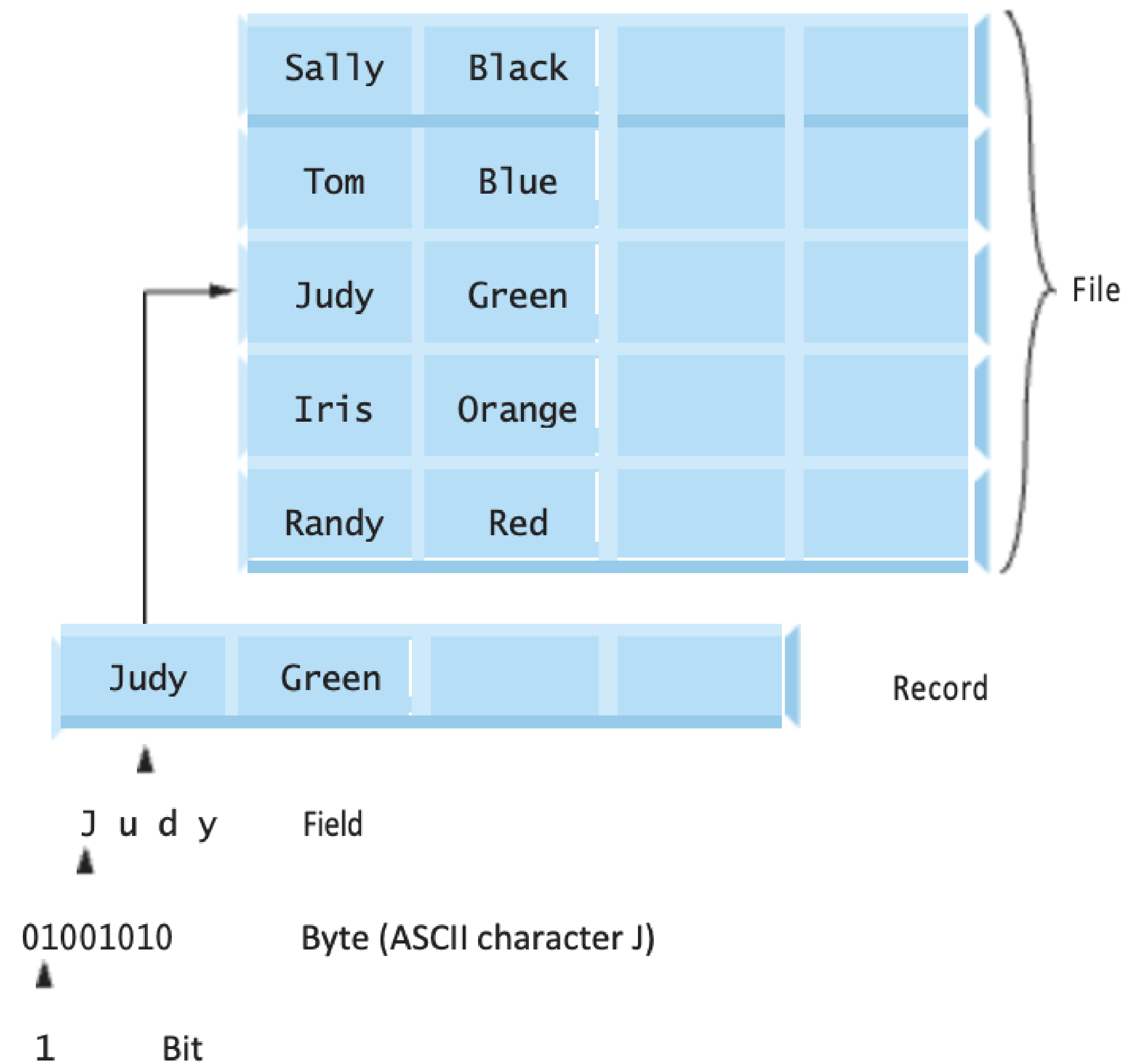
Object-Oriented Programming

Lecture 3: Introduction to Class Vector

Computer Organization

- Input unit
- Output unit
- Memory unit
- Arithmetic and logic unit (ALU)
- Central processing unit (CPU)
- Secondary storage unit

Data Hierachy



Computer Language

- Machine Languages
- Assembly Languages
- High-Level Languages
- Interpreters
- Compilers

