Introduction to C++

- DAL bert & +ition

TOC

Introduction + Objective

Essential Elements of C++ (Basic)

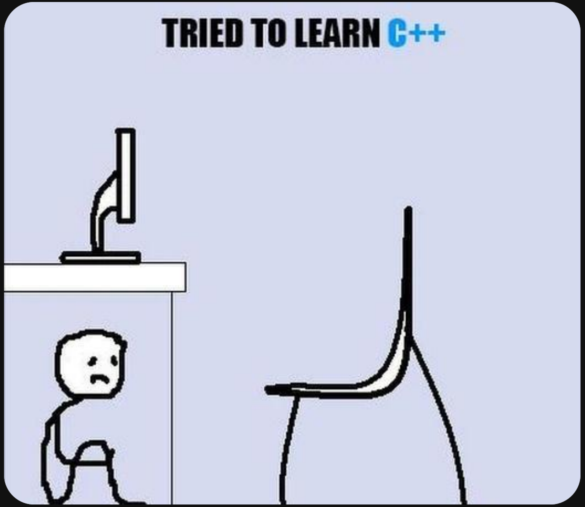
- ○ Data Types and Variables
- ○ Operators and Expressions
- ○ Control Flow Structures

break;

Object-Oriented Programming in C++ (Intermediate)

- ○ Classes and Objects
- ○ Inheritance and Polymorphism
- ○ Data Encapsulation

If (needed){ break; }



TOC

Advanced Topics in C++ (Advance)

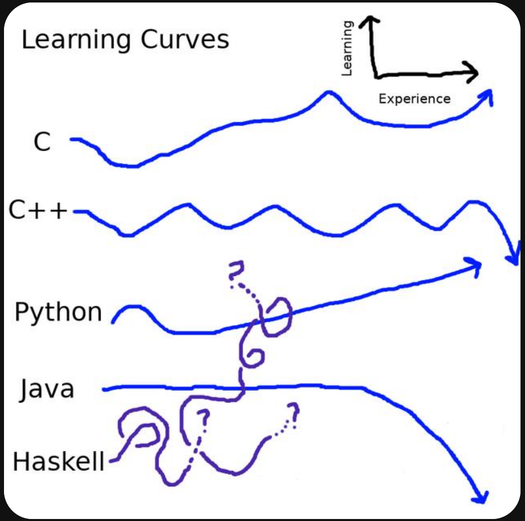
- ○ Pointers
- ○ Exception Handling and Error Management

Best Practices in C++ (Summarized)

- ○ Memory Management and Resource Management
- ○ Debugging and Testing Techniques

Wrap-up 👋

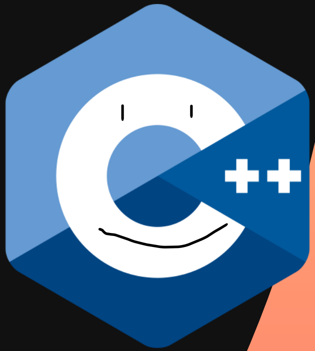
- ○ Summary of Key Concepts and Topics



How to C=C+1

What is C++ ?_?

- **General purpose** programming language
- Extension of C, supports **Object-Oriented Programming (OOP)**
- Created by Danish Computer Scientist Bjarne Stroustrup
- Popular due to **speed & efficiency**
- **Harder to shoot your foot**

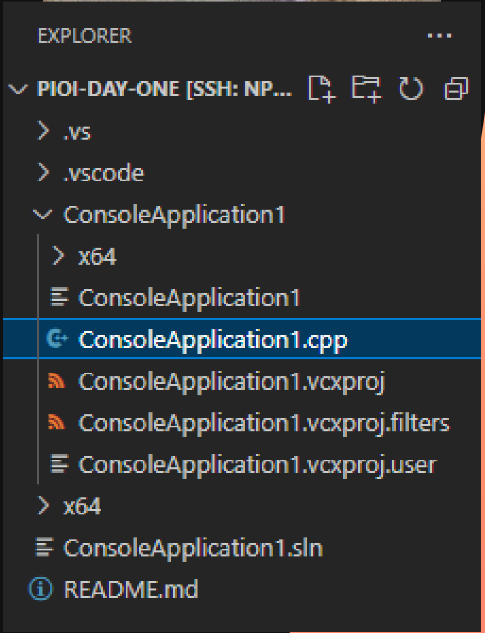


C++ Uses

- Operating Systems
- Video Games
- Database Management Systems (DBMS)
- Web Browsers
- Embedded Systems

C++ Compiler

1. Install **VSC!** https://code.visualstudio.com/download
2. Visit https://github.com/np-overflow/PIOIDay1
3. Copy temporary Password, Paste
4. Wait while extensions install bottom right
5. Navigate to ConsoleApplication1.cpp and press F5
6. Choose first options for prompts
7. Output under “**Terminal**”

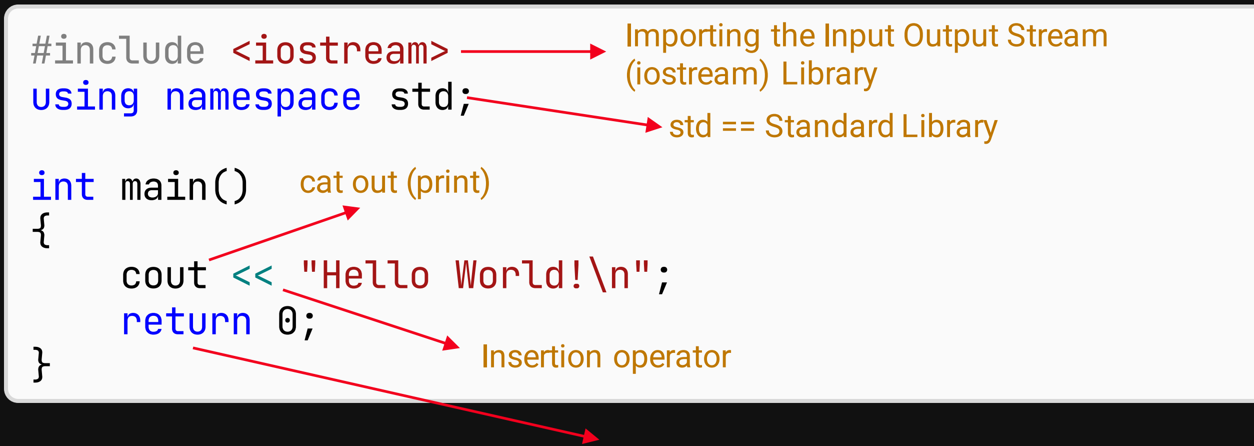


- .vs
- .vscode
- ConsoleApplication1
  - x64
  - ConsoleApplication1
  - ConsoleApplication1.cpp
  - ConsoleApplication1.vcxproj
  - ConsoleApplication1.vcxproj.filters
  - ConsoleApplication1.vcxproj.user
- x64
- ConsoleApplication1.sln
- README.md

