

# Introduction to C++ Programming

Yulei Sui

University of Technology Sydney, Australia

# Introduction to C++ Programming

What is C++?

- A general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm.

# Introduction to C++ Programming

What is C++?

- A general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm.

Why learn C++?

- Language for building system software (e.g., operating systems, web browsers, game engines, database engines, language runtimes and cloud/distributed systems)

# Introduction to C++ Programming

What is C++?

- A general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm.

Why learn C++?

- Language for building system software (e.g., operating systems, web browsers, game engines, database engines, language runtimes and cloud/distributed systems)
- Object-oriented yet high performance
- Pointer and direct memory-access

# Introduction to C++ Programming

What is C++?

- A general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm.

Why learn C++?

- Language for building system software (e.g., operating systems, web browsers, game engines, database engines, language runtimes and cloud/distributed systems)
- Object-oriented yet high performance
- Pointer and direct memory-access
- One of the most popular languages and fastest-growing in 2020
  - [www.techrepublic.com/article/c-is-now-the-fastest-growing-programming-language](http://www.techrepublic.com/article/c-is-now-the-fastest-growing-programming-language)
  - [www.techrepublic.com/article/most-popular-programming-languages-c-knocks-python-out-of-top-three](http://www.techrepublic.com/article/most-popular-programming-languages-c-knocks-python-out-of-top-three)

# Introduction to C++ Programming

- This short introduction does not aim to cover every detailed aspect of C++, but rather the basic C++ syntax/features in order to develop algorithms to fulfil the assignment tasks in this subject.

# Introduction to C++ Programming

- This short introduction does not aim to cover every detailed aspect of C++, but rather the basic C++ syntax/features in order to develop algorithms to fulfil the assignment tasks in this subject.
- You are encouraged to learn and practice more advanced C++ syntax/features.
  - [https://www.w3schools.com/cpp/cpp\\_intro.asp](https://www.w3schools.com/cpp/cpp_intro.asp)
  - <https://www.youtube.com/watch?v=BC1S40yzssA>
  - Google search 'C++ programming' or 'introduction to C++ programming'

# Configure Your Integrated Development Environment (IDE)

VSCode + Docker:

[https://github.com/SVF-tools/SVF-Teaching/wiki/  
Installation-of-Docker,-VSCode-and-its-extensions](https://github.com/SVF-tools/SVF-Teaching/wiki/Installation-of-Docker,-VSCode-and-its-extensions)



# Write Your First C++ Program

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World! \n";
    return 0;
}
```

A Hello World example under SVF-Teaching:

<https://github.com/SVF-tools/SVF-Teaching/blob/main/HelloWorld/hello.cpp>

# C++ Primitive Data Types and Variables

- 'type variable = value; '
  - Primitive types including int, float, double, char, bool, string.

```
int myNum = 5;           // Integer (whole number)
float myFloatNum = 5.99; // Floating point number
double myDoubleNum = 9.98; // Floating point number
char myLetter = 'D';     // Character
bool myBoolean = true;   // Boolean
string myText = "Hello"; // String
```

# C++ Classes and Objects

- C++ class: new data type compared with C for
  - **Abstraction:** "shows" essential attributes and "hides" unnecessary information
  - **Encapsulation:** 'expose' only the interfaces and hide implementation details
- A C++ class is a template for objects, and an object is an instance of a class.

# C++ Classes and Objects

- C++ class: new data type compared with C for
  - **Abstraction:** "shows" essential attributes and "hides" unnecessary information
  - **Encapsulation:** 'expose' only the interfaces and hide implementation details
- A C++ class is a template for objects, and an object is an instance of a class.

```
#include <iostream>
using namespace std;
class Graph {           // the class
    private:            // private access specifier
        int numOfNodes; // hidden attribute from outside
        int numOfEdges; // hidden attribute from outside
    public:             // public access specifier
        // interface to outside world
        int getNumOfNodes(){ return numOfNodes;}
        // interface to outside world
        void setNumOfNodes(int n){ numOfNodes = n;}
};
```

# C++ Classes and Objects

- C++ class: new data type compared with C for
  - **Abstraction:** "shows" essential attributes and "hides" unnecessary information
  - **Encapsulation:** 'expose' only the interfaces and hide implementation details
- A C++ class is a template for objects, and an object is an instance of a class.

