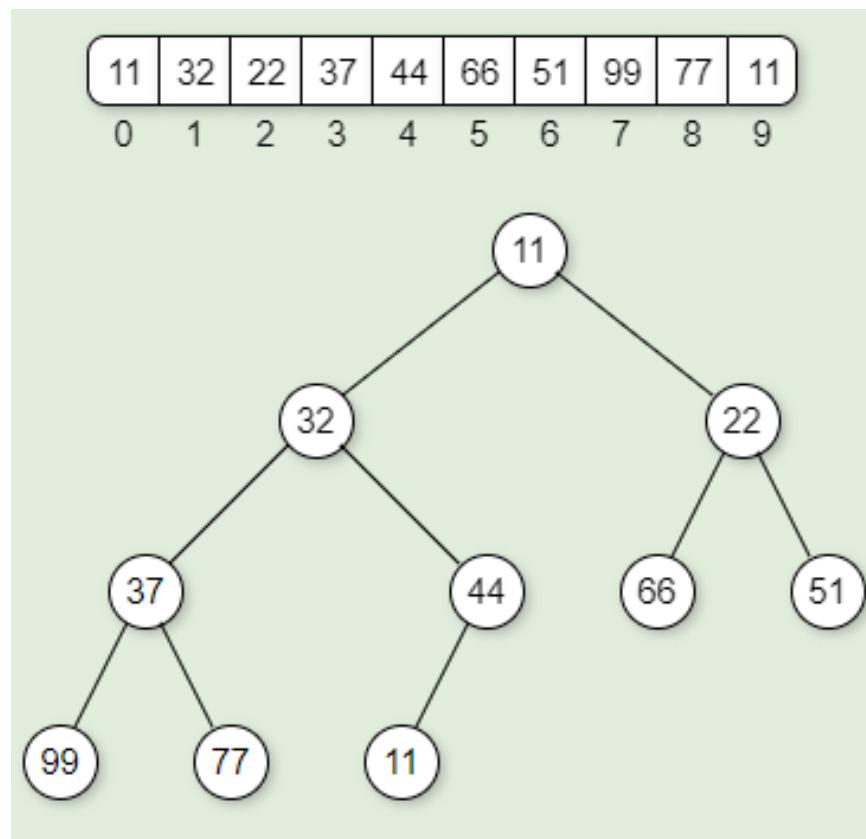
- A binary heap is typically represented as **an array**
 - The root element will be at $\text{Arr}[0]$.
 - For node i , i.e., $\text{Arr}[i]$:
 - $\text{Arr}[(i-1)/2]$: the parent node
 - $\text{Arr}[(2*i)+1]$: the left child node
 - $\text{Arr}[(2*i)+2]$: the right child node



1	3	6	5	9	8
0	1	2	3	4	5

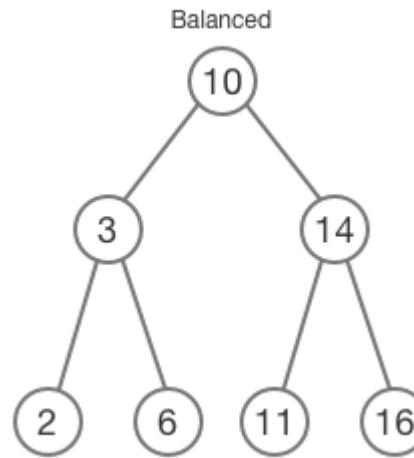
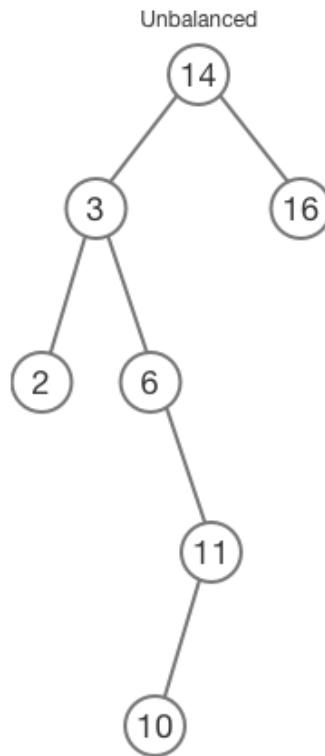
Binary Heap

- Demo: build a min binary heap



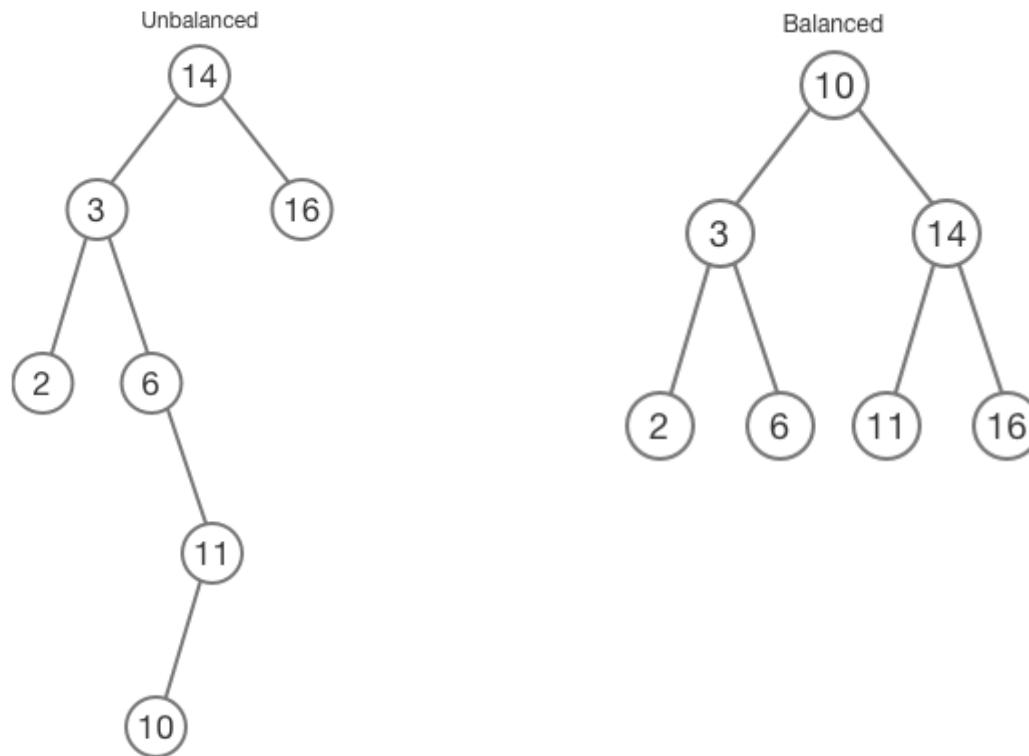
Balanced Binary Tree

- A **binary tree is balanced**
 - if, for **every node**, the height difference between its left and right subtrees is no more than a fixed constant, typically **1**.