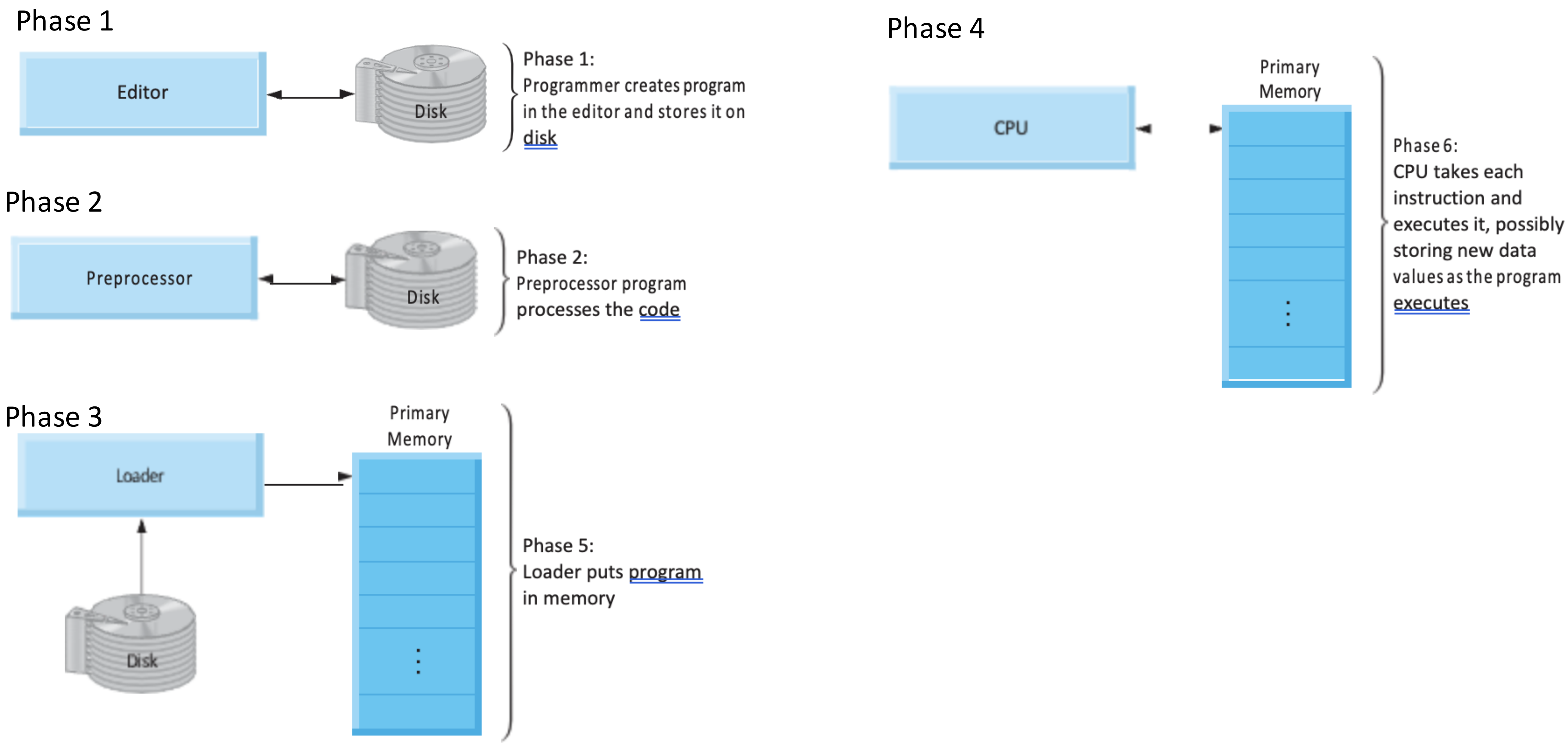
Introduction to Object Technology

- Functions
- Member functions
- Class
- Instantiation
- Message and memberfunction calls
- Attributes and data members

Introduction to Object Technology

- Encapsulations
- Inheritance – New class
- Object-Oriented Analysis and Design (OOAD)
- UML (Unified Modeling Language) - graphical scheme for modeling object-oriented systems

Typical C++ Development Environment



First Program in C++: Printing a Line of Text

```cpp
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
#include <iostream> // enables program to output data to the screen

// function main begins program execution
int main() {
   std::cout << "Welcome to C++!\n"; // display message

   return 0; // indicate that program ended successfully
} // end function main
```

```
Welcome to C++!
```

Escape sequence	Description
\n	Newline. Position the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert. Sound the system bell.
\\	Backslash. Used to print a backslash character.
\'	Single quote. Used to print a single-quote character.
\"	Double quote. Used to print a double-quote character.