Essential Elements of C++

On your screen right now…

`main.cpp`



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

int main()
{
    cout << "Hello World!\n";
    return 0;
}
```

Importing the Input Output Stream (iostream) Library

std == Standard Library

cat out (print)

Insertion operator

Ends the main function

0 = No errors met
1/-1 = Usually represents errors met

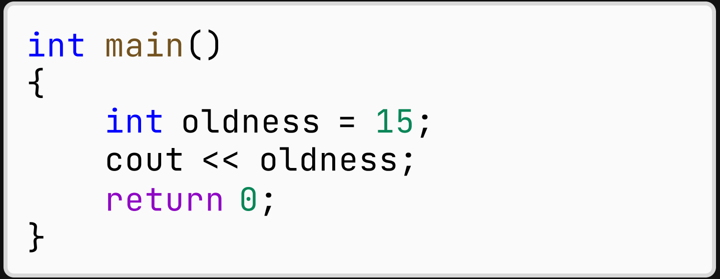
Data Types & Variables

- Standard Data Types

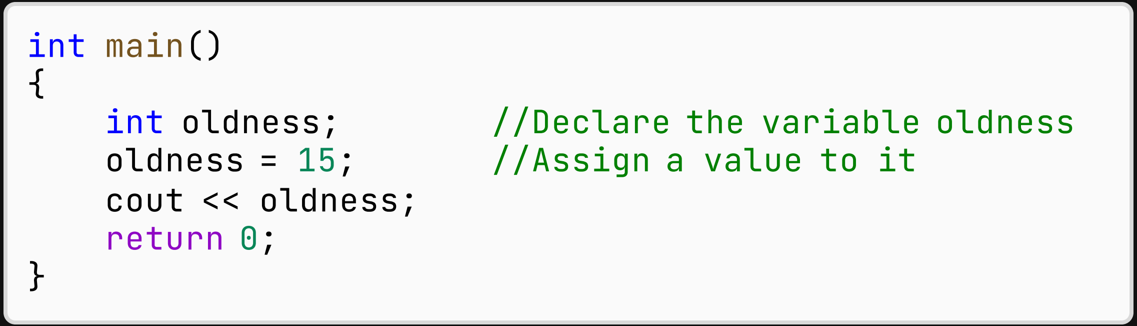
Data Type	Description	E.g.
bool	true/false	true
char	Single char. or ASCII Values	A, 65
int	Whole numbers, **NO decimals**	100
float	Up to **6-7** decimal digits	100.111111
double	Up to **15** decimal digits	100.123456789
string	Sequence of characters, req. **#include<string>**	“I love PIOI ✨”

Data Types & Variables

- Declaration and Assignment of Variables



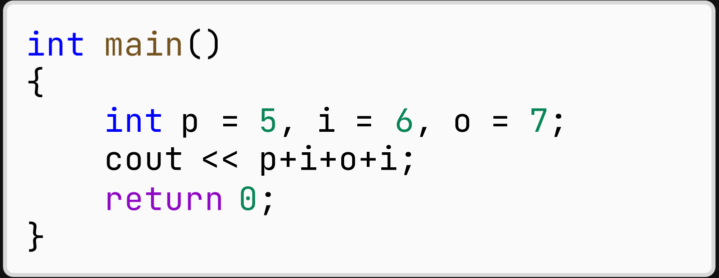
```
int main()
{
    int oldness = 15;
    cout << oldness;
    return 0;
}
```



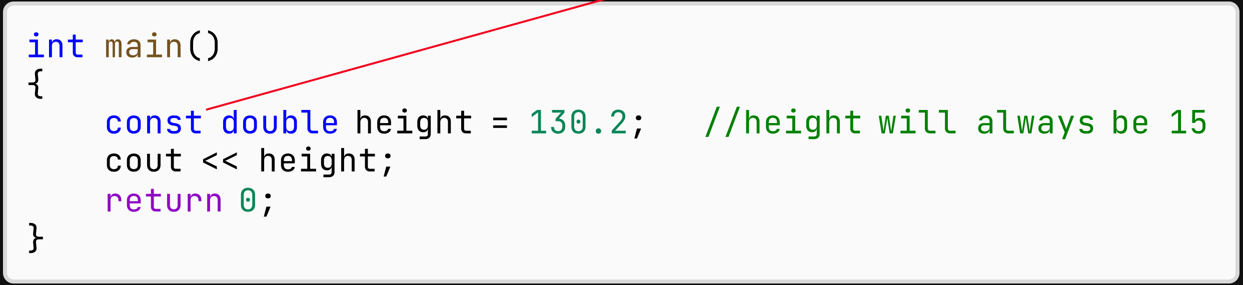
```
int main()
{
    int oldness;            //Declare the variable oldness
    oldness = 15;           //Assign a value to it
    cout << oldness;
    return 0;
}
```

Declare Multiple Variables

- Tips and Tricks



```
int main()
{
    int p = 5, i = 6, o = 7;
    cout << p+i+o+i;
    return 0;
}
```