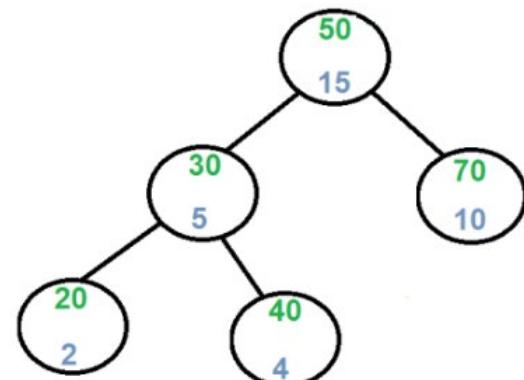
Balanced Binary Tree

- Question: how to maintain/build a balance binary tree after insertion and deletion
 - There are many ways, and a **trep** is one of them to **approximately** achieve balanced binary tree.



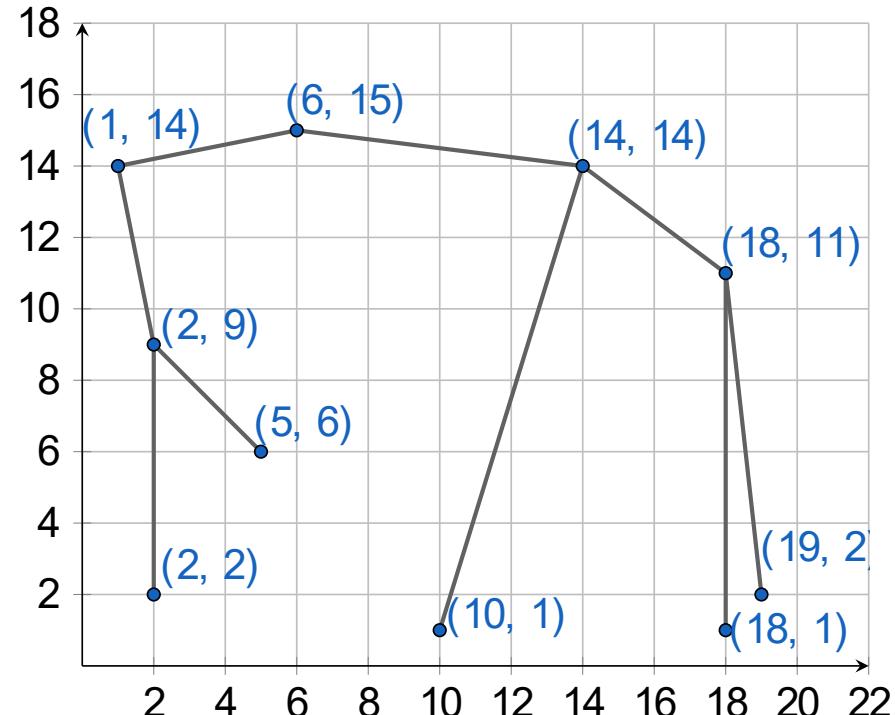
Treap

- A treap is a data structure which combines binary tree and binary heap
 - Tree + Heap => Treap
 - a **randomized balanced** binary search tree that also satisfies the **heap property**
- nodes contain two values:
 - a **key** and a **priority** (a random value)
 - the key obeys the BST property
 - the priority obeys the binary heap (min or max)



Treap

- A treap is also often referred to as a "cartesian tree".
- (key, priority)



Treap

```
template <typename T>
class TreapNode {
private:
    T key;
    int priority;
    shared_ptr<TreapNode<T>> left;
    shared_ptr<TreapNode<T>> right;

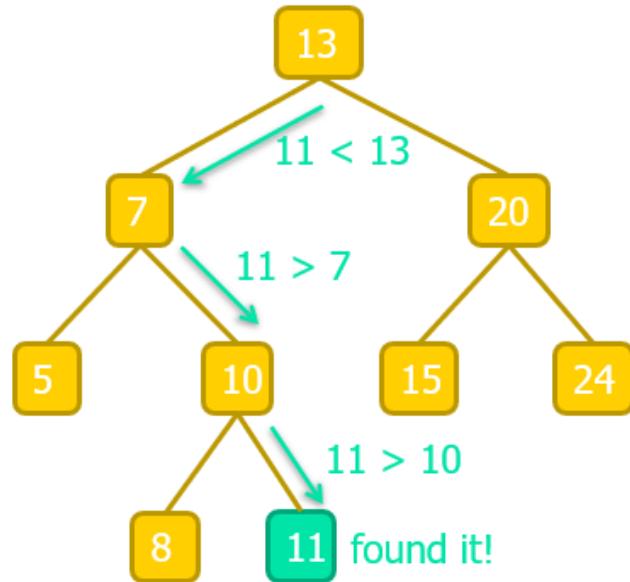
public:
    TreapNode(T key)
        : key(key), priority(rand()%100), left(nullptr), right(nullptr) {}
    TreapNode(T key, int priority)
        : key(key), priority(priority), left(nullptr), right(nullptr) {}
    ~TreapNode() {
        cout << "Key " << key << " destroyed." << endl;
    }
    ...
}
```

Treap - Search

- It's exactly the same algorithm used in BST.