Another C++ Program: Adding Integers

```cpp
// Fig. 2.5: fig02_05.cpp
// Addition program that displays the sum of two integers.
#include <iostream> // enables program to perform input and output

// function main begins program execution
int main() {
   // declaring and initializing variables
   int number1{0}; // first integer to add (initialized to 0)
   int number2{0}; // second integer to add (initialized to 0)
   int sum{0}; // sum of number1 and number2 (initialized to 0)

   std::cout << "Enter first integer: "; // prompt user for data
   std::cin >> number1; // read first integer from user into number1

   std::cout << "Enter second integer: "; // prompt user for data
   std::cin >> number2; // read second integer from user into number2

   sum = number1 + number2; // add the numbers; store result in sum

   std::cout << "Sum is " << sum << std::endl; // display sum; end line
} // end function main
```

list initialization

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

```cpp
int number1 = 0; // first integer to add (initialized to 0)
int number2 = 0; // second integer to add (initialized to 0)
int sum = 0; // sum of number1 and number2 (initialized to 0)
```

Arithmetic

Operation	Arithmetic operator	Algebraic expression	C++ expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	bm or $b \cdot m$	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Remainder
17 % 10 -> 7
171 % 10 -> 1
171 % 100 = 71
1775 % 1000 -> 775
3771 % 1000 -> ???

```
#include <cstdlib> // for std::rand and std::srand
#include <ctime> // for std::time

// Initialize random seed and generate random number
std::srand(std::time(nullptr));
numberToGuess = std::rand() % 101; // Random number between 0 and 100
```

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. For *nested* parentheses, such as in the expression `a * (b + c / (d + e))`, the expression in the *innermost* pair evaluates first. [*Caution:* If you have an expression such as `(a + b) * (c - d)` in which two sets of parentheses are not nested, but appear "on the same level," the C++ Standard does *not* specify the order in which these parenthesized subexpressions will evaluate.]
* / %	Multiplication Division Remainder	Evaluated second. If there are several, they're evaluated left to right.
+ -	Addition Subtraction	Evaluated last. If there are several, they're evaluated left to right.

Arithmetic

Algebra: $m = \dfrac{a + b + c + d + e}{5}$

C++: `m = (a + b + c + d + e) / 5;`

Algebra: $y = mx + b$

C++: `y = m * x + b;`

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

```
Step 1.  y = 2 * 5 * 5 + 3 * 5 + 7;     (Leftmost multiplication)
             2 * 5 is 10

Step 2.  y = 10 * 5 + 3 * 5 + 7;        (Leftmost multiplication)
             10 * 5 is 50

Step 3.  y = 50 + 3 * 5 + 7;            (Multiplication before addition)
                  3 * 5 is 15

Step 4.  y = 50 + 15 + 7;               (Leftmost addition)
             50 + 15 is 65

Step 5.  y = 65 + 7;                    (Last addition)
             65 + 7 is 72

Step 6.  y = 72                         (Low-precedence assignment—place 72 in y)
```

Decision Making

Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Decision Making

Operators	Associativity	Type
()	*[See caution in Fig. 2.10]*	grouping parentheses
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	stream insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
=	right to left	assignment

```cpp
// Fig. 2.13: fig02_13.cpp
// Comparing integers using if statements, relational operators
// and equality operators.
#include <iostream> // enables program to perform input and output

using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

// function main begins program execution
int main() {
   int number1{0}; // first integer to compare (initialized to 0)
   int number2{0}; // second integer to compare (initialized to 0)

   cout << "Enter two integers to compare: "; // prompt user for data
   cin >> number1 >> number2; // read two integers from user

   if (number1 == number2) {
      cout << number1 << " == " << number2 << endl;
   }

   if (number1 != number2) {
      cout << number1 << " != " << number2 << endl;
   }

   if (number1 < number2) {
      cout << number1 << " < " << number2 << endl;
   }

   if (number1 > number2) {
      cout << number1 << " > " << number2 << endl;
   }

   if (number1 <= number2) {
      cout << number1 << " <= " << number2 << endl;
   }

   if (number1 >= number2) {
      cout << number1 << " >= " << number2 << endl;
   }
} // end function main
```

