

C++

# 程式語言（二）

Introduction to Programming (II)

Templates

Joseph Chuang-Chieh Lin

Dept. CSE, NTOU

# Platform/IDE

- Dev-C++



<https://www.pngegg.com/en/search?q=Dev-C>

- Codeblocks



<https://icons8.com/icons/set/code-blocks>

- OnlineGDB (<https://www.onlinegdb.com/>)



- Real-Time Collaborative Online IDE (<https://ide.usaco.guide/>)



# Textbooks (We focusing on C++11)

- *Learn C++ Programming by Refactoring* ( 由重構學習 C++ 程式設計 ). Pang-Feng Liu ( 劉邦鋒 ). NTU Press. 2023.
- *C++ Primer. 5th Edition.* Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. 2019.
- *Effective C++.* Scott Meyers. O'Reilly. 2016.
- *Thinking in C++. Vol. 1: Introducing to Standard C++.* 2nd Edition. Bruce Eckel. Prentice Hall PTR. 2000.

# Useful Resources

- Tutorialspoint
  - <https://www.tutorialspoint.com/cplusplus/index.htm>
  - Online C++ Compiler
- Programiz
  - <https://www.programiz.com/cpp-programming>
- LEARN C++
  - <https://www.learncpp.com/>
- MIT OpenCourseWare - Introduction to C++
  - <https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/pages/lecture-notes/>
- Learning C++ Programming
  - <https://www.programiz.com/cpp-programming>
- GeeksforGeeks
  - <https://www.geeksforgeeks.org/c-plus-plus/>

# Template

- **Goal:** Pass **data type** as a **parameter** so that we don't have to write the same code or the same function for different data types.
- For example, we can write one function `sort()` and pass data type as a parameter so that we can sort data of many kinds of types.
- When does template expand/kick in?

# Template

- **Goal:** Pass **data type** as a **parameter** so that we don't have to write the same code or the same function for different data types.
- For example, we can write one function `sort()` and pass data type as a parameter so that we can sort data of many kinds of types.
- When does template expand/kick in?
  - Compile time.
  - The source code contains only the function or class, yet the compiled code contains multiple copies of it.

# Function Template

- A generic function that can be used for different data types.

```
template <class T> T flex_max(T x, T y) {  
    return (x > y) ? x : y;  
}
```

```
int main() {  
    cout << flex_max<int>(3, 7) << endl;  
    cout << flex_max<double>(3.0, 7.0)  
        << endl;  
    cout << flex_max<char>('g', 'e')  
        << endl;  
  
    return 0;  
}
```

# Function Template

- A generic function that can be used for different data types.

```
template <typename T> T flex_max(T x, T y) {  
    return (x > y) ? x : y;  
}
```