Suppose we're looking for key 11

```
shared_ptr<TreapNode<T>> search(shared_ptr<TreapNode<T>>& root, T key) {  
    if (!root || root->key == key)  
        return root;  
    if (root->key < key)  
        return search(root->right, key);  
    if (root->key > key)  
        return search(root->left, key);  
    return nullptr; // To prevent warning  
}
```

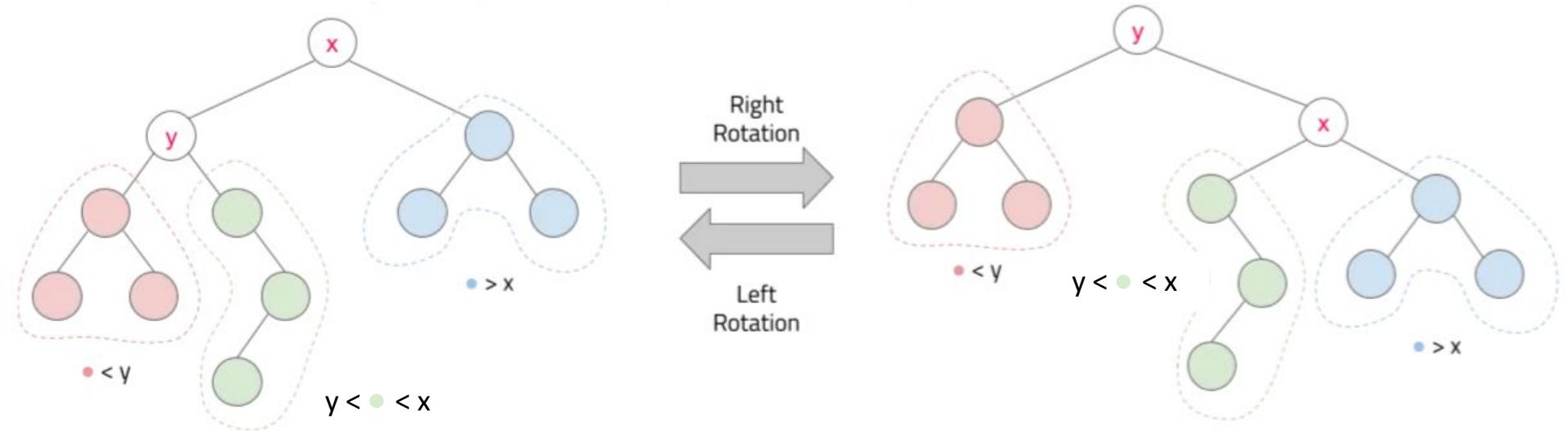


Treap - Insertion

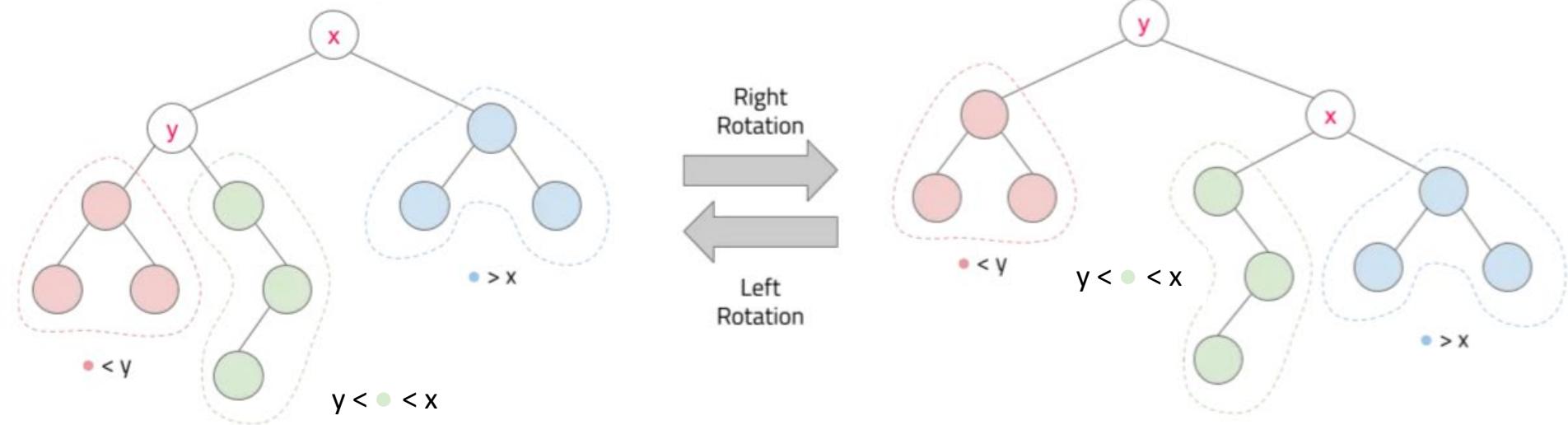
- Insert (X, P) a new node to the tree.
 - X : key
 - P : priority (usually a random number, max heap)
- two simple operations modify the tree and keep the BST property
 - Right rotation
 - Left rotation

Treap - Insertion

- two simple operations keep the BST property
 - Right rotation
 - Left rotation
- Nodes: x, y