Account Object

```
// Creating and manipulating an Account object.
#include <iostream>
#include <string>

using namespace std;

// Definition of the Account class
class Account {
private:
   string name;

public:
   Account() : name("") {} // Constructor with default name

   void setName(string accountName) {
      name = accountName;
   }

   string getName() const {
      return name;
   }
};

int main() {
   Account myAccount; // Create Account object myAccount

   // Show that the initial value of myAccount's name is the empty string
   cout << "Initial account name is: " << myAccount.getName();

   // Prompt for and read name
   cout << "\nPlease enter the account name: ";
   string theName;
   getline(cin, theName);
   myAccount.setName(theName); // Set the name in the myAccount object

   // Display the name stored in object myAccount
   cout << "Name in object myAccount is: " << myAccount.getName() << endl;
}
```

constructor : automatically called function when an object of that class is created: initialize the object's properties, setting up initial states or performing any setup steps necessary for the object to be used.

```
class MyClass {
private:
    int x;

public:
    // Constructor
    MyClass(int value) : x(value) {
        // Initialization and setup tasks here
    }
};

int main() {
    MyClass obj(10);  // Creates an object of MyClass, calling the constructor
    // The value 10 is passed to the constructor and used to initialize 'x'
}
```

getName promises not to modify any member variables of the Account object on which it's called and can be called on both const and non-const instances of the class

```
Initial account name is:
Please enter the account name: Jane Green
Name in object myAccount is: Jane Green
```

Account Object

```cpp
// Creating and manipulating an Account object.
#include <iostream>
#include <string>

using namespace std;

// Definition of the Account class
class Account {
private:
   string name;

public:
   Account() : name("") {} // Constructor with default name

   void setName(string accountName) {
      name = accountName;
   }

   string getName() const {
      return name;
   }
};

int main() {
   Account myAccount; // Create Account object myAccount

   // Show that the initial value of myAccount's name is the empty string
   cout << "Initial account name is: " << myAccount.getName();
  //if not using std namespace , must be std::cout

   // Prompt for and read name
   cout << "\nPlease enter the account name: ";
   string theName;
   getline(cin, theName);
   myAccount.setName(theName); // Set the name in the myAccount object

   // Display the name stored in object myAccount
   cout << "Name in object myAccount is: " << myAccount.getName() << endl;
}
```

Without ‘explicit’

```cpp
Account(std::string accountName) : name{accountName} { }

Account account1("John"); // Direct initialization
Account account1 = "John"; // Copy-initialization (implicit conversion)
```

With ‘explicit’

```cpp
explicit Account(std::string accountName) : name{accountName} { }

Account account1("John"); // Direct initialization is required.
Account account1 = "John"; // Error: cannot use implicit conversion.
```

lect03_01.cpp
lect03_02cpp

Account Object

Constructors and Implicit Conversion

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    // Constructor that initializes 'value' with 'x'
    MyClass(int x) : value(x) {

    }

    // A member function to display 'value'
    void displayValue() const {
        std::cout << "Value: " << value << std::endl;
    }
};

// Function that takes 'MyClass' object as parameter
void someFunction(MyClass obj) {
    std::cout << "Inside someFunction: ";
    obj.displayValue();
}

int main() {
    // This will implicitly convert 10 to MyClass and call someFunction
    someFunction(10);

    return 0;
}
```

Constructors and Explicit Conversion

```
class MyClass {
public:
    explicit MyClass(int x) { ... }
};

void someFunction(MyClass obj) { ... }

someFunction(10); // Error: no implicit conversion allowed.
someFunction(MyClass(10)); // Correct: explicit conversion.
```

- Use explicit constructors to prevent implicit conversions for classes where such behavior could be harmful or unclear.
- It's a good practice, especially for single-argument constructors, to avoid subtle bugs related to implicit conversions.
- Remember that explicit constructors can still be used for direct initialization and explicit conversions.

lect03_03.cpp

Account Object - Constructor

```
class Account {
private:
   std::string a_name;
   std::string b_name;

public:
   explicit Account(std::string aName, std::string bName) :
a_name{aName}, b_name{bName} { }

   // ... other member functions ...
};


Account myAccount("John", "Doe");


void someFunction(Account account) {
   // ...
}

someFunction(Account("first_name", "last_name"));
```

```
class Account {
private:
   std::string a_name;
   std::string b_name;

public:
   Account(std::string aName, std::string bName) : a_name{aName},
b_name{bName} { }

   // ... other member functions ...
};

