

CS541 6 Recitation 3

Continue with containers and classes

09/12/2025

Jamal Hashim

Logistics

Extra learning sessions: every Monday evening

- **Session 3: const & template** (prelim preparation I)
 - Time: 7:15 PM – 8:15 PM, Monday (09/15)
 - Led by TAs: Jephthah Kwame Mensah, Nam Anh Dang
 - Location: Phillips 101

Extra HW helping & working sessions:

- **Session 1: HW1 part 1** (due on September 15th)
 - Time: 2PM – 5 PM, Sunday (9/14)
 - Led by TAs: Briaana Liu, Tina Cheng, Ruichen Bao
 - Location: Uris G01

Overview

- C++ classes
 - Constructor, destructor
 - Copy constructor, move constructor
 - C++ objects and containers
- C++ templates
 - Template function, template class
 - Template instantiation
 - Variadic templates

What is C++?

A federation of related languages, with four primary sublanguages

- **C:** C++ is based on C, while offering approaches superior to C. Blocks, statements, processor, built-in data types, arrays, pointers, etc., all come from C
- **Object-Oriented C++:** “C with Classes”, classes including constructor, destructors, inheritance, virtual functions, etc.
- **Template C++:** generic programming language. Gives a template, define rules and pattern of computation, to be used across different classes.
- **STL(standard template library):** a special template library with conventions regarding containers, iterators, algorithms, and function objects

C++ Class

C++ Class

- A class is a user-defined type
- Usually, defined by header file (.hpp) and implementation file (.cpp)
- A class can have following members
 - Data members
 - Constructor, destructor
 - Member functions
 - Copy constructor, move constructors

C++ Class

Why do we need header files?

- The **names** of program elements must **be declared before** they can be used.
- The declaration **tells the compiler** the type of an element.
- C++ uses header files to contain declarations. Use **#include** in other files that require the declarations.

C++ Class

rectangle.hpp

--- header file

Data
members

```
#pragma once  
class Rectangle{  
    float width;  
    float length;  
    float area;  
public:  
    Rectangle();  
    Rectangle(float w, float l);  
    ~Rectangle();  
    float& getArea();  
    ... };
```

For compiler,
indicate that
it only be
parsed once.

C++ Class

rectangle.cpp

--- implementation file

```
#include "rectangle.hpp"
```

```
Rectangle::Rectangle(){
```

```
... ..
```

```
}
```

```
Rectangle::Rectangle(float w, float l){
```

```
... ..
```

```
}
```

```
float& Rectangle::getArea(){
```

```
... }
```

Scope resolution
operator



C++ Class

- A class is a user-defined type
- Usually, defined by header file (.hpp) and implementation file (.cpp)
- A class can have following members
 - Data members
 - **Constructor, destructor**
 - Member functions
 - Copy constructor, move constructors

Constructor: construct and initialize objects of that class

- Default constructor: a constructor can be called with no argument

```
Rectangle::Rectangle():
```

```
    width(0),
```

```
    length(0),
```

```
    area(0)
```

```
{
```

```
    // Constructor body (can be empty or contain additional logic)
```

```
}
```



Initializer list

Constructor: construct and initialize objects of that class

- Implicit default constructor:
 - If there is no user-declared constructor for a class type, **the compiler** will implicitly declare a default constructor as an inline public class member.

Constructor: construct and initialize objects of that class

Parameterized constructor: a constructor that accepts argument

```
Rectangle::Rectangle(int w, int l):
```

```
    width(w),
```

```
    length(l)
```

```
{
```

```
    area = _width * _length;
```

```
}
```

Note: When the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly create the default constructor

Constructor: construct and initialize objects of that class

Parameterized constructor: a constructor that accepts argument

Note:

- When the **parameterized constructor is defined** and no default constructor is defined explicitly, the compiler will **NOT** implicitly create the default constructor