```
#include <iostream>
using namespace std;
class Graph {           // the class
    private:            // private access specifier
        int numOfNodes; // hidden attribute from outside
        int numOfEdges; // hidden attribute from outside
    public:             // public access specifier
        // interface to outside world
        int getNumOfNodes(){ return numOfNodes;}
        // interface to outside world
        void setNumOfNodes(int n){ numOfNodes = n;}
};
```

```
int main() {
    // create an object of Graph
    Graph graphObj;
    // Access attribute via interface
    graphObj.setNumOfNodes(10);
    // print out value of the attribute
    cout << graphObj.getNumOfNodes();
    cout << "\n";
}
```

# Constructor

- A constructor is a special method automatically called when an object is created.

```
#include <iostream>
```

```
using namespace std;
```

```
class Graph {           // the class
```

```
private:                // private access specifier
```

```
    int numOfNodes; // hidden attribute from outside
```

```
    int numOfEdges; // hidden attribute from outside
```

```
public:                  // public access specifier
```

```
    Graph(int n, int e){ // constructor
```

```
        numOfNodes = n;
```

```
        numOfEdges = e;
```

```
    }
```

```
    // interface to outside world
```

```
    int getNumOfNodes(){ return numOfNodes;}
```

```
};
```

```
int main() {
```

```
    // Create an object via its constructor
```

```
    Graph graphObj(5,10);
```

```
    // print out value of the attribute
```

```
    cout << graphObj.getNumOfNodes();
```

```
    cout << "\n";
```

```
}
```

# Containers/Collections

A container is an object that stores a collection of elements.

- Standard container type
  - Plain C array `int myNum[3] = {10, 20, 30};`
- C++ STL container types.
  - **Sequence containers** (data structures accessed sequentially)
    - `vector`: Dynamic contiguous array (class template)
    - `deque`: Double-ended queue (class template)
    - `list`: Doubly-linked list (class template)
    - `stack`: Last In First Out (class template)
  - **Associative containers** (sorted data structures that can be quickly searched)
    - `set`: Collection of unique keys, sorted by keys (class template)
    - `map`: Collection of key-value pairs, sorted by keys, keys are unique (class template).

# Containers/Collections

```
#include <vector>
#include <iostream>
using namespace std;
int main ()
{
    vector<int> nodeIDs;
    nodeIDs.push_back(1);
    nodeIDs.push_back(2);
    nodeIDs.push_back(2);
    // iterating elements via loop
    for(auto i : nodeIDs)
        cout << i << "\n";
}
```



# Containers/Collections

```
#include <vector>
#include <iostream>
using namespace std;
int main ()
{
    vector<int> nodeIDs;
    nodeIDs.push_back(1);
    nodeIDs.push_back(2);
    nodeIDs.push_back(2);
    // iterating elements via loop
    for(auto i : nodeIDs)
        cout << i << "\n";
}
```

```
#include <set>
#include <iostream>
using namespace std;
int main ()
{
    set<int> nodeIDs;
    nodeIDs.insert(1);
    nodeIDs.insert(2);
    nodeIDs.insert(2);
    // iterating elements via loop
    for(auto i : nodeIDs)
        cout << i << "\n";
}
```

# Containers/Collections Used in a Class

```
#include <set>
using namespace std;
class Graph {
private:
    int numOfNodes;
    int numOfEdges;
    set<int> nodeIDs;
public:
    Graph(int n, int e) {
        numOfNodes = n;
        numOfEdges = e;
    }
    void addNode(int id){
        nodeIDs.insert(id);
    }
};

int main() {
    // Create an object of Graph
    Graph graphObj(5,10);
    // Increase nodes;
    graphObj.addNode(1);
    graphObj.addNode(2);
}
```

# Pointers for Primitive Types

- The memory address of a variable can be taken through the & operator.
- A pointer however, is a variable that stores the memory address as its value.