Account myAccount("John", "Doe");


void someFunction(Account account) {
   // ...
}

someFunction({"first_name", "last_name"}); // This will work
because the constructor isn't explicit.
```

Difference between pointers and references

```
// With pointers
Account* actPtr = &myAccount;
cout << (*actPtr).getName();  // Need dereference operator *
// or
cout << actPtr->getName();    // Or arrow operator ->

// With references
Account& act = myAccount;
cout << act.getName();        // Use directly like a normal object
```

Once a reference is set up, use it exactly as if it were the original object. The compiler handles the "behind the scenes" work of accessing the referenced object.

Computer Organization

- Vector
- Organizing Programs
- Programming Style
- Organizing Data

Vector std::vector

A sequence container that encapsulates dynamic size arrays

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

int main() {
    // Declare a vector of integers
    std::vector<int> myVector {1,2,3};

    // Add elements to the vector
    myVector.push_back(10);
    myVector.push_back(20);
    myVector.push_back(30);

    // Iterate and print elements
    for (int i = 0; i < myVector.size(); ++i) {
        std::cout << myVector[i] << ' ';
    }
    std::cout << std::endl;

    // Range-based for loop
    for (int element : myVector) {
        std::cout << element << ' ';
    }
    std::cout << std::endl;

```

The error "expected ';' at end of declaration" in Visual Studio Code when using C++ on a Mac

- Go to the .vscode folder in your project directory.
- Open tasks.json.
- Find the args
- Add the compiler flag for the C++ version you want to use. Ex. -std=c++11.

```json
"type": "cppbuild",
"label": "C/C++: g++ build active file",
"command": "/usr/bin/g++",
"args": [
    "-std=c++11",
    "-fdiagnostics-color=always",
    "-g",
    "${file}",
    "-o",
    "${fileDirname}/${fileBasenameNoExtension}"
],
"options": {
    "cwd": "${fileDirname}"
},
"problemMatcher": [
```

```
phairojjatanachai@Phai
1 2 3 10 20 30
1 2 3 10 20 30
```

Methods of std::vector

- **push_back(const T& value):** Adds a new element to the end of the vector, resizing it if necessary. This element is a copy of value.
- **pop_back():** Removes the last element in the vector, effectively reducing its size by one. This does not return the removed element.
- **size() const:** Returns the number of elements in the vector. This is the number of actual objects held in the vector, which is not necessarily equal to its storage capacity.
- **empty() const:** Checks if the vector has no elements and returns true if the vector size is 0, false otherwise.

const T& value:
const: parameter will not be modified by the function.
T: This represents the data type of the parameter being passed to the function. T is a placeholder and could be any type, like int, double, std::string, etc.
&: The ampersand indicates that the parameter is passed by reference. This means that instead of passing a copy of the variable, the function will receive a reference to the original variable, avoiding the overhead of copying and allowing the function to access the actual variable.

Methods of std::vector

- **clear():** Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.
- **at(size_t n)**: Returns a reference to the element at specified location **n**, with bounds checking. If **n** is not within the range of the vector, an exception of type **std::out_of_range** is thrown.
- **front():** Returns a reference to the first element in the vector. Using this on an empty vector causes undefined behavior.
- **back():** Returns a reference to the last element in the vector. Using this on an empty vector causes undefined behavior.
- **reserve(size_t n):** Requests that the vector capacity be at least enough to contain n elements. This is a non-binding request to optimize memory allocations if you know the vector will grow to a certain size.

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

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

    // Access elements
    std::cout << "First element: " << vec.front() << std::endl;
    std::cout << "Last element: " << vec.back() << std::endl;

    // Size and capacity
    std::cout << "Size: " << vec.size() << std::endl;

    // Check if the vector is empty
    if (!vec.empty()) {
        std::cout << "Vector is not empty" << std::endl;
    }

    // Remove the last element
    vec.pop_back();

    // Iterate over the vector
    for (int i = 0; i < vec.size(); ++i) {
        std::cout << "Element at index " << i << ": " << vec[i] << std::endl;
    }

    // at
    std::cout << "Element at index 2: " << vec.at(2) << std::endl;

    // Reserve space for 10 elements
    vec.reserve(10);