Constructor: construct and initialize objects of that class

```
#include "rectangle.hpp"
```

```
int main(){
```

```
    Rectangle rec;
```

```
    Rectangle explicit_rec = Rectangle(10.0,12.0);
```

```
    Rectangle implicit_rec(30.0, 28.1);
```

```
}
```

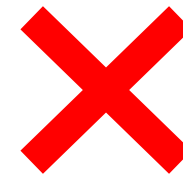
Constructor: const data member

```
class Rectangle{  
    const float width;  
    const float length;  
    const float area;  
  
public:  
    Rectangle(float w, float l, float a):  
        width(w), length(l), area(a)  
    {}  
};
```

Constructor: const data member

```
class Rectangle{  
    const float width;  
    const float length;  
    const float area;  
  
public:  
    Rectangle(float w, float l, float a) {  
        width=w;  
        length = l;  
        area = a;  
    }  
};
```

Compilation error



Recap: C++ Classes

- Once a class is defined, you can define instances(called objects)
- Unlike JAVA, class objects are NOT null references in C++
- This means that when you create an object, all of its internal must be initialized. When the object goes out of scope, it is destroyed (deconstructed).
- Each class has at least one constructor and one destructor

Question

Say you have an object in your code that may or may not exist. Since C++ doesn't allow null objects, what can you do?

Constructors

- Constructor are used to **initialize** object of the class type
- A constructor has the **same name** as the class and **no** return type. It can have as many argument as needed
- e.g.,
 - `myClass();` // default constructor
 - `myClass(int x, std::string str);` // Parameterized constructor
 - ...

Destructor

- Destructor is called when the lifetime of an object **ends**
- It is used to **free** the resources that the object acquired during its lifetime
- e.g.,
 - **`~myClass();`**

Implicit constructor and destructor

- Implicit default constructor:
 - If there is **no user-declared constructor** for a class type, **the compiler** implicitly provides a (**public**) **default constructor**.
 - If there **is** user-declared constructor, **the compiler** will **NOT** implicitly create the default constructor
- Implicit default destructor
 - The implicitly-defined destructor defined by the compiler has an empty body