```
int nodeID = 5;    // A nodeID variable of type int
int* ptr = &nodeID;    // A pointer `ptr` storing the address of nodeID
```

```
cout << nodeID << "\n";
```

```
cout << &nodeID << "\n";
```

```
cout << ptr << "\n";
```

```
cout << *ptr << "\n";
```

# Pointers for Primitive Types

- The memory address of a variable can be taken through the & operator.
- A pointer however, is a variable that stores the memory address as its value.

```
int nodeID = 5;  // A nodeID variable of type int
int* ptr = &nodeID;  // A pointer `ptr` storing the address of nodeID
// Output the value of NodeID (i.e., 5)
cout << nodeID << "\n";
// Output the memory address of NodeID (e.g., 0x6dfed4)
cout << &nodeID << "\n";
// Output the memory address of nodeID with the pointer (e.g., 0x6dfed4)
cout << ptr << "\n";
// Output the value of nodeID via dereferencing the pointer ptr
cout << *ptr << "\n";
```

## References for Primitive Types

- When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting '&' in the declaration.

```
int nodeID = 5;    // A nodeID variable of type int
int& ref = nodeID; // `ref` is a reference to nodeID.
```

```
ref = 20;
cout << "nodeID = " << nodeID << endl ;
```

```
nodeID = 30;
cout << "ref = " << ref << endl ;
```

## References for Primitive Types

- When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting '&' in the declaration.

```
int nodeID = 5;    // A nodeID variable of type int
int& ref = nodeID; // `ref` is a reference to nodeID.
```

```
ref = 20;          // Value of nodeID is now changed to 20
cout << "nodeID = " << nodeID << endl ;
```

```
nodeID = 30;       // Both nodeID and ref are now 30
cout << "ref = " << ref << endl ;
```

# C++ const Type Qualifier

- The **const** keyword allows you to specify whether or not a variable is modifiable. It can help (1) document your program more clearly and (2) enable more compiler optimization opportunities.

*// a constant integer.*

*// modifying `nodeID` will get a compilation error.*

```
const int nodeID = 5;
```

*// pointer to a const variable.*

*// `ptr` is a pointer that can point to a const int type variable.*

*// modifying `nodeID` via `\*ptr` will get a compilation error.*

```
const int* ptr = &nodeID;
```

*// const Pointer.*

*// `cptr` is a pointer, which is const, that points to an int.*

*// modifying `cptr` will get a compilation error*

```
int* const cptr = &nodeID;
```

# Parameter Passing using Pointers and References

- Both references and pointers can be used to change local variables of one function inside another function.

```
/// parameters as values  
/// (pass by value)  
void swap(int n1, int n2){  
    int tmp = n1;  
    n1 = n2;  
    n2 = tmp;  
}  
  
int main(){  
    int node1 = 2, node2 = 3;  
    swap(node1, node2);  
    cout << node1 << " " << node2;  
}
```



# Parameter Passing using Pointers and References

- Both references and pointers can be used to change local variables of one function inside another function.

```
/// parameters as values  
/// (pass by value)  
void swap(int n1, int n2){  
    int tmp = n1;  
    n1 = n2;  
    n2 = tmp;  
}  
  
int main(){  
    int node1 = 2, node2 = 3;  
    swap(node1, node2);  
    cout << node1 << " " << node2;  
}
```

**pass by value:** caller and callee have two independent variables with the same value (effect not visible to caller)

# Parameter Passing using Pointers and References

- Both references and pointers can be used to change local variables of one function inside another function.

*/// parameters as values*

*/// (pass by value)*

```
void swap(int n1, int n2){
```

```
    int tmp = n1;
```

```
    n1 = n2;
```

```
    n2 = tmp;
```

```
}
```

```
int main(){
```

```
    int node1 = 2, node2 = 3;
```

```
    swap(node1, node2);
```

```
    cout << node1 << " " << node2;
```

```
}
```

**pass by value:** caller and callee have two independent variables with the same value (effect not visible to caller)

*/// parameters as references*

*/// (Pass by reference)*