Treap - Insertion



```
void right_rotation(shared_ptr<TreapNode<T>>& x) {
    shared_ptr<TreapNode<T>> y = x->left;
    shared_ptr<TreapNode<T>> g = y->right;
    shared_ptr<TreapNode<T>> b = x->right;

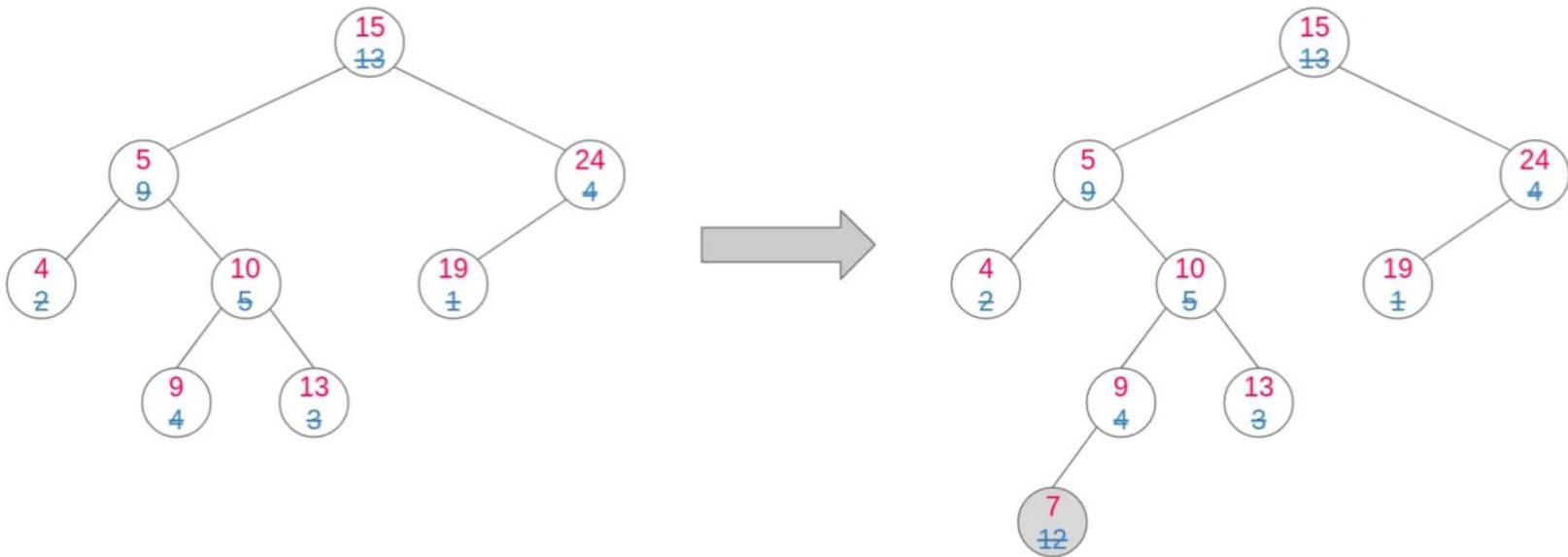
    x->left = g;
    y->right = x;
    x = y;
}
```

```
void left_rotation(shared_ptr<TreapNode<T>>& y) {
    shared_ptr<TreapNode<T>> x = y->right;
    shared_ptr<TreapNode<T>> g = x->left;
    shared_ptr<TreapNode<T>> b = x->right;

    x->left = y;
    y->right = g;
    y = x;
}
```

Treap - Insertion

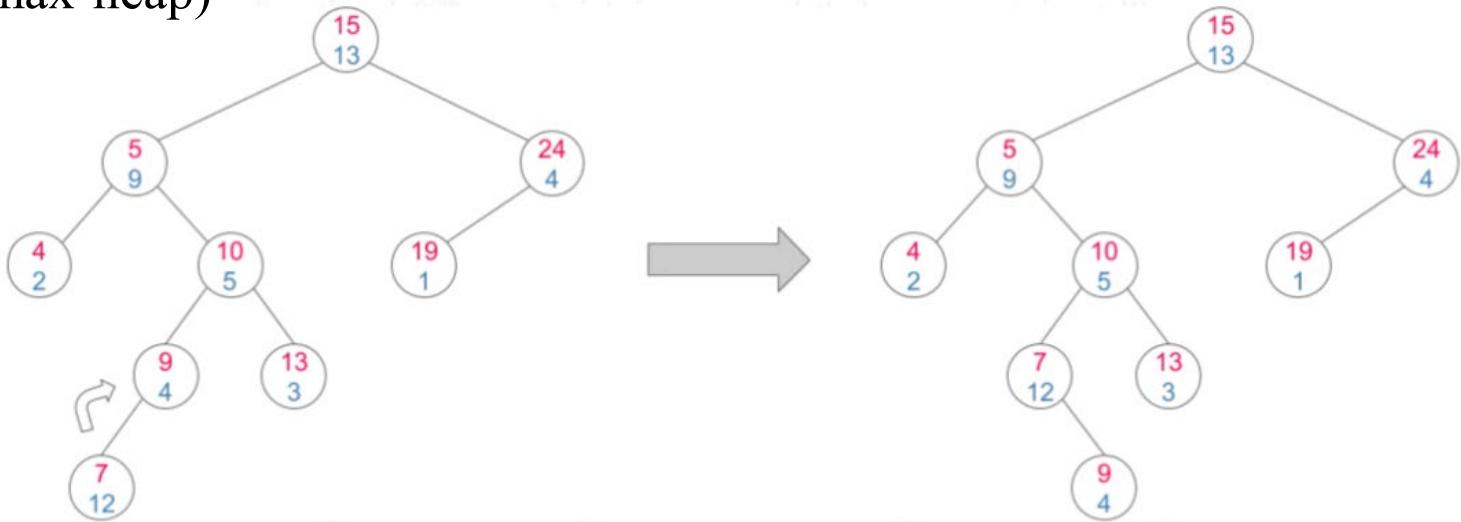
- Insert (7,12) into the treap
 - Insert the pair as a new leaf, using the BST.



```
void insert(shared_ptr<TreapNode<T>>& root, T key) {
    if (!root) {
        root = make_shared<TreapNode<T>>(key);
        return;
    }
    insert(key <= root->key ? root->left : root->right, key);
    if (root->left && root->left->priority > root->priority)
        right_rotation(root);
    if (root->right && root->right->priority > root->priority)
        left_rotation(root);
}
```