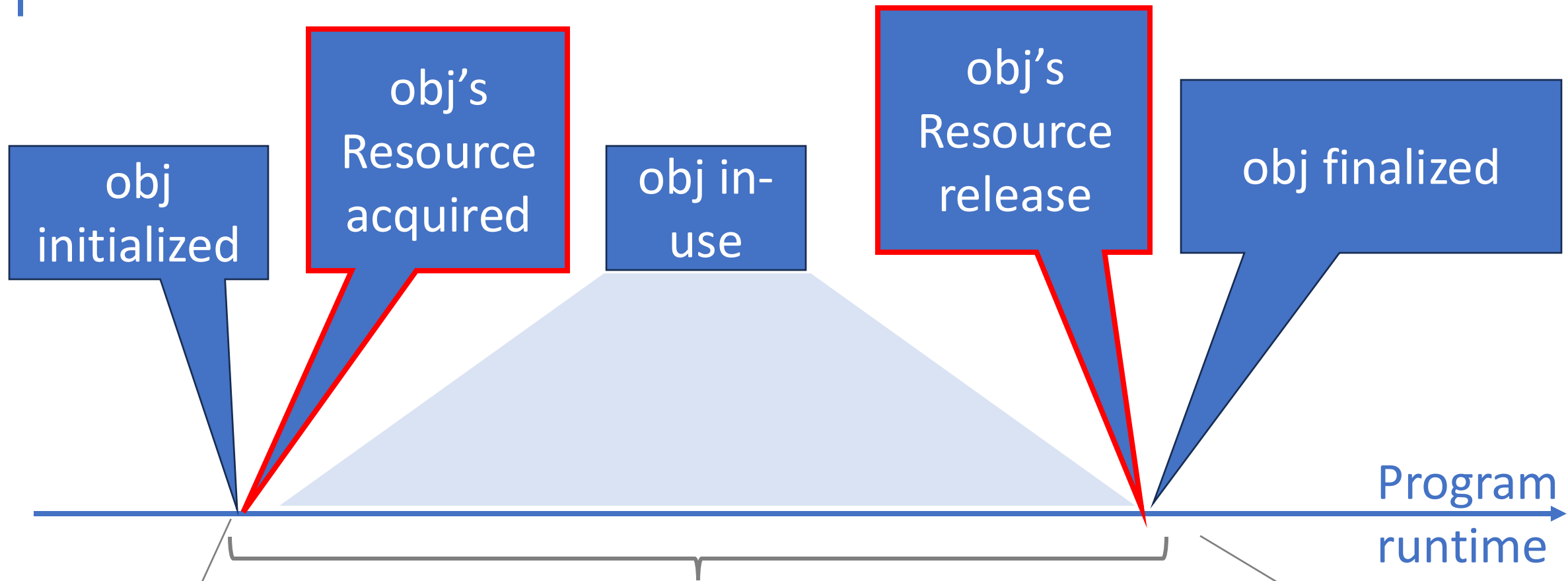
Implicit constructor and destructor



Do I ever need to
define my own
destructor?

Yes, if the object has
pointers to a runtime
allocation of resources

Object lifetime



Begin:

- Object constructor is called

An object, obj's lifetime

End

- Object destructor is called
 - Release the resources

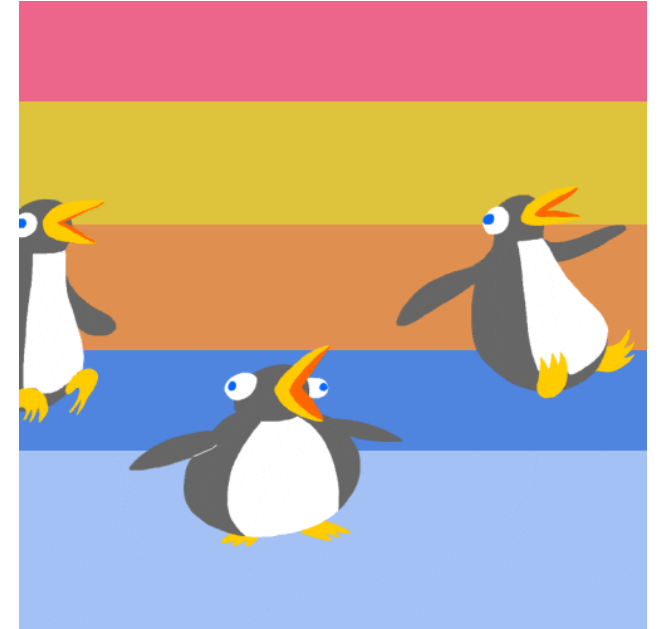
MyIntVector example

```
class MyIntVector {  
private:  
    int* data;          // Pointer to dynamically allocated array  
    size_t size;        // Number of elements in the vector  
public:  
    MyIntVector(size_t s) : size(s), data(new int[s]) {  
        for (size_t i = 0; i < size; ++i) {  
            data[i] = 0;  
        }  
    }  
    .....  
};
```

MyIntVector example

```
class MyIntVector {  
private:  
    int* data;        // Pointer to dynamically allocated array  
    size_t size;      // Number of elements in the vector  
public:  
    MyIntVector(size_t s) : size(s), data(new int[s]) {  
        ...  
    }  
    ~MyIntVector() {  
        delete[] data; // Free the dynamically allocated array  
    }  
};
```

Copying in C++



- Copying data, copying memory from one place to another
- Copying takes time

Copying in C++

```
int a = 5;  
int b = a;
```

// creating a copy of int a

```
Rectangle obj1 = Rectangle(10.0, 11.0);  
Rectangle obj2 = obj1;
```

// obj2 is a copy of object obj1

Copying in C++

```
Rectangle obj1 = Rectangle(10.0, 11.0);  
Rectangle obj2 = obj1;  
obj2.width = 100.0;
```

Now, what's the value
of obj1.wdith?