```
int main()
{
    const double height = 130.2;   //height will always be 15
    cout << height;
    return 0;
}
```

Note: assigning height = 129
(any new value)
will lead to errors 💀

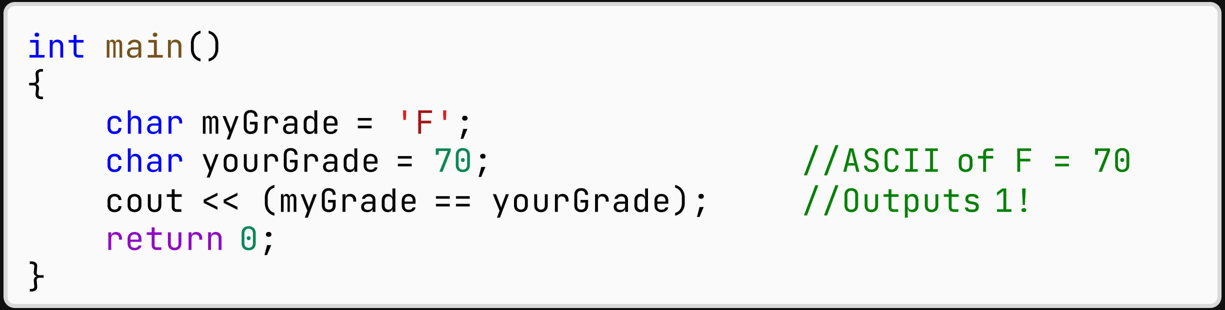
Boolean Variables

- true == 1, false == 0 (Note: **Case-sensitive**, True != true)

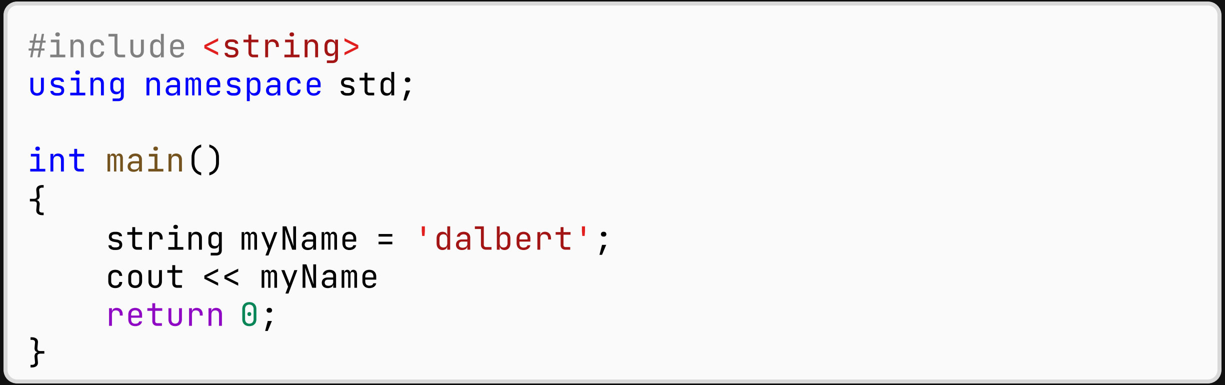
```
int main()
{
    bool excited = true;
    bool tired = false;
    cout << excited;        // Outputs 1
    cout << tired;          // Outputs 0
    return 0;
}
```

Strings and Characters

- Cool Interactions



```
int main()
{
    char myGrade = 'F';
    char yourGrade = 70;                 //ASCII of F = 70
    cout << (myGrade == yourGrade);      //Outputs 1!
    return 0;
}
```



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

int main()
{
    string myName = 'dalbert';
    cout << myName
    return 0;
}
```

User Input

- cin → cat in
- Used with extraction operator, **>>**

```
int main()
{
    char x;
    cout << "Type your grade: ";          // Prints text
    cin >> x;                             // Gets user input
    cout << "Your grade is: " << x;       // Display the user input value
}
```

Operators & Expressions

- Standard Math Operators

Operator	Description	E.g.
+	Add	x + 3
-	Subtract	x – y
*	Multiply	x * 5
/	Divide	x / 3
%	Modulus, returns remainder	x % y
++	Increment, + by 1	++x
--	Decrement, - by 1	--x

Operators & Expressions

- Math Assignment Operators

Operator	Description	E.g.
+=	Add	x += 3 is x = x + 3
-=	Subtract	x -= 3
*=	Multiply	x *= 5
/=	Divide	x /= 3
%=	Modulus, returns remainder	x %= y

Operators & Expressions

- Bitwise Assignment Operators; int x = 10 (binary = 1010)

Operator	Description	E.g.
&=	Bitwise **AND**	x &= 3
|=	**OR**	x |= 3
^=	**XOR**	x ^= 1
>>=	Bit shift **RIGHT**	x >>= 2 → 2 (binary 10)
<<=	Bit shift **LEFT**	x <<= 2 → 40 (binary 101000)

Operators & Expressions

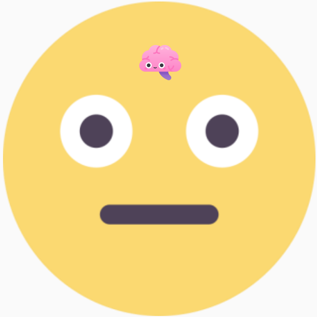
- Comparison and Logical Operators; int x = 5; int y = 10

Operator	Description	E.g.
==	Equal to	x == y // 0
!=	Not Equal to	x != y // 1
>	More than	x > y // 0
<	Less than	x < y // 1
>=	More than or Equals to	x >= y // 0
<=	Less than or Equals to	x <= y // 1
&&	Logical AND	(x != y && x <= y) // 1
||	Logical OR	(x == y || x > y) // 0
!	Logical NOT	!(x == y) // 1

Control Flow Structures

- If, else, else if, remember **brackets**

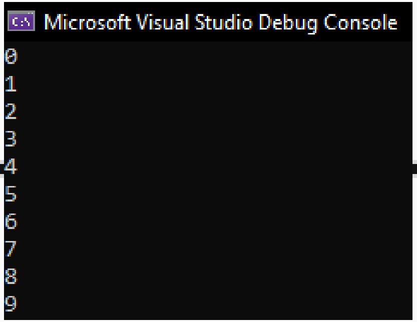
```
int main()
{
    int myBrainCells = 19;
    if (myBrainCells < 10) {
        cout << "You need more.";
    }
    else if (myBrainCells < 20) {
        cout << "Average.";
    }
    else {
        cout << "Giga Brain.";
    }
    return 0;
}
```



Control Flow Structures

- While Loop

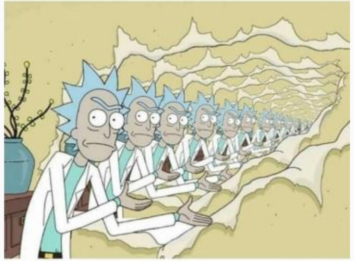
```
int main()
{
    int x = 0;
    while (x < 10)
    {
        cout << x << "\n";
        x++;
    }
}
```