```
void swap(int& n1, int& n2){
```

```
    int tmp = n1;
```

```
    n1 = n2;
```

```
    n2 = tmp;
```

```
}
```

```
int main(){
```

```
    int node1 = 2, node2 = 3;
```

```
    swap(node1, node2);
```

```
    cout << node1 << " " << node2;
```

```
}
```

# Parameter Passing using Pointers and References

- Both references and pointers can be used to change local variables of one function inside another function.

*/// parameters as values*

*/// (pass by value)*

```
void swap(int n1, int n2){
```

```
    int tmp = n1;
```

```
    n1 = n2;
```

```
    n2 = tmp;
```

```
}
```

```
int main(){
```

```
    int node1 = 2, node2 = 3;
```

```
    swap(node1, node2);
```

```
    cout << node1 << " " << node2;
```

```
}
```

**pass by value:** caller and callee have two independent variables with the same value (effect not visible to caller)

*/// parameters as references*

*/// (Pass by reference)*

```
void swap(int& n1, int& n2){
```

```
    int tmp = n1;
```

```
    n1 = n2;
```

```
    n2 = tmp;
```

```
}
```

```
int main(){
```

```
    int node1 = 2, node2 = 3;
```

```
    swap(node1, node2);
```

```
    cout << node1 << " " << node2;
```

```
}
```

**passed by reference:** caller and callee share the same variable for the parameter (effect visible to caller)

# Parameter Passing using Pointers and References

- Both references and pointers can be used to change local variables of one function inside another function.

*/// parameters as values*

*/// (pass by value)*

```
void swap(int n1, int n2){
    int tmp = n1;
    n1 = n2;
    n2 = tmp;
}

int main(){
    int node1 = 2, node2 = 3;
    swap(node1, node2);
    cout << node1 << " " << node2;
}
```

**pass by value:** caller and callee have two independent variables with the same value (effect not visible to caller)

*/// parameters as references*

*/// (Pass by reference)*

```
void swap(int& n1, int& n2){
    int tmp = n1;
    n1 = n2;
    n2 = tmp;
}

int main(){
    int node1 = 2, node2 = 3;
    swap(node1, node2);
    cout << node1 << " " << node2;
}
```

**passed by reference:** caller and callee share the same variable for the parameter (effect visible to caller)

*/// parameters as pointers*

*/// (Pass by pointers)*

```
void swap(int* n1, int* n2){
    int tmp = *n1;
    *n1 = *n2;
    *n2 = tmp;
}

int main(){
    int node1 = 2, node2 = 3;
    swap (&node1, &node2);
    cout << node1 << " " << node2;
}
```

# Parameter Passing using Pointers and References

- Both references and pointers can be used to change local variables of one function inside another function.

*/// parameters as values*  
*/// (pass by value)*

```
void swap(int n1, int n2){
    int tmp = n1;
    n1 = n2;
    n2 = tmp;
}

int main(){
    int node1 = 2, node2 = 3;
    swap(node1, node2);
    cout << node1 << " " << node2;
}
```

**pass by value:** caller and callee have two independent variables with the same value (effect not visible to caller)

*/// parameters as references*  
*/// (Pass by reference)*

```
void swap(int& n1, int& n2){
    int tmp = n1;
    n1 = n2;
    n2 = tmp;
}

int main(){
    int node1 = 2, node2 = 3;
    swap(node1, node2);
    cout << node1 << " " << node2;
}
```

**passed by reference:** caller and callee share the same variable for the parameter (effect visible to caller)

*/// parameters as pointers*  
*/// (Pass by pointers)*

```
void swap(int* n1, int* n2){
    int tmp = *n1;
    *n1 = *n2;
    *n2 = tmp;
}

int main(){
    int node1 = 2, node2 = 3;
    swap (&node1, &node2);
    cout << node1 << " " << node2;
}
```

**pass by pointer:** caller and callee share the same variable via pointer dereferences (effect visible to caller)

# Parameter Passing using Pointers and References

- Both of them can also be used to **save copying of big objects** when passed as arguments to functions or returned from functions, to be more efficient.