Copying in C++

```
Rectangle* obj1 = new Rectangle(10.0, 11.0);  
Rectangle* obj2 = obj1;  
Obj2->width = 100.0;
```

Now, what's the value
of obj1.wdith?

Copy constructor

```
class Rectangle{
```

```
    Rectangle(const Rectangle& other);
```

```
}
```

```
// Construct an object of class Rectangle by copying  
Rectangle object, other, passing by reference.
```

Copy constructor

- **Create** a new object by **initializing** it with an object of the same class
- Called when
 - Initialization `Rectangle obj2 = obj1;`
 - Function argument passing by value `func(Rectangle obj);`
 - Function return by value `return obj;`

Implicitly-defined default copy-constructor

- If no user-defined copy constructor is defined, the compiler will implicitly define a copy constructor for you.
- It performs member-wise copy and initializes the new object it initializes.

Do I ever need to
define my own
copy-constructor?



Yes, if an object has
pointers or any runtime
allocation of resources

Implicitly-defined default copy-constructor

- If no user-defined copy constructor, the compiler declare and define a copy constructor
 - It performs member-wise copy of the object's bases and members to the new object it initializes
 - Default constructor does only **shallow** copy

myIntVector example

```
class myIntVector{
public:
    int* data;
    size_t size;
    size_t capacity;

    myIntVector();
    myIntVector(size_t s);
    ~myIntVector();
    ...
};
```

```
myIntVector::myIntVector(size_t s) {
    size = s;
    capacity = s;
    data = new int[capacity];
    for (size_t i = 0; i < size; ++i) {
        data[i] = 0;
    }
}

myIntVector::~~myIntVector(){
    delete[] data;
}
```

Default copy-constructor

```
myIntVector vect1 = myIntVector(3);
```

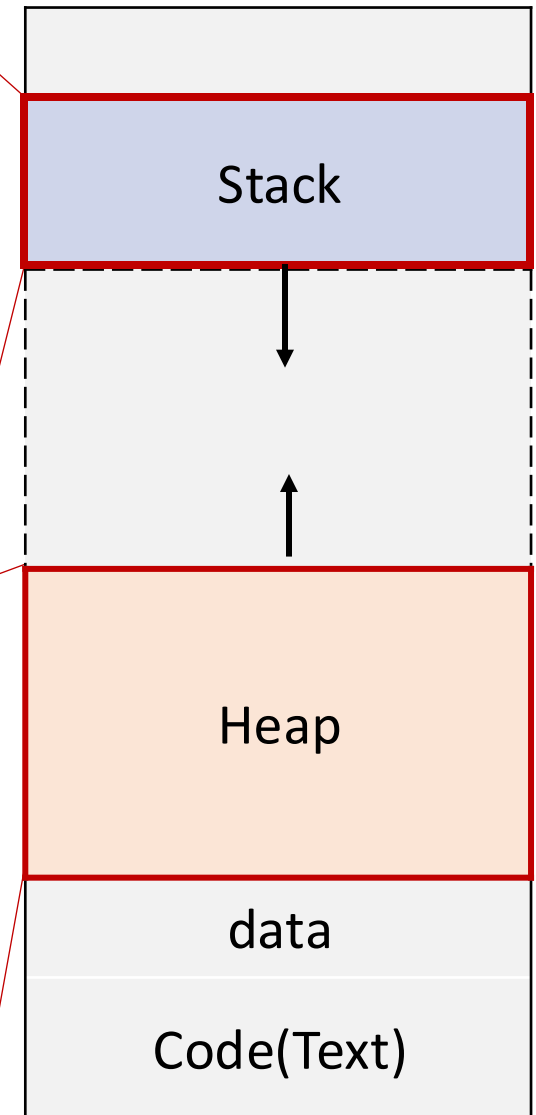
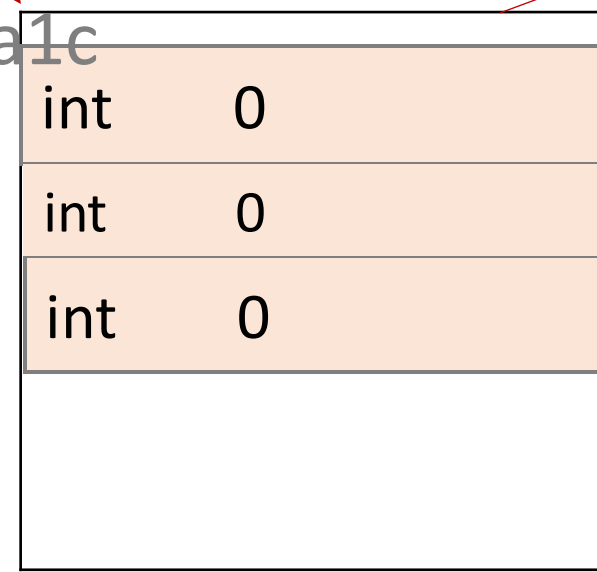
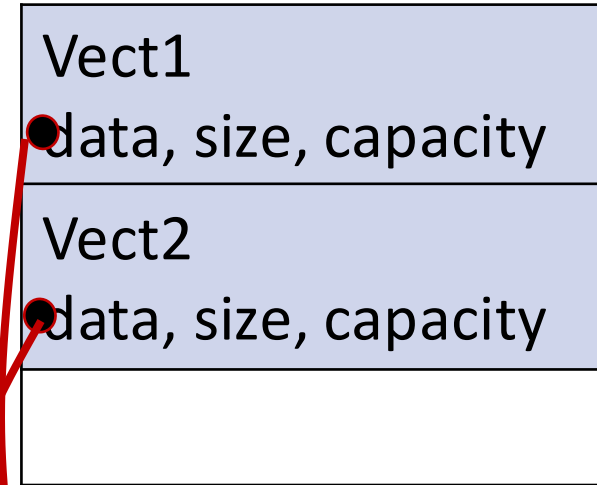
```
myIntVector vect2 = vect1;
```

```
vect1.data = 5;
```