Treap - Insertion

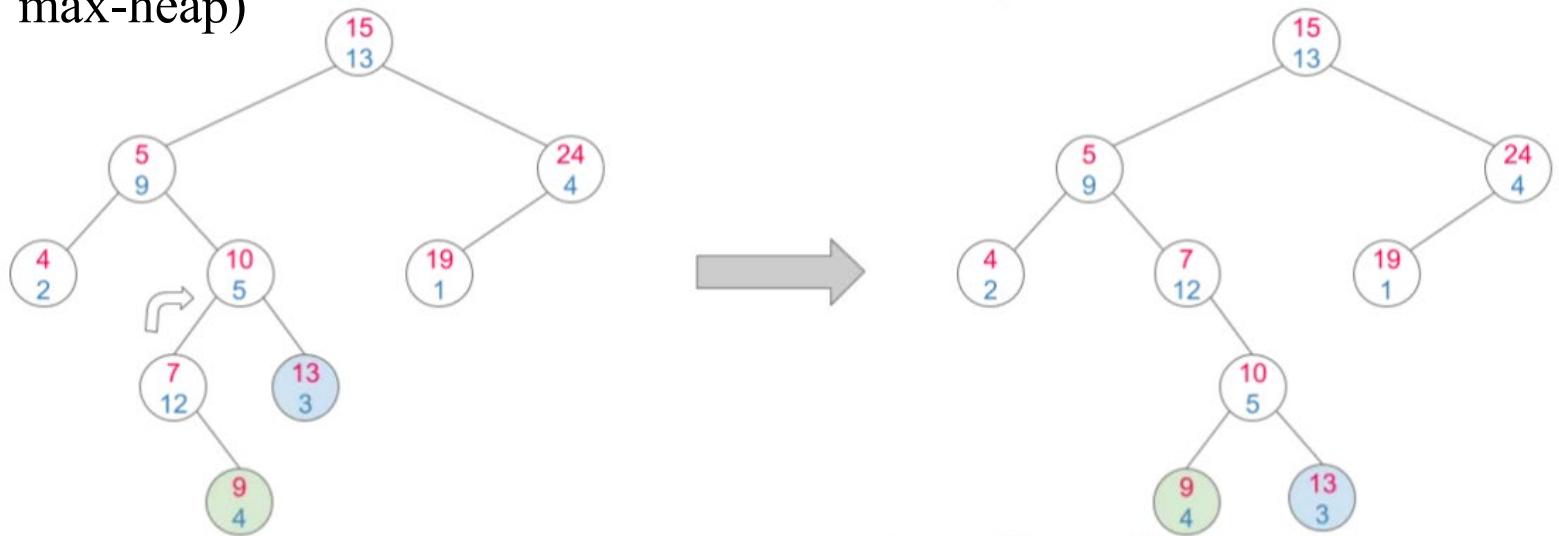
- Insert (7,12) into the treap
 - then rotate the node up to ensure priority obeying the binary heap (here max-heap)



```
void insert(shared_ptr<TreapNode<T>>& root, T key) {  
    1 if (!root) {  
    2         root = make_shared<TreapNode<T>>(key);  
    3         return;  
    4     }  
    5     insert(key <= root->key ? root->left : root->right, key);  
    6     if (root->left && root->left->priority > root->priority)  
    7         right_rotation(root);  
    8     if (root->right && root->right->priority > root->priority)  
        left_rotation(root);  
}
```

Treap - Insertion

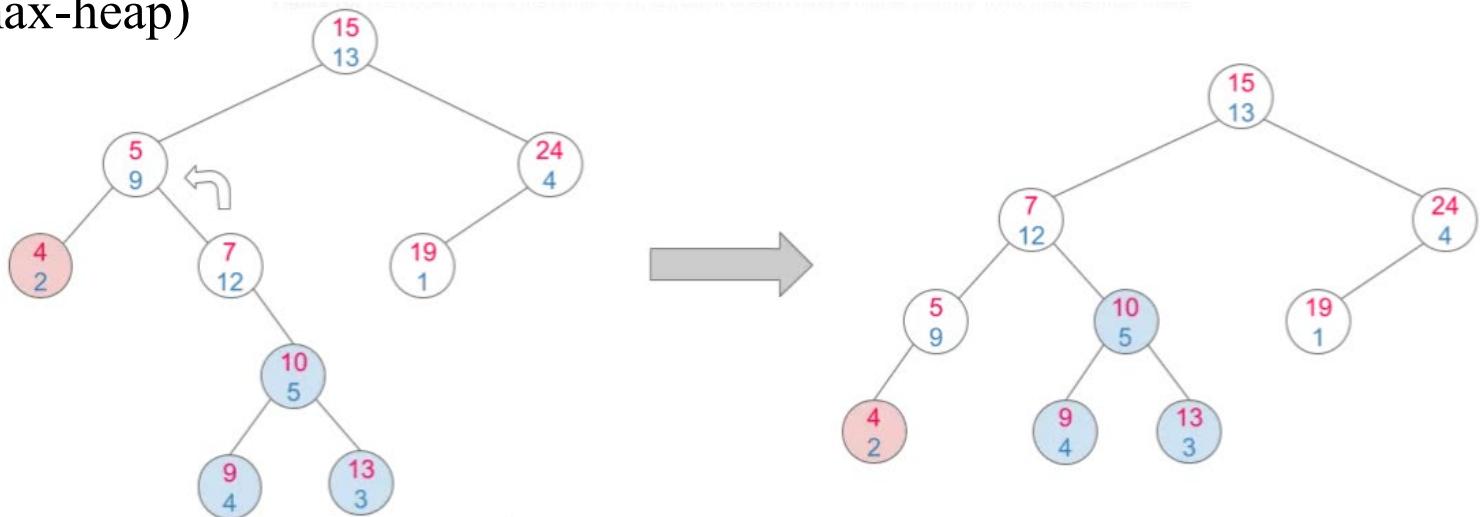
- Insert (7,12) into the treap
 - then rotate the node up to ensure priority obeying the binary heap (here max-heap)



```
void insert(shared_ptr<TreapNode<T>>& root, T key) {  
    1 if (!root) {  
    2         root = make_shared<TreapNode<T>>(key);  
    3         return;  
    4     }  
    5     insert(key <= root->key ? root->left : root->right, key);  
    6     if (root->left && root->left->priority > root->priority)  
    7         right_rotation(root);  
    8     if (root->right && root->right->priority > root->priority)  
        left_rotation(root);  
}
```

Treap - Insertion

- Insert (7,12) into the treap
 - then rotate the node up to ensure priority obeying the binary heap (here max-heap)



```
void insert(shared_ptr<TreapNode<T>>& root, T key) {
    1 if (!root) {
    2     root = make_shared<TreapNode<T>>(key);
    3     return;
    4 }
    5 insert(key <= root->key ? root->left : root->right, key);
    6 if (root->left && root->left->priority > root->priority)
    7     right_rotation(root);
    8 if (root->right && root->right->priority > root->priority)
        left_rotation(root);
}
```

Treap – Offline Build

- **Insert** is $O(\log N)$.
- Given an array, if insert one by one, it will take $O(N \log N)$.
- If we have all the keys sorted, we can build the treap in $O(n)$ time complexity.
 - BST can be easily constructed in linear time
 - split the array into two parts,
 - apply the same rule recursively to each part
 - The heap values are initialized randomly and then can be heapified independent of the keys in linear time.