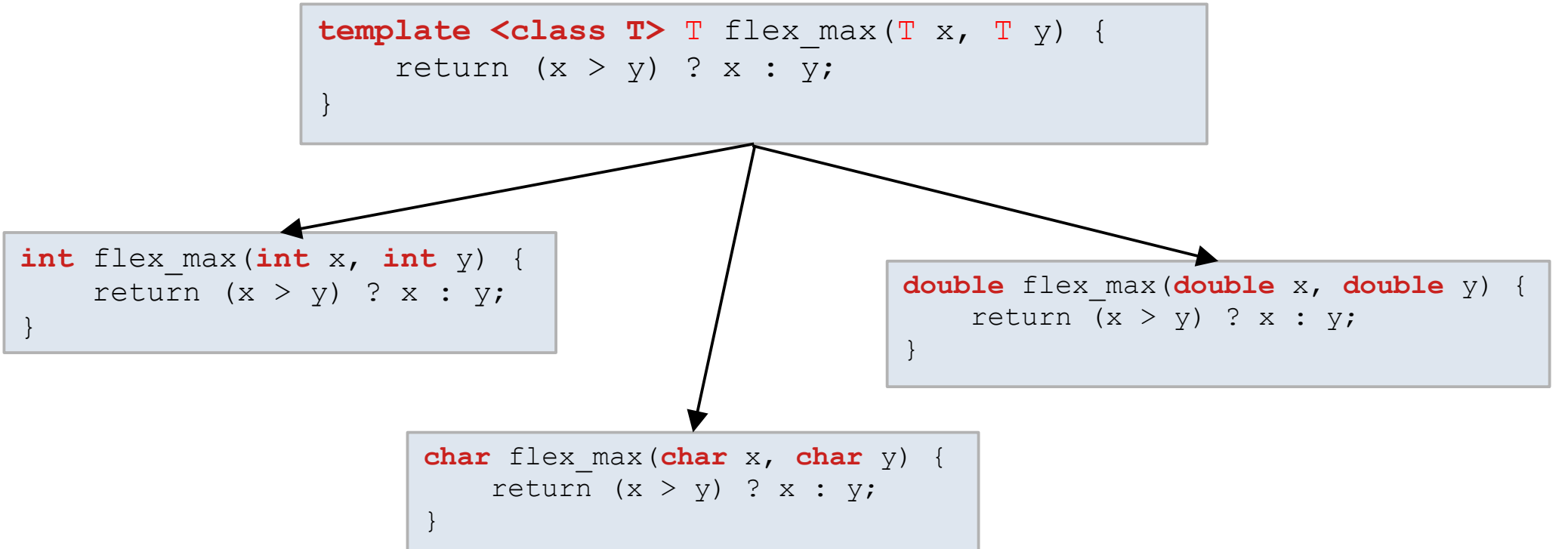
→ either "typename" or  
"class" will be fine

```
int main() {  
    cout << flex_max<int>(3, 7) << endl;  
    cout << flex_max<double>(3.0, 7.0)  
        << endl;  
    cout << flex_max<char>('g', 'e')  
        << endl;  
  
    return 0;  
}
```



# Function Template

```
template <class T> T flex_max(T x, T y) {  
    return (x > y) ? x : y;  
}
```



```
graph TD; A["template <class T> T flex_max(T x, T y) {  
    return (x > y) ? x : y;  
}"] --> B["int flex_max(int x, int y) {  
    return (x > y) ? x : y;  
}"]; A --> C["double flex_max(double x, double y) {  
    return (x > y) ? x : y;  
}"]; A --> D["char flex_max(char x, char y) {  
    return (x > y) ? x : y;  
}"];
```

```
int flex_max(int x, int y) {  
    return (x > y) ? x : y;  
}
```

```
double flex_max(double x, double y) {  
    return (x > y) ? x : y;  
}
```

```
char flex_max(char x, char y) {  
    return (x > y) ? x : y;  
}
```

# Function Template

- A generic function that can be used for different data types.

```
template <class T> void try_swap(T& x, T& y) {  
    T temp = x;  
    x = y;  
    y = temp;  
}
```

```
int main() {  
    int a = 5, b = 3;  
    double c = 11.2, d = 23.7;  
    try_swap<int>(a, b);  
    try_swap<double>(c, d);  
    cout << a << ", " << b << endl;  
    cout << c << ", " << d << endl;  
    return 0;  
}
```

# Function Template

- A generic function that can be used for different data types.

```
template <class T>
void try_swap(T& x, T& y) {
    T temp = x;
    x = y;
    y = temp;
}
```

```
int main() {
    int a = 5, b = 3;
    double c = 11.2, d = 23.7;
    try_swap<int>(a, b);
    try_swap<double>(c, d);
    cout << a << ", " << b << endl;
    cout << c << ", " << d << endl;
    return 0;
}
```

# Example: Bubble sort

<https://www.geeksforgeeks.org/templates-cpp/>

```
template <class T> void bubbleSort(T a[], int n) {  
    for (int i=0; i<n-1; i++)  
        for (int j=n-1; i<j; j--)  
            if (a[j] < a[j-1])  
                swap(a[j], a[j-1]);  
}
```

i					j	
2	1	6	4	5		

```
int main() {  
    int a[5] = { 10, 50, 30, 40, 20 };  
    int n = sizeof(a) / sizeof(a[0]);  
  
    bubbleSort<int>(a, n);  
    for (int i = 0; i < n; i++) // print the sorted array  
        cout << a[i] << " ";  
    cout << endl;  
  
    return 0;  
}
```