Microsoft Visual Studio Debug Console

```
0
1
2
3
4
5
6
7
8
9
```

When you forget to break out of the while loop

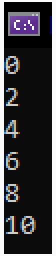


```
int main()
{
    int x = 0;
    do {
        cout << x << "\n";      //Ran before checking if x < 10
        x++;
    } while (x < 10);
}
```

Control Flow Structures

- For Loop (Variable; Condition; Increment)

```
int main()
{
    for (int i = 0; i <= 10; i += 2) {
        cout << i << "\n";
    }
}
```

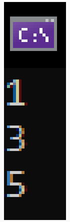


```
0
2
4
6
8
10
```

Control Flow Structures

- Continue → Goes to the next iteration
- Break → Ends the loop

```
int main()
{
    for (int i = 0; i < 10; i++) {
        if (i % 2 == 0) {       //Skip even numbers
            continue;
        }
        else if (i == 7) {      //End when 7
            break;
        }
        cout << i << "\n";
    }
}
```

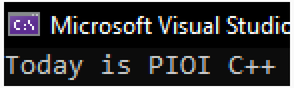


```
1
3
5
```

Control Flow Structures

- Switch

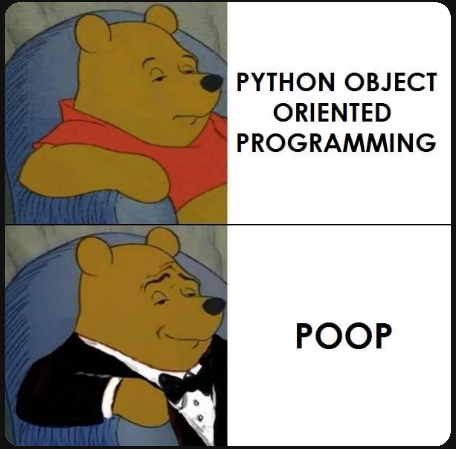
```
int main()
{
    int day = 1;
    switch (day) {
    case 1:
        cout << "Today is PIOI C++\n";
        break;
    case 2:
        cout << "Today is PIOI Data Structures\n";
        break;
    default:
        cout << "I won PIOI :D\n";
    }
}
```



```
Today is PIOI C++
```

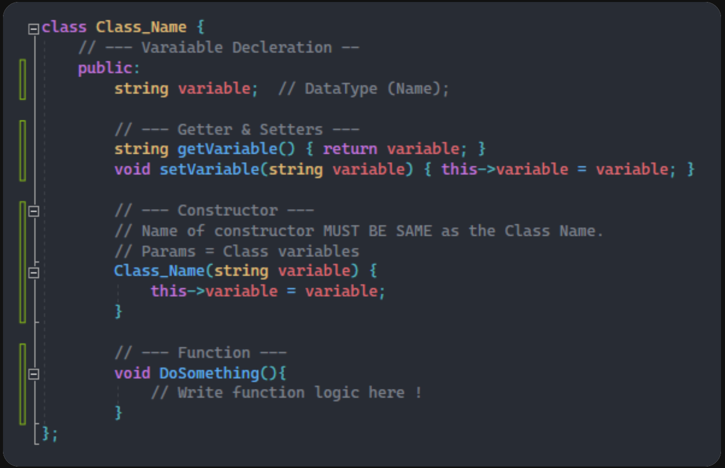
Note: if switch doesn't fit in any cases, **default** is ran 🎉

Object Oriented Programming in C++



Classes

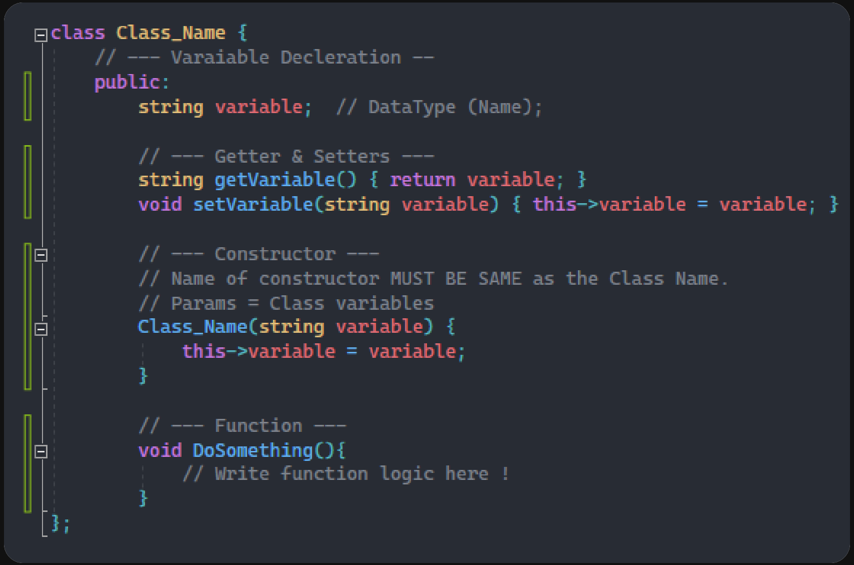
- A **blueprint** or **template** used to create an ***object***.
- Contains **attributes (data) and behaviors (functions)** of an ***object***.
- Typically contains **Getter/Setters** & **Constructors**



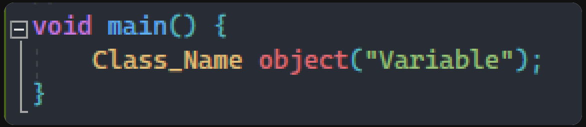
```
class Class_Name {
    // --- Varaiable Decleration --
    public:
        string variable;  // DataType (Name);

        // --- Getter & Setters ---
        string getVariable() { return variable; }
        void setVariable(string variable) { this->variable = variable; }

        // --- Constructor ---
        // Name of constructor MUST BE SAME as the Class Name.
        // Params = Class variables
        Class_Name(string variable) {
            this->variable = variable;
        }

        // --- Function ---
        void DoSomething(){
            // Write function logic here !
        }
};
```