Treap – Offline Build

```
shared_ptr<TreapNode<T>> build(vector<T>& v, int l, int r) {
    if (l > r)
        return nullptr;

    int m = (l + r) / 2;
    shared_ptr<TreapNode<T>> t = make_shared<TreapNode<T>>(v[m]);
    t->left = build(v, l, m - 1);
    t->right = build(v, m + 1, r);

    heapify(t);
    return t;
}

void heapify(shared_ptr<TreapNode<T>> t) {
    if (!t) return;

    shared_ptr<TreapNode<T>> max = t;
    if (t->left && t->left->priority > max->priority)
        max = t->left;
    if (t->right && t->right->priority > max->priority)
        max = t->right;

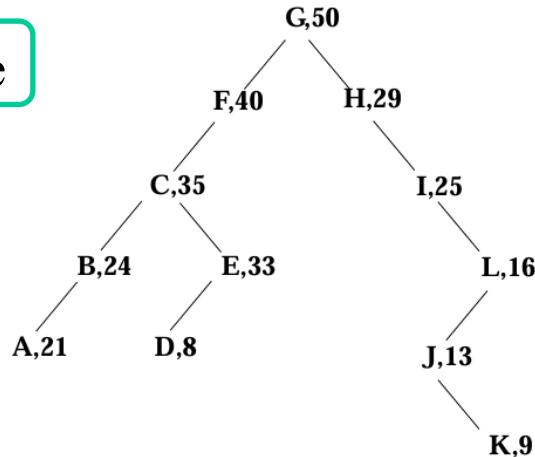
    if (max != t) {
        swap(t->priority, max->priority);
        heapify(max);
    }
}
```

Treap - Deletion

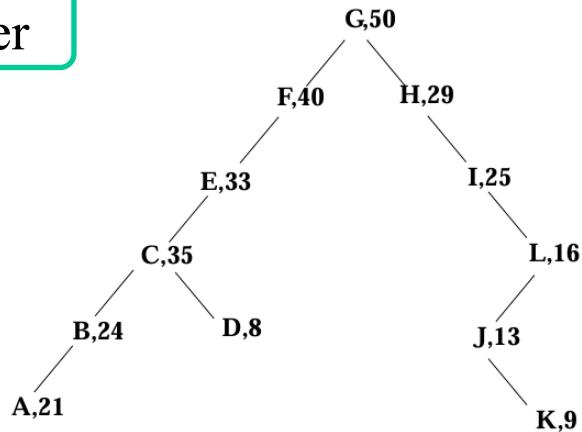
- To delete a key K, do the following:
 - Search for the node X containing K using the usual BST find algorithm
 - If the node X is a leaf, just delete the node
 - Otherwise, use rotations to rotate the node down until it becomes a leaf; then delete it
 - If there are 2 children, always rotate with the child that has the larger priority, to preserve heap ordering
- Since rotations are constant-time operations, delete in a treap can be performed in time $O(h)$, where h is the height of the treap

Delete the key C from this treap

before



after



```
bool remove(shared_ptr<TreapNode<T>>& root, T key) {
    1 if (!root)
    2     return false;
    3
    4 if (key < root->key)
    5     return remove(root->left, key);
    6 if (key > root->key)
    7     return remove(root->right, key);
    8
    9 if (!root->left && !root->right) {
    10     root.reset();
    11 } else if (!root->left || !root->right) {
    12     shared_ptr<TreapNode<T>>
    13     child = (root->left) ? root->left : root->right;
    14     root = child;
    15 } else {
    16     if (root->left->priority < root->right->priority) {
    17         left_rotation(root);
    18         remove(root->left, key);
    19     } else {
    20         right_rotation(root);
    21         remove(root->right, key);
    22     }
    23 }
    24 return true;
}
```

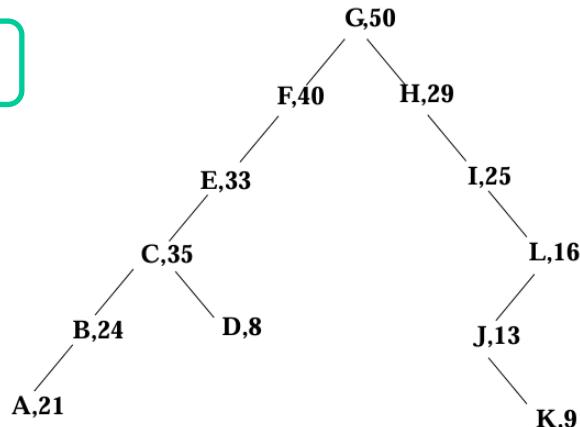
Find the node containing C.

It is not a leaf, so need to rotate it down;

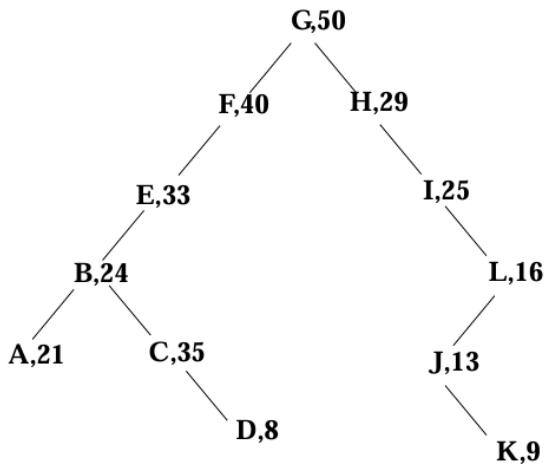
Rotating the node with its larger priority child.