```
class Graph {  
public:  
    int numOfNodes;  
    int numOfEdges;  
};
```

*// If we remove `\*` or `&` in below functions, a new copy of the graph object is created.  
// `const` used to avoid accidentally updates `g` as the purpose is to print `g` only.*

```
void print(const Graph *g){  
    cout << g->numOfNodes << " " << g->numOfEdges << " ";  
}  
void print(const Graph &g){  
    cout << g.numOfNodes << " " << g.numOfEdges << " ";  
}
```

# Using Pointers in Classes

```
#include <iostream>
using namespace std;
class Node {      // The class
private:
    int nodeID;   // Node ID
public:           // Access specifier
    Node(int i){ nodeID = i; } // constructor
    int getNodeID() { return nodeID;}
};
```

```
class Edge {      // The class
private:          // Access specifier
    Node* src;    // source node of an edge
    Node* dst;    // target node of an edge
public:
    Edge(Node* s, Node* d){ // constructor
        src = s; dst = d;
    }
    Node* getSrc() { return src;}
    Node* getDst() { return dst;}
};
```

# Using Pointers in Classes

```
#include <iostream>
using namespace std;
class Node {      // The class
private:
    int nodeID;   // Node ID
public:           // Access specifier
    Node(int i){ nodeID = i; } // constructor
    int getNodeID() { return nodeID; }
};

class Edge {      // The class
private:          // Access specifier
    Node* src;    // source node of an edge
    Node* dst;    // target node of an edge
public:
    Edge(Node* s, Node* d){ // constructor
        src = s; dst = d;
    }
    Node* getSrc() { return src; }
    Node* getDst() { return dst; }
};
```

```
int main () {
    Node* srcNode = new Node(1);
    Node* dstNode = new Node(2);
    // Assess public member functions or attributes
    // through field access `->` operator
    // similar to pointer dereferences
    cout << srcNode->getNodeID() << " ";
    cout << dstNode->getNodeID() << "\n";

    Edge* edge = new Edge(srcNode, dstNode);
    cout << edge->getSrc()->getNodeID() << " ";
    cout << edge->getDst()->getNodeID() << "\n";
}
```



# Putting All the Above Classes Together to Build a Graph

```
#include <set>

using namespace std; class Edge;

class Node {
private:
    int nodeID;
    set<Edge*> outEdges; // outgoing edges
public:
    Node(int i){ nodeID = i; }
    int getNodeID() { return nodeID;}
    set<Edge*>& getOutEdges(){ return outEdges;}
};

class Edge {
private:
    Node* src;
    Node* dst;
public:
    Edge(Node* s,Node* d){ src = s; dst = d; }
    Node* getSrc() { return src;}
    Node* getDst() { return dst;}
};

class Graph { // The class
private: // Access specifier
    set<Node*> nodes; // a set of nodes
public:
    Graph() { } // constructor
    set<Node*>& getNodes(){ return nodes;}
};
```

# Putting All the Above Classes Together to Build a Graph

```
#include <set>

using namespace std; class Edge;

class Node {
private:
    int nodeID;
    set<Edge*> outEdges; // outgoing edges
public:
    Node(int i){ nodeID = i; }
    int getNodeID() { return nodeID;}
    set<Edge*>& getOutEdges(){ return outEdges;}
};

class Edge {
private:
    Node* src;
    Node* dst;
public:
    Edge(Node* s,Node* d){ src = s; dst = d; }
    Node* getSrc() { return src;}
    Node* getDst() { return dst;}
};

class Graph {
private:
    set<Node*> nodes; // a set of nodes
public:
    Graph() { }
    set<Node*>& getNodes(){ return nodes;}
};

int main () {
    Node* src = new Node(1);
    Node* dst = new Node(2);
    Edge* edge = new Edge(src,dst);
    // add src's outgoing edge
    src->getOutEdges().insert(edge);
    // create a graph object
    Graph* graph = new Graph();
    // add two nodes into the graph
    graph->getNodes().insert(src);
    graph->getNodes().insert(dst);
}
```

# C++ Inheritance

Allow a child class to inherit attributes and methods from its parent class.

# C++ Inheritance

Allow a child class to inherit attributes and methods from its parent class.