Not ideal, because changing only one changes both of them.
Want two identical independent objects

0x7cd10a1c

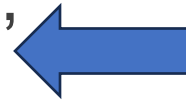


Default copy-constructor

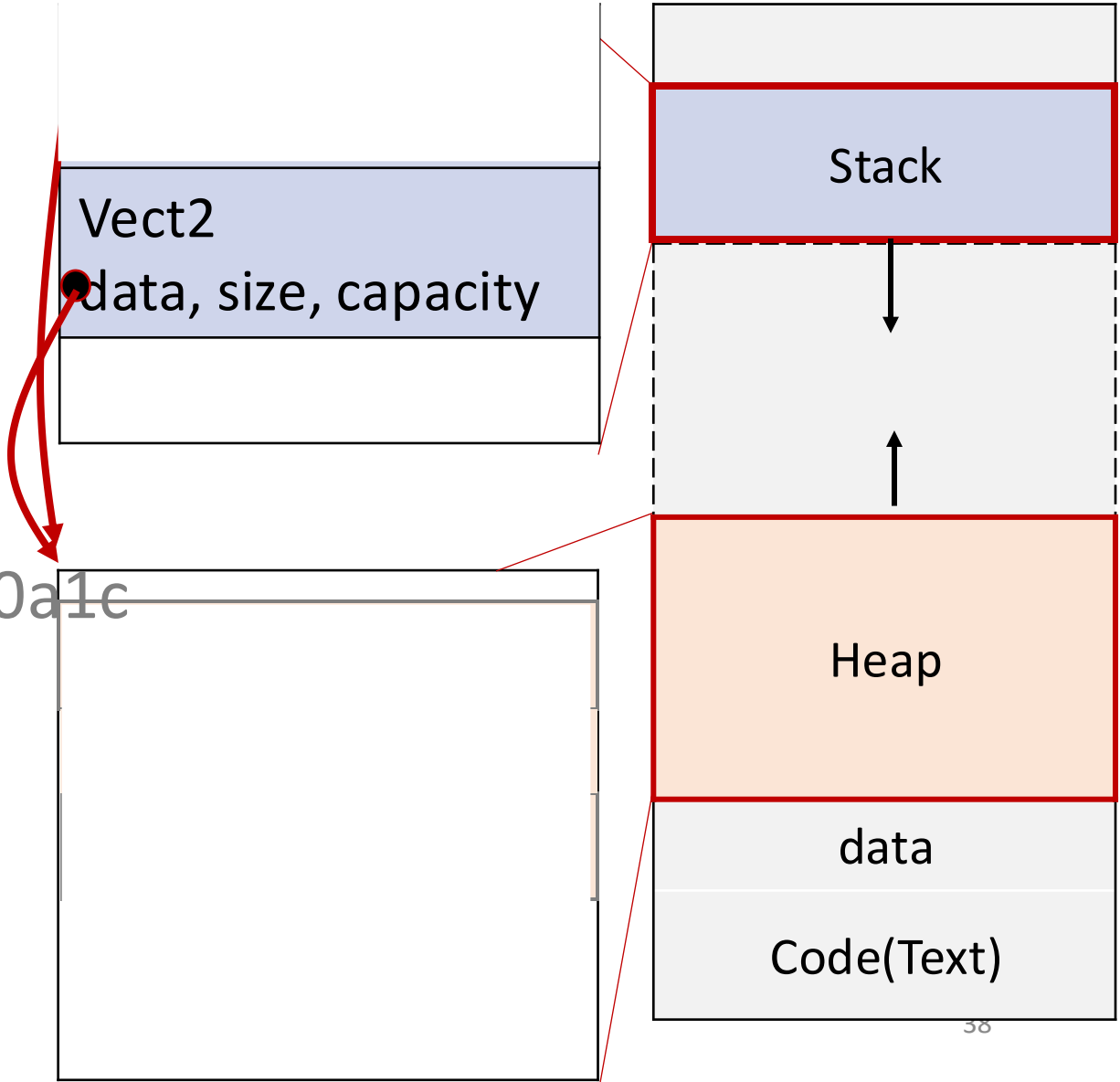
```
myIntVector func(){  
    myIntVector vect1 = myIntVector(3);  
    myIntVector vect2 = vect1;  
}  
  
myIntVector vect = func();
```



Bad, because now
vect2.data points to
nothing



0x7cd10a1c



Fix: User-defined copy constructor

```
myIntVector(const myIntVector& other) :
```

```
    size(other.size), data(new int[other.size]) {
```

```
        for (size_t i = 0; i < size; ++i) {
```

```
            data[i] = other.data[i];
```

```
        }
```

```
    }
```

Deep copy the object's
members

Move constructor

```
class myIntVector{
```

```
    myIntVector(myIntVector && other);
```

```
}
```

```
// Transfer the ownership of the resources from the  
// object, other, to the new object
```

Move constructor

- Transfer the ownership of resources from one object to another, instead of making a copy
- Called when
 - Initialization `Rectangle obj2 = std::move(obj1);`
 - Function argument passing `func(std::move(obj));`
 - Function return with Return Value Optimization(RVO)

Why use move constructor?

- Improve the performance of the program by avoiding the overhead caused by unnecessary copying.

```
myIntVector(myIntVector&& other) : size(0), data(nullptr) {  
    data = other.data;           // copy the pointer of the memory  
    size = other.size;           address of other.data  
  
    other.data = nullptr;        // Transfer the ownership of other's  
    other.size = 0;              resource to this new object  
  
}
```