    // Clear the vector
    vec.clear();
    std::cout << "Vector cleared. Size: " << vec.size() << std::endl;

    return 0;
}
```

```
First element: 10
Last element: 50
Size: 5
Vector is not empty
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 2: 30
Vector cleared. Size: 0
```


IF YOU WANT TO CRY, I'LL GIVE YOU SOMETHING TO CRY ABOUT!
THE C PROGRAMMING LANGUAGE
SketchbookSilliness.com

List std::list

A container that supports constant time insertion and deletion of elements from anywhere in the container. It is implemented as a doubly-linked list, which means each element keeps a link to both the previous and the next element in the list

```cpp
#include <iostream>
#include <list>

// Function that prints all elements in the list
void printList(const std::list<int>& lst) {
    for (int element : lst) {
        std::cout << element << " ";
    }
    std::cout << "\n";
}

int main() {
    // Creating a list of integers
    std::list<int> myList = {1, 2, 3, 4, 5};

    std::cout << "Original list: ";
    printList(myList);

    // Adding elements to the list
    myList.push_front(0); // Add at the beginning
    myList.push_back(6);  // Add at the end

    std::cout << "List after adding elements: ";
    printList(myList);

    // Removing elements from the list
    myList.pop_front();  // Remove from the beginning
    myList.pop_back();   // Remove from the end

    std::cout << "List after removing elements: ";
    printList(myList);

    return 0;
}
```

```
Original list: 1 2 3 4 5
List after adding elements: 0 1 2 3 4 5 6
List after removing elements: 1 2 3 4 5
```

List std::list

- push_back: Adds an element to the end of the list.
- push_front: Inserts an element at the beginning of the list.
- pop_back: Removes the last element of the list.
- pop_front: Removes the first element of the list.
- size: Returns the number of elements in the list.
- sort: Sorts the elements of the list.
- clear removes all elements from the list.
- reverse reverses the order of the elements in the list.
- remove_if remove elements from the list based on a specific condition

List std::list

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> myList = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::cout << "Original list: ";
    for (int num : myList) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    // Using remove_if to remove even numbers
    myList.remove_if([](int n) { return n % 2 == 0; });

    std::cout << "List after removing even numbers: ";
    for (int num : myList) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> myList = {0, 1, 2, 0, 3, 0, 4, 5};

    std::cout << "Original list: ";
    for (int num : myList) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    // Using remove_if to remove elements equal to zero
    myList.remove_if([](int n) { return n == 0; });

    std::cout << "List after removing zeros: ";
    for (int num : myList) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

List std::list

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<float> myList = {1.5, -2.3, 3.7, -4.1, 5.2, -6.8};

    std::cout << "Original list: ";
    for (float num : myList) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    // Using remove_if to remove negative numbers
    myList.remove_if([](float n) { return n < 0; });

    std::cout << "List after removing negative numbers: ";
    for (float num : myList) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```cpp
#include <iostream>
#include <list>
#include <string>

int main() {
    std::list<std::string> myList = {"hello", "remove", "world", "remove", "example"};

    std::cout << "Original list: ";
    for (const auto& str : myList) {
        std::cout << str << " ";
    }
    std::cout << "\n";

    // Using remove_if to remove strings that are "remove"
    myList.remove_if([](const std::string& s) { return s == "remove"; });

    std::cout << "List after removing 'remove': ";
    for (const auto& str : myList) {
        std::cout << str << " ";
    }
    std::cout << "\n";

    return 0;
}
```

Vector (std::vector)

- Dynamic array that grows automatically
- Fast random access (constant time) using index: vec[i]
- Fast insertion/deletion at the end
- Slow insertion/deletion in the middle (needs to shift elements)
- Contiguous memory storage
- Good memory locality for iteration

```
vector<int> vec = {1, 2, 3};
vec.push_back(4);  // Fast
vec[0];            // Fast random access
```

List (std::list)

- Doubly-linked list
- No random access - must traverse from beginning/end
- Fast insertion/deletion anywhere once position is found
- Elements can be scattered in memory
- More memory overhead per element (needs to store prev/next pointers)

```
list<int> lst = {1, 2, 3};
lst.push_back(4);     // Fast
lst.push_front(0);    // Fast
// No lst[0] - must iterate
```

Array (std::array):

- Fixed-size array (size set at compile time)
- Fast random access
- Cannot grow or shrink
- Smallest memory overhead
- Contiguous memory storage

```
array<int, 3> arr = {1, 2, 3};
arr[0];             // Fast random access
// Can't add or remove elements
```

When to use each

- Vector: Default choice for most cases - good balance of features
- List: When you need lots of insertions/deletions in the middle
- Array: When you know the exact size needed at compile time