Objects

- An **INSTANCE** of a class.
- Has it’s own **states** & can use the **methods** defined by the class.



```
class Class_Name {
    // --- Varaiable Decleration --
    public:
        string variable;  // DataType (Name);

        // --- Getter & Setters ---
        string getVariable() { return variable; }
        void setVariable(string variable) { this->variable = variable; }

        // --- Constructor ---
        // Name of constructor MUST BE SAME as the Class Name.
        // Params = Class variables
        Class_Name(string variable) {
            this->variable = variable;
        }

        // --- Function ---
        void DoSomething(){
            // Write function logic here !
        }
};
```



```
void main() {
    Class_Name object("Variable");
}
```

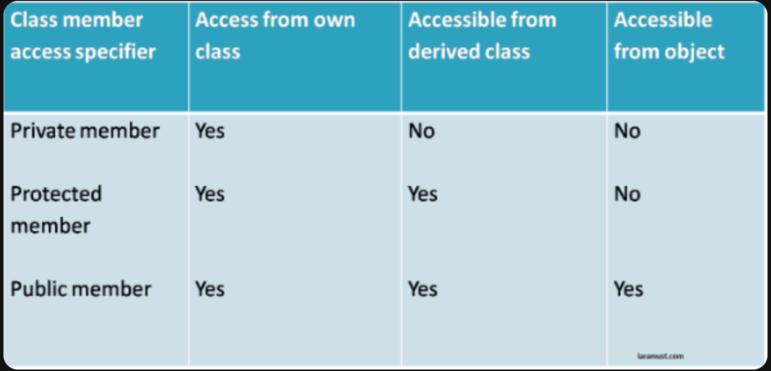
Data Encapsulation

- Practice of **hiding** the internal details of an object's data and behaviour from the outside world, and exposing only a public interface for interacting with the object.
- **TL;DR** : **Limit** ways which an obj data can be Modified & Accessed
- **Why?**
  - Improve Security
  - Improve Reliability
  - And other stuff (Maintenance, Reusability, ..)

Data Encapsulation

Access Specifiers

- **Private** = Only accessible by the class
- **Protected** = Only accessible by the class and its children
- **Public** = Accessible by everyone (everywhere)

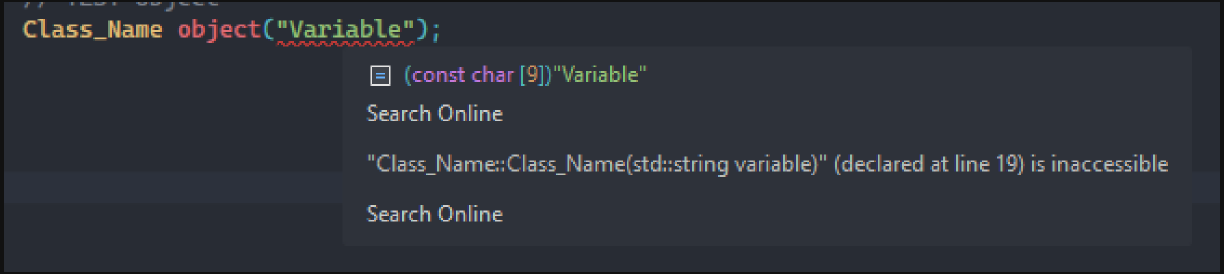


Class member access specifier	Access from own class	Accessible from derived class	Accessible from object
Private member	Yes	No	No
Protected member	Yes	Yes	No
Public member	Yes	Yes	Yes

laramust.com

Data Encapsulation

- **REMEMBER**: Specify Access Specification for your classes!



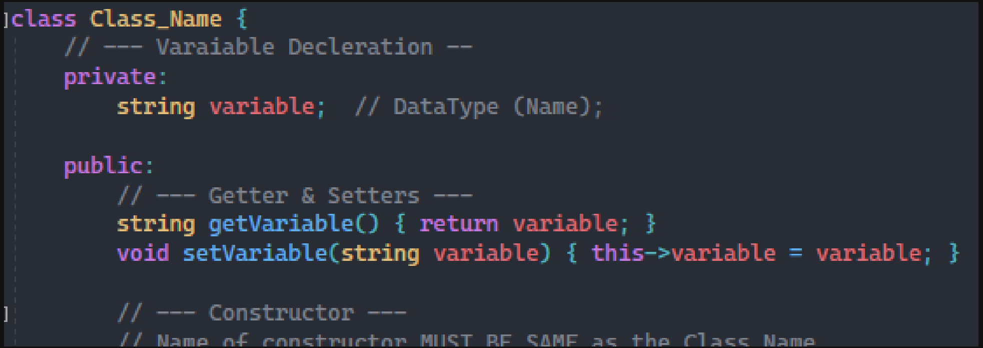
```
// TEST: Object
Class_Name object("Variable");
```

(const char [9])"Variable"

Search Online

"Class_Name::Class_Name(std::string variable)" (declared at line 19) is inaccessible

Search Online



```
class Class_Name {
    // --- Varaiable Decleration --
    private:
        string variable;  // DataType (Name);

    public:
        // --- Getter & Setters ---
        string getVariable() { return variable; }
        void setVariable(string variable) { this->variable = variable; }

        // --- Constructor ---
        // Name of constructor MUST BE SAME as the Class Name
```

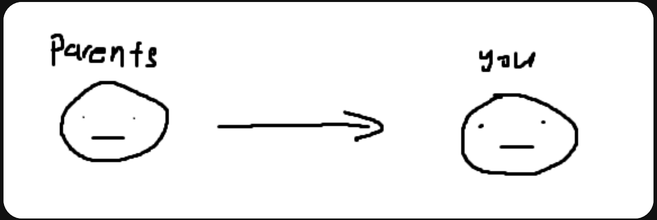
Inheritance

What :

- Allows you to “**Inherit**” properties / methods from another (Parent /base / superclass) class.
- Class that inherits from another class = "**derived**“ / "**subclass**“ / “**child**”

Why :

- Reuse code that has already been written in the base class.