# Example: Bubble sort

<https://www.geeksforgeeks.org/templates-cpp/>

```
template <class T> void bubbleSort(T a[], int n) {  
    for (int i=0; i<n-1; i++)  
        for (int j=n-1; i<j; j--)  
            if (a[j] < a[j-1])  
                swap(a[j], a[j-1]);  
}
```

i		j		
2	1	6	<b>5</b>	4

```
int main() {  
    int a[5] = { 10, 50, 30, 40, 20 };  
    int n = sizeof(a) / sizeof(a[0]);  
  
    bubbleSort<int>(a, n);  
    for (int i = 0; i < n; i++) // print the sorted array  
        cout << a[i] << " ";  
    cout << endl;  
  
    return 0;  
}
```

# Example: Bubble sort

<https://www.geeksforgeeks.org/templates-cpp/>

```
template <class T> void bubbleSort(T a[], int n) {  
    for (int i=0; i<n-1; i++)  
        for (int j=n-1; i<j; j--)  
            if (a[j] < a[j-1])  
                swap(a[j], a[j-1]);  
}
```

i		j		
2	1	6	5	4

```
int main() {  
    int a[5] = { 10, 50, 30, 40, 20 };  
    int n = sizeof(a) / sizeof(a[0]);  
  
    bubbleSort<int>(a, n);  
    for (int i = 0; i < n; i++) // print the sorted array  
        cout << a[i] << " ";  
    cout << endl;  
  
    return 0;  
}
```

# Example: Bubble sort

<https://www.geeksforgeeks.org/templates-cpp/>

```
template <class T> void bubbleSort(T a[], int n) {  
    for (int i=0; i<n-1; i++)  
        for (int j=n-1; i<j; j--)  
            if (a[j] < a[j-1])  
                swap(a[j], a[j-1]);  
}
```

i	j			
2	<b>6</b>	1	5	4

```
int main() {  
    int a[5] = { 10, 50, 30, 40, 20 };  
    int n = sizeof(a) / sizeof(a[0]);  
  
    bubbleSort<int>(a, n);  
    for (int i = 0; i < n; i++) // print the sorted array  
        cout << a[i] << " ";  
    cout << endl;  
  
    return 0;  
}
```

# Example: Bubble sort

<https://www.geeksforgeeks.org/templates-cpp/>

```
template <class T> void bubbleSort(T a[], int n) {  
    for (int i=0; i<n-1; i++)  
        for (int j=n-1; i<j; j--)  
            if (a[j] < a[j-1])  
                swap(a[j], a[j-1]);  
}
```

j

i

6	2	1	5	4
---	---	---	---	---

```
int main() {  
    int a[5] = { 10, 50, 30, 40, 20 };  
    int n = sizeof(a) / sizeof(a[0]);  
  
    bubbleSort<int>(a, n);  
    for (int i = 0; i < n; i++) // print the sorted array  
        cout << a[i] << " ";  
    cout << endl;  
  
    return 0;  
}
```



# Class Template

- Similar to function templates, but useful for **classes** (e.g., LinkedList, BinaryTree, Stack, Queue, ...)

# Example: A Generic Array

```
template <class T> class Array {  
private:  
    T* ptr;  
    int size;  
  
public:  
    Array(T arr[], int s);  
    void print();  
};
```

```
template <class T>  
Array<T>::Array(T arr[], int s) {  
    ptr = new T[s];  
    size = s;  
    for (int i = 0; i < size; i++)  
        ptr[i] = arr[i];  
}
```

```
template <class T>  
void Array<T>::print() {  
    for (int i = 0; i < size; i++)  
        cout << " " << *(ptr + i); //OK!  
    cout << endl;  
}
```

```
int main() {  
    int arr[5] = { 1, 2, 3 };  
    Array<int> a(arr, 3);  
    a.print();  
    return 0;  
}
```