```
class GraphBuilder{
public:
    GraphBuilder(){}

    void build(){
        cout << "parent's way to build..\n";
        Node* src = new Node(1);
        Node* dst = new Node(2);
        Edge* edge = new Edge(src,dst);
        // add src's outgoing edge
        src->addOutEdge(edge);
        // create a graph object
        Graph* graph = new Graph();
        // add two nodes into the graph
        graph->addNode(src);
        graph->addNode(dst);
    }
};
```

# C++ Inheritance

Allow a child class to inherit attributes and methods from its parent class.

```
class GraphBuilder{
public:
    GraphBuilder(){}

    void build(){
        cout << "parent's way to build..\n";
        Node* src = new Node(1);
        Node* dst = new Node(2);
        Edge* edge = new Edge(src,dst);
        // add src's outgoing edge
        src->addOutEdge(edge);
        // create a graph object
        Graph* graph = new Graph();
        // add two nodes into the graph
        graph->addNode(src);
        graph->addNode(dst);
    }
};

// SubGraphBuilder is a child (derived) class
// of GraphBuilder
class SubGraphBuilder : public GraphBuilder{
public:
    SubGraphBuilder(){}
};

int main () {
    SubGraphBuilder* builder = new SubGraphBuilder();
    // reuse the build method in GraphBuilder
    builder->build();
}
```

# C++ Function Overriding

Allow a child class to override a function (with same signature) in its parent class.

# C++ Function Overriding

Allow a child class to override a function (with same signature) in its parent class.

```
class GraphBuilder{
public:
    GraphBuilder(){}

    void build(){
        cout << "parent's way to build..\n";
        Node* src = new Node(1);
        Node* dst = new Node(2);
        Edge* edge = new Edge(src,dst);
        // add src's outgoing edge
        src->addOutEdge(edge);
        // create a graph object
        Graph* graph = new Graph();
        // add two nodes into the graph
        graph->addNode(src);
        graph->addNode(dst);
    }
};
```

```
class SubGraphBuilder : public GraphBuilder{
public:
    SubGraphBuilder(){}
    // override `build` method in GraphBuilder
    void build(){
        cout << "child's way to build..\n";
    }
};

int main () {
    SubGraphBuilder* builder1 = new SubGraphBuilder();
    // Which `build` method will be called?
    builder1->build();

    GraphBuilder* builder2 = new SubGraphBuilder();
    // Which `build` method will be called?
    builder2->build();
}
```

## C++ Virtual Function and Polymorphism

A function declared with a '**virtual**' keyword in a parent class can be overridden by a child class. When you refer to **a child class object** using a **pointer/reference to the parent class**, it will call child class's version of this virtual function.



# C++ Virtual Function and Polymorphism

A function declared with a '**virtual**' keyword in a parent class can be overridden by a child class. When you refer to **a child class object** using a **pointer/reference to the parent class**, it will call child class's version of this virtual function.

```
class GraphBuilder{
public:
    GraphBuilder(){}
    virtual void build(){
        cout << "parent's way to build..\n";
        Node* src = new Node(1);
        Node* dst = new Node(2);
        Edge* edge = new Edge(src,dst);
        // add src's outgoing edge
        src->addOutEdge(edge);
        // create a graph object
        Graph* graph = new Graph();
        // add two nodes into the graph
        graph->addNode(src);
        graph->addNode(dst);
    }
};
```

```
class SubGraphBuilder : public GraphBuilder{
public:
    SubGraphBuilder(){}
    void build(){ // override `build` in GraphBuilder
        cout << "child's way to build..\n";
    }
};

int main () {
    SubGraphBuilder* builder1 = new SubGraphBuilder();
    builder1->build(); // Which `build` will be called?

    GraphBuilder* builder2 = new SubGraphBuilder();
    builder2->build(); // Which `build` will be called?

    GraphBuilder* builder3 = new GraphBuilder();
    builder3->build(); // Which `build` will be called?
}
```

# Debugging Your C++ Programs

- VSCode (<https://code.visualstudio.com/docs/cpp/cpp-debug>)
- GDB (<https://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html>)
- LLDB (<https://lldb.llvm.org/use/tutorial.html>)
- Eclipse CDT (<https://wiki.eclipse.org/CDT/StandaloneDebugger>)
- Other tactics, such as printing your results  
(<https://www.learncpp.com/cpp-tutorial/basic-debugging-tactics/>)