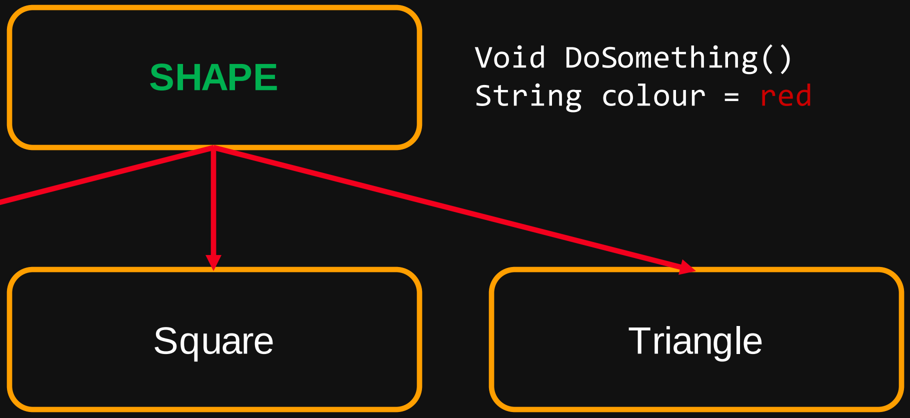
Polymorphism

- Enables **flexible** and **modular** code
- Allows you to convert a “generic” object to a more “**specific**” object

Types of polymorphism:

- **Compile-time** (aka Function overloading) –
  - Function with same name, but different functions / execution stuff
- **Run-time** (aka dynamic polymorphism) **–**
  - Allows function to be **overridden** by it’s child class
  - Functions can **behave differently** based on the TYPE of object/class

Inheritance & Polymorphism



```
Void DoSomething()
String colour = red
```

```
Void DoSomething(cout<<“1”;)
String colour = red
```

```
Void DoSomething()
String colour = red
```

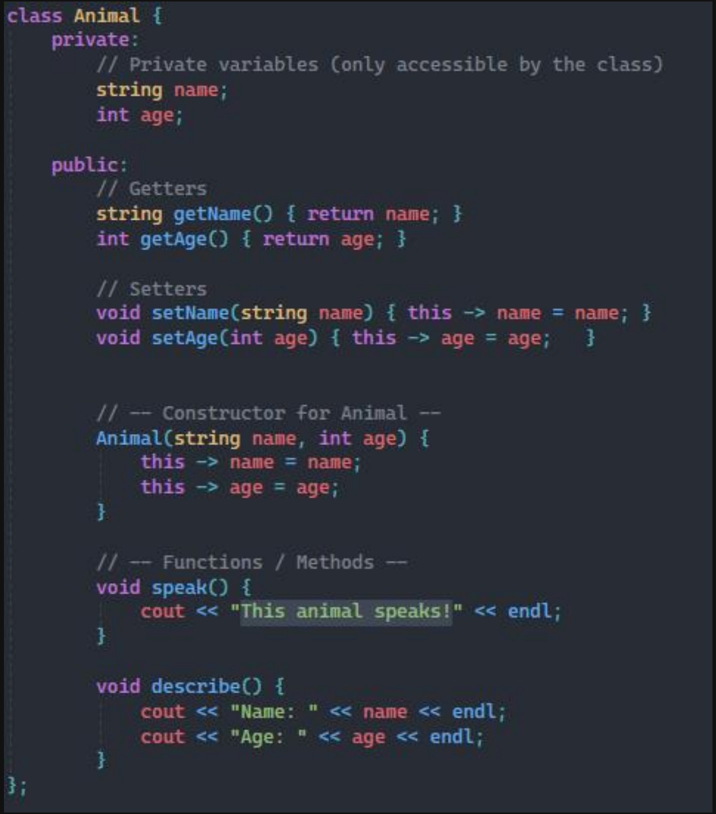
```
Void DoSomething()
String colour = red
```

Let’s Try it out!

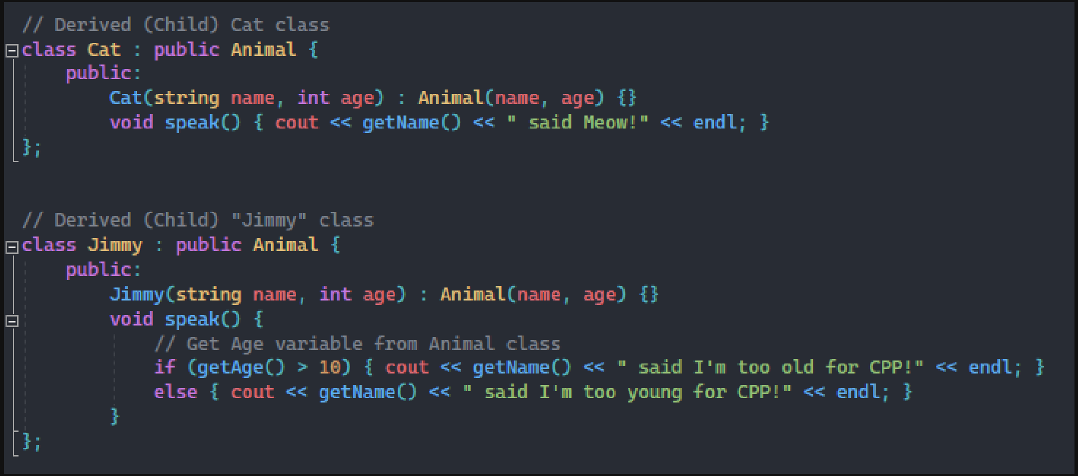
1. **Create a parent (Animal) class that has the following attributes :**
   - Name
   - Age
   - (INCLUDE GETTERS & SETTERS)
2. **Create a FUNCTION in the parent class called “Speak” so that it says :**
   - “This animal speaks!”
3. **Create a child (Cat) class that inherits from the parent class & overwrite the “Speak” function such that it would say “Meow” when called.**
4. **(BONUS) –** *May be a bit tricky!*

   Create a “Jimmy” class that inherits from the parent class, and make it say different things based on the AGE of the object!