Combining what we learnt about classes with vector

std::vector

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

push_back

- Appends the given element **value** to the end of the container.
- The **new** element could be constructed via

```
std::vector<Rectangle> rec_vec;
```

```
Rectangle rec1;
```

```
rec_vec.push_back(rec1);           // Copy constructor
```

```
rec_vec.push_back(std::move(rec1)); // Move constructor
```

emplace_back

- Appends a new element to the end of the container
- Besides the capabilities of push_back, it allows construct the new element **in-place**

```
std::vector<Rectangle> rec_vec;
```

```
rec_vec.emplace_back(10.0, 11.0);    // parameterized constructor
```

```
Rectangle rec1;
```

```
rec_vec.emplace_back(rec1);          // Copy constructor
```



Exercise: Find the error

```
class myClass {  
    public:  
        myClass(int x) {}  
    private:  
        int myInt;  
};  
std::vector<myClass> myObjects(4);
```

Exercise: Find the error

```
class myClass {  
public:  
    myClass(int x) {}  
  
private:  
    int myInt;  
};
```

```
std::vector<myClass> myObjects( 4 );
```

Compiler no longer provides default constructor, because of user-defined constructor

std::vector needs a way to create default-constructed elements when resizing or initializing the vector with a specified size.

Exercise: Find the error

Fix

```
std::vector<myClass> myObjects; // size 0
```

```
myClass obj1(5);
```

// constructed elements

```
myClass obj2(7);
```

```
myObjects.push_back(obj1);
```

```
myObjects.push_back(obj2);
```

// push_back invokes the
copy constructor to copy
the object into the vector

C++ Template

| What is C++?

A federation of related languages, with four primary sublanguages

- **C:** C++ is based on C, while offering approaches superior to C. Blocks, statements, processor, built-in data types, arrays, pointers, etc., all come from C
- **Object-Oriented C++:** “C with Classes”, classes including constructor, destructors, inheritance, virtual functions, etc.
- **Template C++:** generic programming language. Gives a template, define rules and pattern of computation, to be used across different classes.
- **STL(standard template library):** a special template library with conventions regarding containers, iterators, algorithms, and function objects

Motivation

- You got assigned a task at work to build a digital calculator
- You come up with something like this

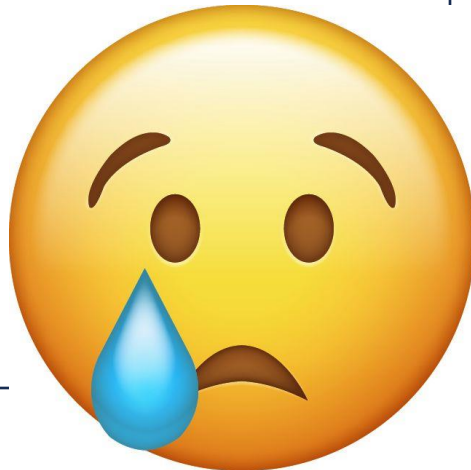
```
int subtract(int a, int b){  
    return a-b;  
}  
int main(){  
    int x = 10;  
    int y = 7;  
    int z = subtract(x,y);  
    std::cout << z << std::endl;  
    ...}
```

Motivation

```
int subtract(int a, int b){  
    return a - b;  
}  
double subtract (double a, double b){  
    return a - b;  
}  
float subtract(float a, float b){  
    return a - b;  
}  
  
int main(){  
    .....  
}
```

I don't want to
copy-paste code
just because the
type changes?

- But calculators should be able to subtract floats and doubles too!
And much more...
- So you come up with this...



Solution: generic programming

- What if you could just replace `int` with a generic data type
- How? Let's code!
- Limitation in shown example: parameters in `subtract()` must share type
- Uh-oh!
- No worries actually – let's code again!

Templated function

template < *parameter-list* > *function-declaration*

```
template <typename T>
T subtract(T a, T b) {
    return a - b;
}
```

```
int main(){
    int x = 10;
    int y = 7;
    std::cout << subtract(x, y) << std::endl;

    double p = 5.5;
    double q = 2.2;
    std::cout << subtract(p, q) << std::endl;
    ...}
```

typename is the type-parameter-key,
which could be either **typename** or **class**.