# Example: A Generic Array

```
template <class T> class Array {  
private:  
    T* ptr;  
    int size;  
  
public:  
    Array(T arr[], int s);  
    void print();  
};
```

```
template <class T>  
Array<T>::Array(T arr[], int s) {  
    ptr = new T[s];  
    size = s;  
    for (int i = 0; i < size; i++)  
        ptr[i] = arr[i];  
}
```

```
template <class T>  
void Array<T>::print() {  
    for (int i = 0; i < size; i++)  
        cout << " " << ptr[i]; // OK!  
    cout << endl;  
}
```

```
int main() {  
    int arr[5] = { 1, 2, 3 };  
    Array<int> a(arr, 3);  
    a.print();  
    return 0;  
}
```

# Another Example: Matrix

<https://www.cs.uregina.ca/Links/class-info/115/07-templates/>

```
class Matrix {
private:
    int twoDimArray[MAXROWS][MAXCOLS];
    int rows;
    int cols;
public:
    Matrix(); // constructor
    void printMatrix();
    void setElement(int row, int col, int value); //set an element of the matrix
    void setMatrix(int[][MAXCOLS]); //set the twoDimArray to what is sent
    void addMatrix(int[][MAXCOLS]); //add an array to twoDimArray
    void addMatrix(int[][MAXCOLS], int[][MAXCOLS]); //add two arrays together
};
```

Try to make it a template class

# Another Example: Matrix

<https://www.cs.uregina.ca/Links/class-info/115/07-templates/>

```
template <typename M_type>
class Matrix {
private:
    M_type twoDimArray[MAXROWS][MAXCOLS];
    int rows;
    int cols;
public:
    Matrix();
    void printMatrix();
    void setElement(int row, int col, M_type value); //set an element of the matrix
    void setMatrix(M_type [][][MAXCOLS]); //set the twoDimArray to what is sent
    void addMatrix(M_type [][][MAXCOLS]); //add an array to twoDimArray
    void addMatrix(M_type [][][MAXCOLS], M_type [][][MAXCOLS]); //add two arrays together
};
```

# Class Instantiation

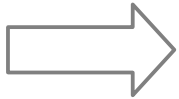
<https://www.cs.uregina.ca/Links/class-info/115/07-templates/>

```
Matrix<float> floatMatrix;
```

# Member Function Definition

<https://www.cs.uregina.ca/Links/class-info/115/07-templates/>

```
void Matrix::addMatrix(int otherArray[][MAXCOLS]) {  
    for (int i=0; i< rows; i++) {  
        for(int j=0; j< cols; j++) {  
            twoDimArray[i][j] += otherArray[i][j];  
        }  
    }  
}
```



```
template <typename M_type>  
void Matrix<M_type>::addMatrix(M_type otherArray[][MAXCOLS]) {  
    for (int i=0; i< rows; i++) {  
        for(int j=0; j< cols; j++) {  
            twoDimArray[i][j] += otherArray[i][j];  
        }  
    }  
}
```

# Function Overloading vs. Templates

- When should we use templates?
  - When we want to perform **the same action** just on different types.

```
template <typename T>  
T foo(const T& a, const T& b) { return a + b; }
```



# Function Overloading vs. Templates

- When should we use templates?
  - When we want to perform **the same action** just on different types.
- When should we use function overloading?
  - When we may apply **different operations on different types**.

```
class Foo{ void run() const {} };  
  
void foo(int i) { std::cout << "i = " << i << "\n"; }  
void foo(const Foo& f) { f.run(); }
```

# Exercise: print\_max()

```
Template ... {  
    /* implement the function template print_max*/  
}
```

```
int main() {  
    int a[6] = { 10, 50, 30, 40, 20, -20 };  
    float b[] = { 2.3, 0.0, -1.2, 17.2 };  
    char c[] = "TKUCS";  
    int n1 = sizeof(a) / sizeof(a[0]);  
    int n2 = sizeof(b) / sizeof(b[0]);  
    int n3 = sizeof(c) / sizeof(c[0]);  
    print_max<int>(a, n1);  
    print_max<float>(b, n2);  
    print_max<char>(c, n3);  
    return 0;  
}
```