Delete the key C from this treap

before



after



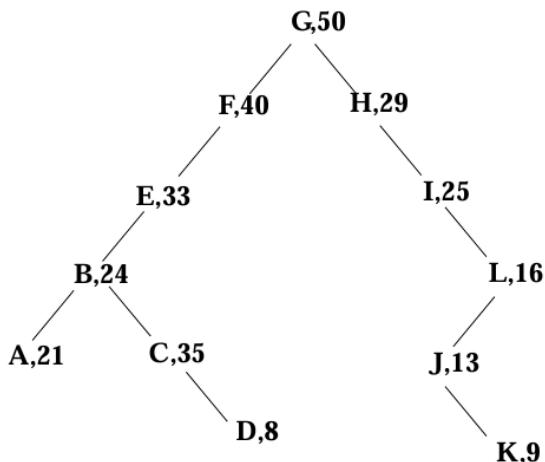
```
bool remove(shared_ptr<TreapNode<T>>& root, T key) {  
    1 if (!root)  
    2     return false;  
    3  
    4 if (key < root->key)  
    5     return remove(root->left, key);  
    6 if (key > root->key)  
    7     return remove(root->right, key);  
    8  
    9 if (!root->left && !root->right) {  
10     root.reset();  
11 } else if (!root->left || !root->right) {  
12     shared_ptr<TreapNode<T>>  
13     child = (root->left) ? root->left : root->right;  
14     root = child;  
15 } else {  
16     if (root->left->priorit y < root->right->priorit y) {  
17         left_rotation(root);  
18         remove(root->left, key);  
19     } else {  
20         right_rotation(root);  
21         remove(root->right, key);  
22     }  
23 }  
24 return true;  
}
```

The node C is still not a leaf, so need to rotate down again.

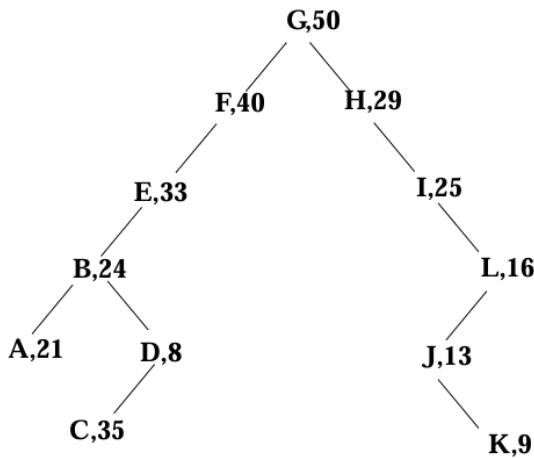
Rotating the node C with its larger priority child

Delete the key C from this treap

before



after



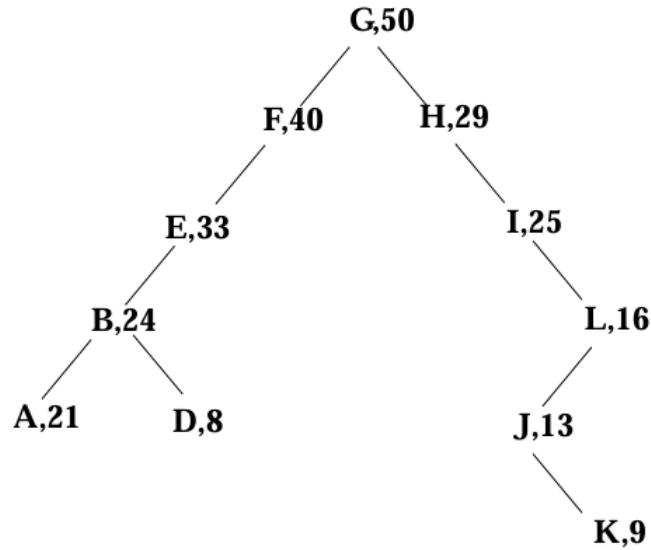
```
bool remove(shared_ptr<TreapNode<T>>& root, T key) {  
    1 if (!root)  
    2     return false;  
    3  
    4 if (key < root->key)  
    5     return remove(root->left, key);  
    6 if (key > root->key)  
    7     return remove(root->right, key);  
    8  
    9 if (!root->left && !root->right) {  
   10     root.reset();  
   11 } else if (!root->left || !root->right) {  
   12     shared_ptr<TreapNode<T>>  
   13     child = (root->left) ? root->left : root->right;  
   14     root = child;  
   15 } else {  
   16     if (root->left->priority < root->right->priority) {  
   17         left_rotation(root);  
   18         remove(root->left, key);  
   19     } else {  
   20         right_rotation(root);  
   21         remove(root->right, key);  
   22     }  
   23 }  
   24 return true;  
}
```

The node C is still not a leaf, so need to rotate down again.

Now the node containing C is a leaf, so just delete it.

Delete the key C from this treap

result



After clipping off the leaf, the result is again a treap.

```
bool remove(shared_ptr<TreapNode<T>>& root, T key) {
    1 if (!root)
    2     return false;
    3
    4 if (key < root->key)
    5     return remove(root->left, key);
    6 if (key > root->key)
    7     return remove(root->right, key);
    8
    9 if (!root->left && !root->right) {
    10     root.reset();
    11 } else if (!root->left || !root->right) {
    12     shared_ptr<TreapNode<T>>
    13     child = (root->left) ? root->left : root->right;
    14     root = child;
    15 } else {
    16     if (root->left->priority < root->right->priority) {
    17         left_rotation(root);
    18         remove(root->left, key);
    19     } else {
    20         right_rotation(root);
    21         remove(root->right, key);
    22     }
    23 }
    24 return true;
}
```

Summary

- $\text{Insert}(x,y)$ in $O(\log N)$
- $\text{Delete}(x)$ in $O(\log N)$
- Offline Build can be in $O(N)$
- $\text{Search}(x)$ in $O(\log N)$

2800ICT
Object Oriented Programming

Final Exam & Course Review

Open-book Final Exam

- 8:00 AM - 10:10 AM on Saturday, June 7th, Brisbane Time (AEST).
- 2-hour writing time + 10 minutes reading time
- Open book
- Online exam (course website on Canvas), not ProctorU
- will have 3 programming questions
 - Workshop-like questions

THANK
YOU!