T is the name of **type template parameter**.
Tells the compiler *"I will use a placeholder type
T in this function."*

Templated class

template < *parameter-list* > *class-declaration*

```
template <typename T>
class Subtracter {
public:
    T subtract(T a, T b) {
        return a - b;
    }
};

int main() {
    Subtracter<double> double_sub;
    std::cout << double_sub.subtract(5.5, 2.2) << std::endl;
    ...}
```

Parameter list

```
template <typename T, int N>
class Array {
    T data[N]; // size N known at compile time
public:
    int size() const { return N; }
};

int main() {
    Array<int, 5> arr1; // array of 5 ints
    std::cout << arr1.size() << std::endl; // 5
    ...}
```

Non-type template parameter

- A variable for a constant (e.g., an int ...)
- Known at compile time

Type template parameter

- A variable for a type
- Known at compile time

Template instantiation

- A function template or a class template **by itself** is **not** a type, or a function, or any other entity.
- **No code is generated** from a source file that contains **only** template definitions.
- For any code to appear, a template must be **instantiated**: the template arguments must be determined so that the compiler can generate an actual function

Template instantiation - explicit instantiation

```
template <typename T>
T subtract(T a, T b) {
    return a - b;
}
```

```
template int subtract<int>(int, int);    // Explicit instantiation declarations
template double subtract<double>(double, double);    // Explicit instantiation declarations

int main(){
    int x = 10, y = 7;
    std::cout << subtract(x, y) << std::endl;    // Use subtract<int>
    double p = 5.5, q = 2.2;
    std::cout << subtract(p, q) << std::endl;    // Use subtract<double>
    ...}
```

Template instantiation - implicit instantiation (default)

```
template <typename T>
T subtract(T a, T b) {
    return a - b;
}
```

- The compiler generates code, in this case **subtract<int>**
- If subtract was called on a double another function subtract (overload) will be generated with T = double

```
int main(){
    int x = 10;
    int y = 7;
    std::cout << subtract(x, y) << std::endl; // Compiler generate subtract<int>
    double p = 5.5, q = 2.2;
    std::cout << subtract(p, q) << std::endl; // Compiler generate subtract<double>
    ...}
```

Quick aside: template .hpp files don't come with associated .cpp files

- A template is a “pattern” that the compiler uses to generate a family of classes or functions
- For the compiler to generate the code, it must see both the template definition and the specific types used to “fill in” the template.
 - For example, if you're trying to use a `subtract<int>`, the compiler must see both the `subtract` template and the fact that you're trying to make a specific `subtract<int>`

How do we use templates when our function has an arbitrary number of parameters?

- Common issue...
 - Solution: Variadic templates
 - Let's code

Variadic templates

```
class car {  
public:  
    int price;  
    car(int price) :price(price) {}  
};
```

```
class pc {  
public:  
    int price;  
    pc(int price) : price(price) {}  
};
```

```
class pen {  
public:  
    int price;  
    pen(int price) : price(price) {}  
};
```

Variadic templates

```
int sum() {  
    return 0;  
}  
template <typename T, typename... Args>  
int sum (T item, Args... rest) {  
    return item.price + sum(rest...);  
}  
  
int main() {  
    car c(100);  
    pc pc(10);  
    pen p(1);  
    std::cout << "The sum is " << sum(c, pc, p);  
}
```

Templates in the perspective of programming

- Avoid code duplication
 - Functions are blocks of organized
 - Reusable code that model a particular action
 - Classes model similar set of objects
 - Libraries provide a consistent set of features
- Performance (compile-time resolution)
 - Templates are expanded by the compiler for the types you use

Where to find the resources?

- Copy constructor: <https://www.geeksforgeeks.org/copy-constructor-in-cpp/>
- Move semantics: <https://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html>
- Operator overload: <https://www.geeksforgeeks.org/operator-overloading-cpp/>
- Effective C++: 55 specific ways to improve your programs and designs, Scott Meyers, 3rd edition
- A Tour of C++, Bjarne Stroustrup