Organizing Programs

- C++ offers two fundamental ways of organizing programs
  - Functions (subroutines)
  - Data structures
- We will explore a class which is a way to combine functions and data structures into a single

Writing C++ Functions (1)

A function must be declared in every source file that uses it, and defined only once.

```
ret-type function-name(parm-decls);              // function declaration

[inline] ret-type function-name(parm-decls)      // function definition
{
    // function body goes here
}
```

Example:

```
// compute a student's overall grade
// from midterm and final exam grades and homework grade
double grade(double midterm, double final, double homework)
{
    return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}
```

Writing C++ Functions (2)

Previously, we computed a grade by writing:

```
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     << setprecision(prec) << endl;
```

With `grade` function, we could have written:

```
cout << "Your final grade is " << setprecision(3)
     << grade(midterm, final, sum / count)
     << setprecision(prec) << endl;
```

Example: Finding Medians

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // For std::sort
#include <stdexcept> // For std::domain_error

// Function to compute the median of a vector<double>
double median(std::vector<double> vec) {
    if (vec.empty())
        throw std::domain_error("median of an empty vector");

    std::sort(vec.begin(), vec.end());

    std::vector<double>::size_type mid = vec.size() / 2;

    return vec.size() % 2 == 0 ? (vec[mid] + vec[mid - 1]) / 2 : vec[mid];
}

int main() {
    try {
        std::vector<double> vec {1.5, 3.2, 6.0, 9.1, 4.6, 2.8};
        std::cout << "The median is " << median(vec) << std::endl;
    } catch (std::domain_error& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

The medium is 3.9

Function Overload

Function overloading in C++ is a feature that allows you to have more than one function with the same name but with different parameters (number, type, or both).

```cpp
#include <iostream>

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Overloaded function to add two doubles
double add(double a, double b) {
    return a + b;
}

// Overloaded function to add three integers
int add(int a, int b, int c) {
    return a + b + c;
}

int main() {
    std::cout << "Adding two integers: " << add(1, 2) << std::endl;      // Calls int add(int, int)
    std::cout << "Adding two doubles: " << add(1.5, 2.3) << std::endl;  // Calls double add(double, double)
    std::cout << "Adding three integers: " << add(1, 2, 3) << std::endl;// Calls int add(int, int, int)

    return 0;
}
```

```
Adding two integers: 3
Adding two doubles: 3.8
Adding three integers: 6
```

Programming Style

Background

- Like writing, programming is a form of **communication**
- Code is read much more often than written, so the code must be **understandable**
- Though subjective, **guidelines** or **conventions** are often useful
- No one true style: one size doesn't fit all
- Choose one style and **be consistent**

Code Convention

- **Improves Readability:** Makes it easier for others (and yourself) to read and understand the code.
- **Facilitates Collaboration:** Ensures consistency across a codebase, which is crucial when multiple people are working on the same project.
- **Enhances Maintainability:** Consistent code is easier to maintain and update.
- **Reduces the Chance of Errors:** Certain conventions, especially those related to programming practices, can help prevent common coding errors.

Code Convention

Code convention, often referred to as coding standards or coding style guidelines, is a set of guidelines and best practices for writing code.

- **Naming Conventions:** Guidelines for naming variables, functions, classes, and other entities. For example, using camelCase for variables and PascalCase for class names.
- **Formatting and Indentation:** Rules about how to format code, including the use of tabs vs. spaces for indentation, the placement of braces, line length limits, etc.
- **Commenting and Documentation:** Standards for writing comments and documentation to explain complex parts of the code, the purpose of functions, classes, modules, etc.
- **Programming Practices:** Best practices regarding programming patterns, error handling, avoiding the use of global variables, etc.

Code Convention

- **File and Folder Structure:** Guidelines on how to organize code files and directories.
- **Version Control:** Standards for using version control systems, including commit messages, branching strategies, etc.
- **Language-Specific Conventions:** Certain conventions might be specific to a programming language. For example, Python has PEP 8, which is a set of guidelines for writing Pythonic code.
- **Testing Conventions:** Guidelines for writing and organizing tests.