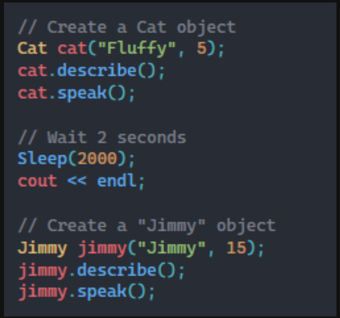
Solution



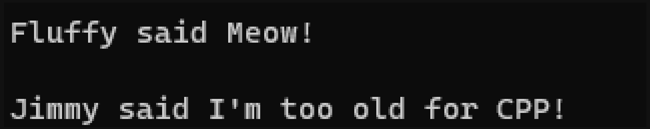
```
class Animal {
    private:
        // Private variables (only accessible by the class)
        string name;
        int age;

    public:
        // Getters
        string getName() { return name; }
        int getAge() { return age; }

        // Setters
        void setName(string name) { this -> name = name; }
        void setAge(int age) { this -> age = age;    }


        // -- Constructor for Animal --
        Animal(string name, int age) {
            this -> name = name;
            this -> age = age;
        }

        // -- Functions / Methods --
        void speak() {
            cout << "This animal speaks!" << endl;
        }

        void describe() {
            cout << "Name: " << name << endl;
            cout << "Age: " << age << endl;
        }
};
```



```
// Derived (Child) Cat class
class Cat : public Animal {
    public:
        Cat(string name, int age) : Animal(name, age) {}
        void speak() { cout << getName() << " said Meow!" << endl; }
};


// Derived (Child) "Jimmy" class
class Jimmy : public Animal {
    public:
        Jimmy(string name, int age) : Animal(name, age) {}
        void speak() {
            // Get Age variable from Animal class
            if (getAge() > 10) { cout << getName() << " said I'm too old for CPP!" << endl; }
            else { cout << getName() << " said I'm too young for CPP!" << endl; }
        }
};
```



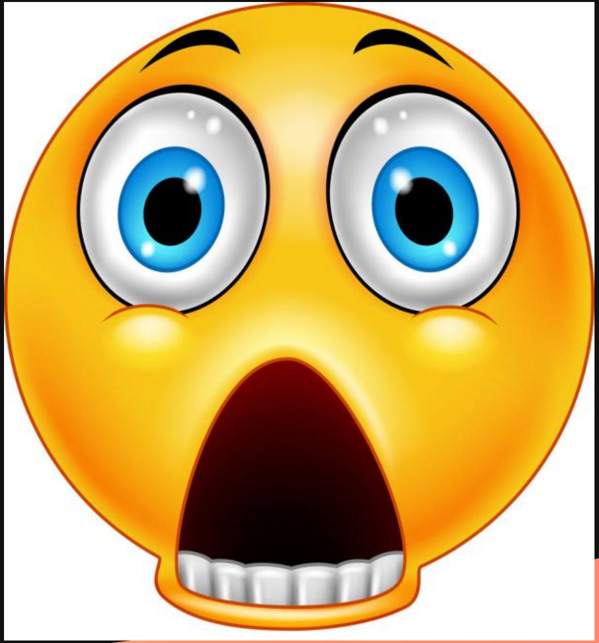
```
// Create a Cat object
Cat cat("Fluffy", 5);
cat.describe();
cat.speak();

// Wait 2 seconds
Sleep(2000);
cout << endl;

// Create a "Jimmy" object
Jimmy jimmy("Jimmy", 15);
jimmy.describe();
jimmy.speak();
```



```
Fluffy said Meow!

Jimmy said I'm too old for CPP!
```

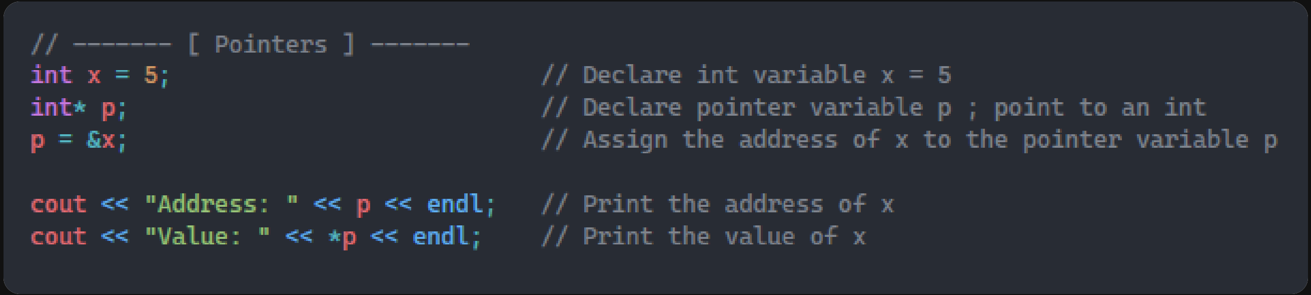
Advanced C++ Topics

Me when C++



Pointers

- Variable that stores the **memory address** of another variable.
- Used to manipulate data directly
- Access and modify the data stored in memory location **(pointed to by a pointer)**



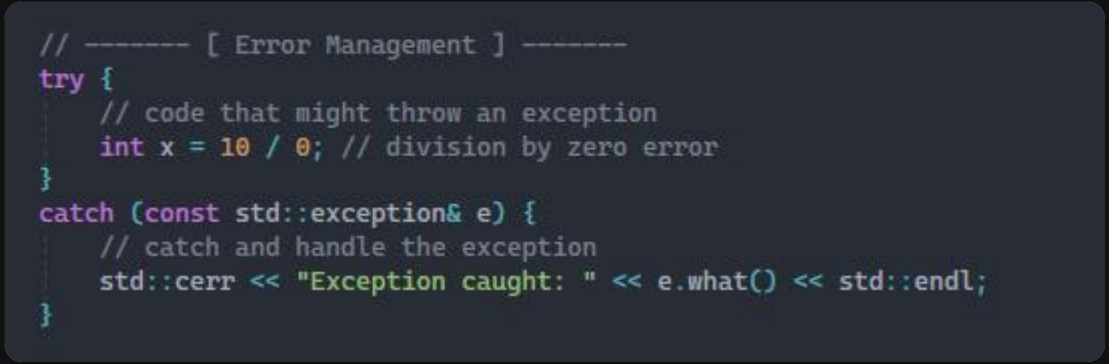
```
// ------- [ Pointers ] -------
int x = 5;                          // Declare int variable x = 5
int* p;                             // Declare pointer variable p ; point to an int
p = &x;                             // Assign the address of x to the pointer variable p

cout << "Address: " << p << endl;   // Print the address of x
cout << "Value: " << *p << endl;    // Print the value of x
```

Pointers

*me
*you
me = &you
you = &me

Error Management

- Process of handling errors or exceptions that occur in a program. – **Try Catch**.
- Code that might throw an exception is placed inside a "**try**" block
- Exceptions are caught and handled in a corresponding "**catch**" block.
- "**throw**" keyword is used to explicitly throw an exception