Student Experience of Course and Teaching Surveys

Winners



- Completing the survey is **easy, on-line, and takes only a few minutes**
www.griffith.edu.au/experience
✓ SA (Strongly Agree) if you enjoy this course!
- **Confidentiality** - information is provided to school anonymously
- Students can nominate teaching staff for an award.
- Every course survey you complete enters you into the draw to win a share of the cash prize pool of \$10,000 each year.

Review Week1

- C++ Basic Input/Output & C++ Building Process
- Basic C++ Programming
- Three commonly-used data types:
 - std::string, std::array, std::vector
- C++ Function
 - Definition, Overloading, Templates, Default Arguments
- Reference variables

Review Week2

- Introduction to STL
- STL Container Fundamentals
- STL Iterator Fundamentals
- The **array** Class
- The **vector** Class
- The **set** Classes
- The **map** Classes <next week>

```
#include <array>
#include <iostream>
using namespace std;
int main()
{
    array<int, 5> myArray {1, 2, 3, 4, 5};
    for (auto it = myArray.begin(); it != myArray.end(); ++it)
    {
        cout << *it << " ";
    }
    cout << endl;
    return 0;
}
```

Review Week3

STL Container – Map class

STL - Algorithms

- **Min/max** algorithms
- **Sorting** algorithms
- **Search** algorithms
- Read-only sequence algorithms
- **Copying** and **moving** algorithms
- Swapping algorithms
- Replacement algorithms
- **Removal** algorithms
- Reversal algorithms
- **Fill** algorithms
- Rotation algorithms
- **Shuffling** algorithms
- **Set** algorithms
- Transformation algorithm
- Partition algorithms
- **Merge** algorithms
- Permutation algorithms
- Heap algorithms
- Lexicographical comparison algorithm

Review Week 4

- Introduction to Classes
- Defining a Class Instance
- Constructors and Destructors
- The this Pointer
- Operator Overloading
- Class Templates

Review Week 5

- Classes
 - Static Members and Friend Functions
- Copying Objects
 - Copy Constructor/Assignment
 - Lvalue & Rvalue
 - Move Constructor/Assignment
- Unified Modelling Language (UML)
- Class Aggregation

Review Week 6

- Inheritance
 - Class Inheritance
 - Protected Members and Class Access
 - Constructors and Destructors in Base and Derived Classes
 - Redefining Base Class Functions
- Polymorphism
 - Virtual Member Functions
- Abstract Base Class
 - Pure Virtual Functions
 - Multiple Inheritance

Review Week 7

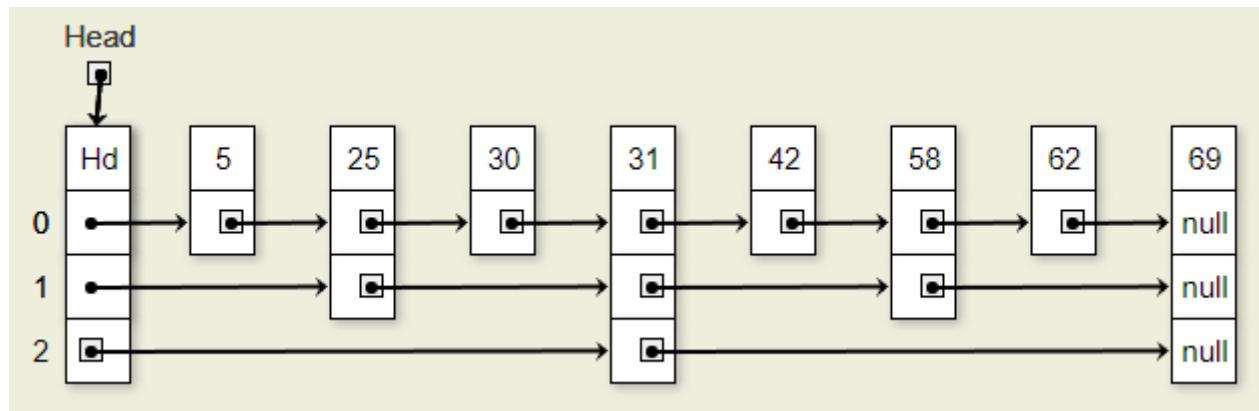
- Exception Handling Basics
 - Handling Multiple Exceptions
 - Stack Unwinding
 - Using Standard Exceptions
 - Custom Exception Class
-
- Function Object/Functor
 - operator ()
 - Anonymous Function Objects
 - Lambda Expressions

Review Week 8

- Smart pointers
 - unique_ptr
 - shared_ptr
 - weak_ptr
- Multi-Threading
 - std::thread threadObj(<function>, <args>...);
 - threadObj.join()
 - push_back vs emplace_back

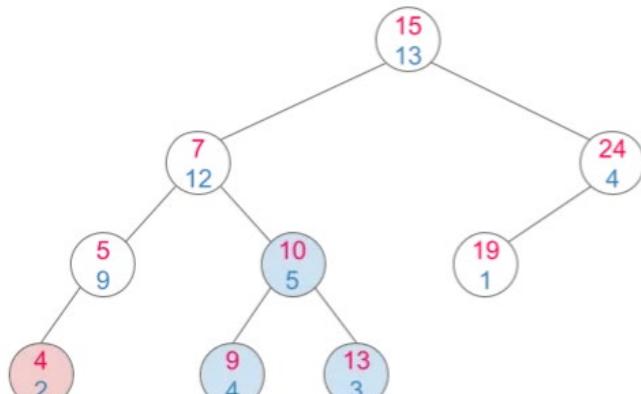
Review Week 9

- STL Container
 - List
 - double linked list
- A New Data Structure
 - skip_list



Review Week 10

- Concepts:
 - Binary search tree
 - Binary heap
- Treap





Thank you for your company and engagement throughout this L&T journey!

Wishing you all the best in the future!

Hope to see you around!

Happy Coding!