Output:

```
50  
17.2  
U
```



# Combining Operator Overloading

# Combining operator overloading

```
struct Node {
    int data;
    int order; // the order of generation
    Node *next;
    Node() { // constructor }
    ~Node() { // destructor }
    static int counter;
    // overloading '<' and '<<'
};

int Node::counter = 0; // total #objects
```

Sample input :

10 20 -5 77 29

Sample output :

2:-5 0:10 1:20 4:29 3:77

```
template <class T>
void bubbleSort(T a[], int n) {
    for (int i=0; i<n-1; i++)
        for (int j=n-1; i<j; j--)
            if (a[j] < a[j-1])
                swap(a[j], a[j-1]);
}
```

```
int main() {
    int n = 5;
    Node *dataList = new Node[n];
    bubbleSort<Node>(dataList, n);
    for (int i = 0; i < n; i++)
        cout << dataList[i] << " ";
    cout << endl;
    delete [] dataList;
    return 0;
}
```

# More Arguments to Templates

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

template <class T, class U> class A {
    T x;
    U y;
public:
    A() { cout << "Constructor Called\n"; }
    A(T a, U b): x(a), y(b) {
        cout << x << ", " << y << endl;
    }
};

int main() {
    A<char, char> a;
    A<int, double> b;
    A<int, char> c(100, 'T');
    return 0;
}
```

# Example: A Generic Array

```
template <class T, int size>
class Array {
private:
    T myArr[size];

public:
    Array(T arr[]);
    void print();
};
```

```
template <class T, int size>
Array<T,size>::Array(T arr[]) {
    ptr = new T[size];
    for (int i = 0; i < size; i++)
        myArr[i] = arr[i];
}
```

```
template <class T, int size>
void Array<T,size>::print() {
    int i = 0;
    for (int i = 0; i < s; i++)
        cout << " " << myArr[i];
    cout << endl;
}
```

```
int main() {
    int arr[3] = { 1, 2, 3};
    Array<int, 3> a(arr, 3);
    a.print();
    return 0;
}
```

# Remark

- Both function overloading and templates are examples of polymorphism of OOP.
  - *Function overloading*: multiple functions do similar tasks.
  - *Templates*: multiple functions do **identical** tasks.

# Inheritance of a Template

```
template<typename T>
class Base {
public:
    Base(T data): mData(data) { }
    virtual void print() {
        cout << mData << endl;
    }
protected:
    T mData;
};
```

```
template<typename T>
class Derived : public Base<T> {
public:
    Derived() = default;
    Derived(T data) : Base<T>(data) { }

    void print() override {
        cout << "data = "
             << this->mData << endl;
    }
};
```

```
int main() {
    Derived<float> d1(5.2f);
    Derived<std::string> d2("TKU_CSIE");
    d1.print();
    d2.print();
    return 0;
}
```



# Exercise

- Please modify the following class `List` so that the main function can run successfully.

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

```
class List {
public:
    List() : head_(nullptr) { }
    virtual void add(int n) {
        Link *p = new Link(n, head_);
        head_ = p;
    }
    void print_head() {
        cout << "head: " << head_->val << endl;
    }
private:
    struct Link {
        int val;
        Link *next;
        Link(int n, Link* nxt): val(n), next(nxt) { }
    };
    Link * head_;
};
```

```
int main() {
    List<int> nums;
    nums.add(1);
    nums.add(2);
    nums.print_head();
    return 0;
}
```

Sample output

```
head: 2
```



# Class Template Specialization

# Class Template Specialization

- **Template Specialization:**
  - It is possible to override the template-generated code by providing specific **definitions** for specific **types**.

# Template Specialization

<https://onlinegdb.com/UQn8nDP9o>

```
template <class T>
class CHECK {
    public:
        void f() { cout << "CHECK<T>::f()" << endl ;}
};
```

```
template <>
class CHECK<char> {
    public:
        void f() { cout << "CHECK<char>::f()" << endl ;}
};
```