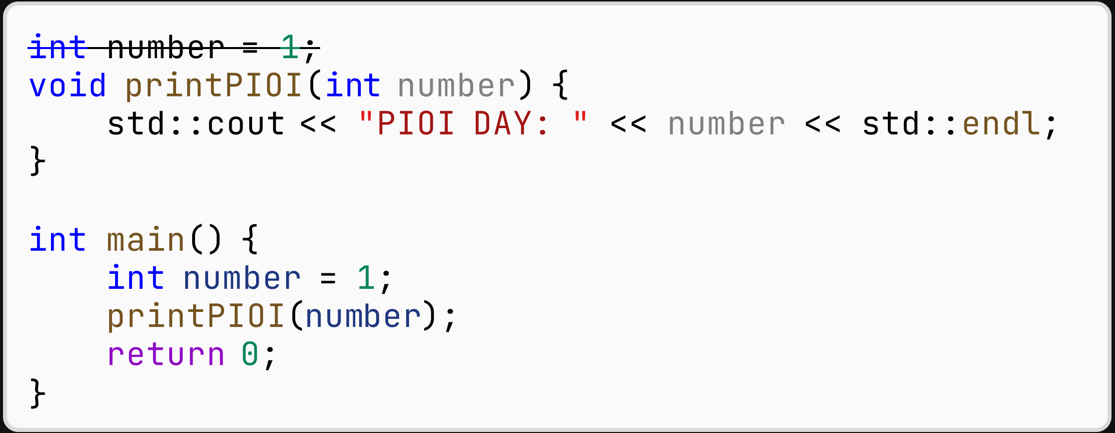


```
// -------- [ Error Management ] --------
try {
    // code that might throw an exception
    int x = 10 / 0; // division by zero error
}
catch (const std::exception& e) {
    // catch and handle the exception
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

Best Practices

Memory & Resource Management

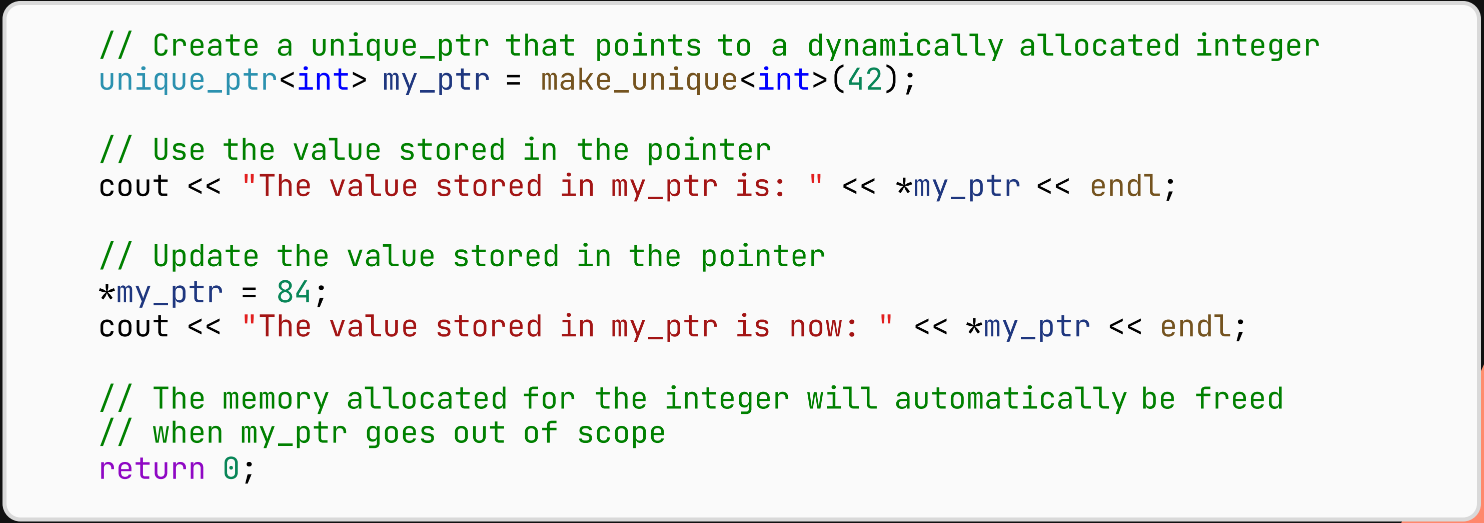
- Avoid Global Variables
  - Name Collisions
  - Dependency issues
  - Testing / Tracking Difficulties



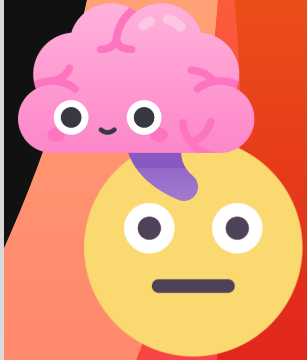
```
~~int number = 1;~~
void printPIOI(int number) {
    std::cout << "PIOI DAY: " << number << std::endl;
}

int main() {
    int number = 1;
    printPIOI(number);
    return 0;
}
```

Memory & Resource Management

- Use Smart Pointers
- Avoid using Raw Pointers



```
// Create a unique_ptr that points to a dynamically allocated integer
unique_ptr<int> my_ptr = make_unique<int>(42);

// Use the value stored in the pointer
cout << "The value stored in my_ptr is: " << *my_ptr << endl;

// Update the value stored in the pointer
*my_ptr = 84;
cout << "The value stored in my_ptr is now: " << *my_ptr << endl;

// The memory allocated for the integer will automatically be freed
// when my_ptr goes out of scope
return 0;
```



Debugging & Testing Techniques

- Use “assert” to validate parameters (and for unit tests)
- Set Breakpoints ◍ (using a debugger) to help find issues in execution
- Use automated testing frameworks (E.g. Google Test, Boost.Test)
- Unit Tests to check that indiv. functions are working

```cpp
#include <cassert>
#include <iostream>

// Function to calculate the factorial of a number
int factorial(int n) {
    assert(n >= 0);                  // Validate input parameter using an assertion
    int result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}

// Unit test for the factorial function
void testFactorial() {
    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
    assert(factorial(5) == 120);
}

int main() {
    // Use the debugger to step through the code (Set break points 🟢)
    int num = 5;
    std::cout << "The factorial of " << num << " is: " << factorial(num) << std::endl;

    // Use automated testing framework to run the unit test (e.g., Google Test)
    testFactorial();

    return 0;
}
```

Thank you!