Naming Convention

- Use meaningful names
  - Noun for variables, verb for functions
  - Simple names (i, x, y, p, etc.) are OK in small scopes
- Don’t use acronyms
- Don’t use excessively long names
- Beware of confusing letters and digits: **0Oo1lL**

Multiple-word Identifiers

- **`isupper`**: flat case
- **`ISUPPER`**: upper flat case
- **`isUpper`**: camel case
- **`IsUpper`**: pascal case, upper camel case
- **`is_upper`**: snake case
- **`IS_UPPER`**: macro case, constant case

Other variants: `is_Upper`, `Is_Upper`, `is-upper`, `IS-UPPER`, `Is-Upper`

Naming Convention: Example

C and C++

- Variables: **`some_var`**
- Functions: **`do_something(…)`**
- Types: **`Student_info`**
- Constants and macros: **`NUM_ITEMS`**

Naming Convention: Example (2)

Java, C#, Javascript, etc.

- Variables: **`someVar`**
- Functions: **`doSomething(…)`**
- Types: **`StudentInfo`**
- Constants and macros: **`NUM_ITEMS`**

Language-specific Name

In C and C++:

- Names are case sensitive
- Keywords are lowercase
- Names from standard library are mostly lowercase
- Reserved names
  - **`_Reserved`** (begin with an underscore and a capital letter)
  - **`__reserved`** (containing double underscore)

Indentation

```
// if statement
if (a == b) {
    // ...
}
else {
    // ...
}

// loop
for (int i = 0; i < 10; ++i) {
    // ...
}
```

```
// switch statement
switch (a) {
case A:
    // ...
    break;
case B:
    // ...
    break;
default:
    // ...
}
```

Indentation

```
/// function
double sqrt(double d)
{
    // ...
}
```

```
/// class or struct:
class Temperature_reading {
public:
    // ...
private:
    // ...
};
```

Whitespace

- Vertical whitespaces (empty lines)
  - Between functions, structs, etc.
  - Separate different sections of code
- Tabs vs spaces
  - Be consistent with indentation
  - Pick one style and stick with it throughout the project

Comments

Comments are good for:

1. Stating **intent** (what the code is supposed to do)
2. Explaining **ideas** related to the code
3. Stating **invariants**, pre- and post-conditions

Things to consider:

- Comments are **not for translating** program statements
- If the code is hard to read, **consider rewriting** it

Documentation

- Requirements
- Developer's Manual
  - Program Design/Model
  - Implementation Details
  - Programming Interface
- User Manual

Organizing Data

Student's Data

- Use **`struct`** to define a data structure that group related data together.
- We can define a data structure for student's data as follows:

```
struct Student_info {
    string name;
    double midterm, final;
    vector<double> homework;
}; // note the semicolon
   // -- it's required
```

Alternatively:

```
struct Student_info {
    string name;
    double midterm;
    double final;
    vector<double> homework;
};
```

```cpp
#include <iostream>
#include <string>
#include <vector>

// Define the Student_info struct
struct Student_info {
    std::string name;
    double midterm;
    double final;
    std::vector<double> homework;
};

// Function to print student information
void print_student_info(const Student_info& s) {
    std::cout << "Name: " << s.name << std::endl;
    std::cout << "Midterm: " << s.midterm << ", Final: " << s.final << std::endl;
    std::cout << "Homework Grades: ";
    for (double grade : s.homework) {
        std::cout << grade << " ";
    }
    std::cout << std::endl << std::endl;
}

```

```cpp
int main() {
    // Create an instance of Student_info and set its member values
    Student_info student1;
    student1.name = "John Doe";
    student1.midterm = 88.5;
    student1.final = 92.0;
    student1.homework = {95.0, 87.0, 90.0};

    // Create another student instance
    Student_info student2 = {"Jane Smith", 90.0, 91.5, {88.0, 92.0, 85.0}};

    // Create a vector to hold multiple students
    std::vector<Student_info> students;
    students.push_back(student1);
    students.push_back(student2);

    // Iterate over the vector to print each student's info
    for (const auto& student : students) {
        print_student_info(student);
    }

    return 0;
}
```

```
Name: John Doe
Midterm: 88.5, Final: 92
Homework Grades: 95 87 90

Name: Jane Smith
Midterm: 90, Final: 91.5
Homework Grades: 88 92 85
```

`#ifndef` Guard Pattern

In every header file, we usually use **`#ifndef`** pattern to guard against multiple inclusions of the header contents into the same source code:

```
#ifndef SOME_UNIQUE_NAME
#define SOME_UNIQUE_NAME

// ...

#endif /* SOME_UNIQUE_NAME */
```

We must ensure that **SOME_UNIQUE_NAME** is **really unique** throughout the entire application project.

Q & A