```
int main() {
    CHECK<int> c1;
    CHECK<char> c2;

    c1.f();
    c2.f();
    return 0;
}
```

# Template Class Partial Specialization

- **Template Partial Specialization:**
  - Generate a specialization of a template class for fewer parameters.

# Partial Specialization

<https://onlinegdb.com/KPkpMzQ4>

```
template<class T, class U, class V> struct S {  
    void foo() {  
        cout << "General case" << endl;  
    }  
};
```

```
template<class U, class V> struct S<int, U, V> {  
    void foo() {  
        cout << "T = int" << endl;  
    }  
};
```

```
template<class V> struct S<int, double, V> {  
    void foo() {  
        cout << "T = int, U = double" << endl;  
    }  
};
```

```
int main() {  
    S<string, int, double> obj1;  
    S<int, float, string> obj2;  
    S<int, double, string> obj3;  
  
    obj1.foo();  
    obj2.foo();  
    obj3.foo();  
    return 0;  
}
```

General case

T = int

T = int, U = double

# Exercise

- Problem 2 of the [page](https://tinyurl.com/2p93tw37) here (<https://tinyurl.com/2p93tw37>).
- **Goal:** Convert a class that is *specialized for integers* into a templated class that can *handle many types*.
- Requirement:
  - Create a templated class named `List` and correctly *initializes, manages, and de-allocated* an array of *specified length*.
  - Use the **designated** main function:

```
int main() {
    List<int> integers(10);
    for (int i = 0; i < integers.length; i++) {
        integers.set(i, i * 100);
        printf("%d ", integers.get(i));
    }
    printf("\n");
    // this loop should print: 0 100 200 300 400 500 600 700 800 900

    List<Point *> points(5);
    for (int i = 0; i < points.length; i++) {
        Point *p = new Point;
        p->x = i * 10;
        p->y = i * 100;
        points.set(i, p);
        printf("(%d, %d) ", points.get(i)->x, points.get(i)->y);
        delete p;
    }
    printf("\n");
    // this loop should print: (0, 0) (10, 100) (20, 200) (30, 300) (40, 400)
}
```



# A typedef struct

```
typedef struct  
Point_ {  
    int x;  
    int y;  
} Point;
```

# The Output

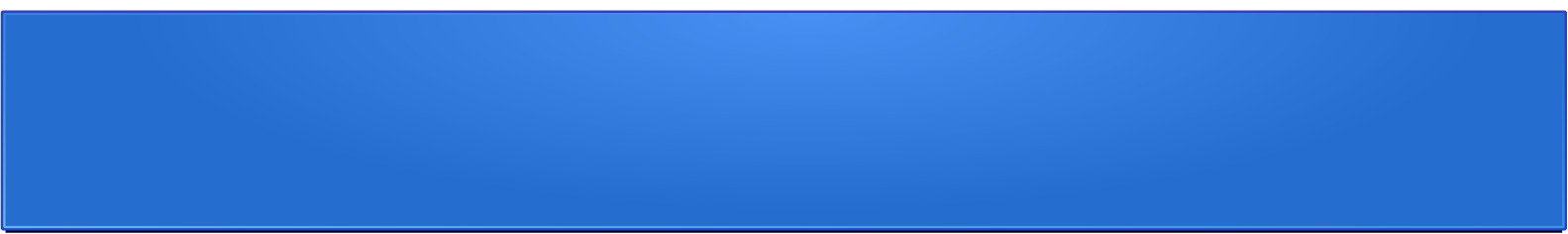
```
0 100 200 300 400 500 600 700 800 900  
(0, 0) (10, 100) (20, 200) (30, 300) (40, 400)
```

# Hint

You may refer to the code here as a reference.

```
class IntList {
    int * list;

public:
    int length;
    IntList(int len) {
        list = new int[len];
        length = len;
    }
    ~IntList() {
        delete[] list;
    }
    int get(int index) {
        return list[index];
    }
    void set(int index, int val) {
        list[index] = val;
    }
};
```



# Discussions & Questions