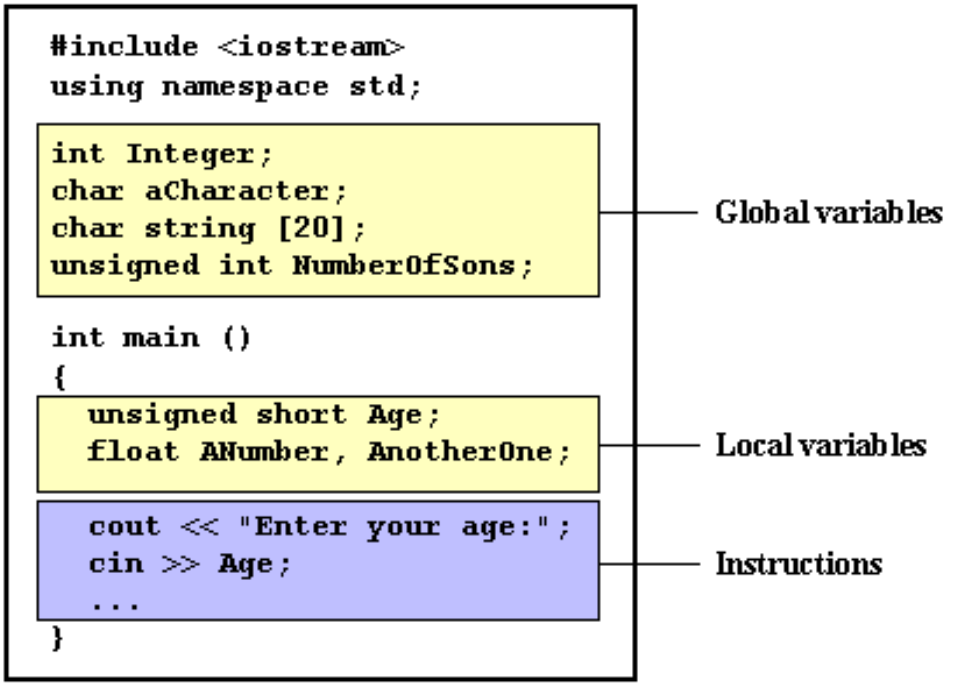
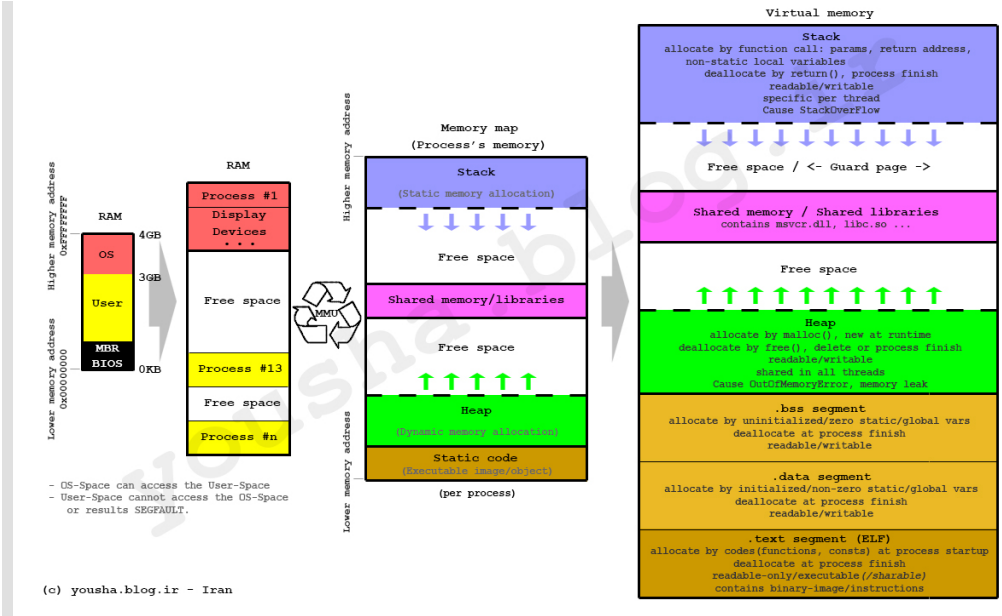
Programming Language - C & C++

Variables





Variable declaration

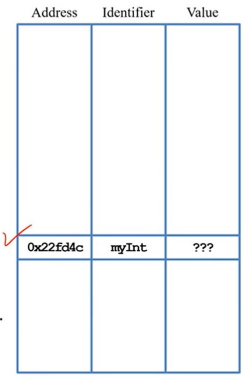
- Before we use a variable, we must first **declare** it.
  - We need to specify its **name** and **data type**.
- The syntax of a variable declaration statement is

  type variable name;

  - For example,

    ```
    int myInt;
    ```

    declares an integer variable called **myInt**.
- A variable name is an **identifier**.
  - We do not need to memorize the memory address (which is a sequence of numbers).
  - We access the space through the variable name.



Declaration and assignment

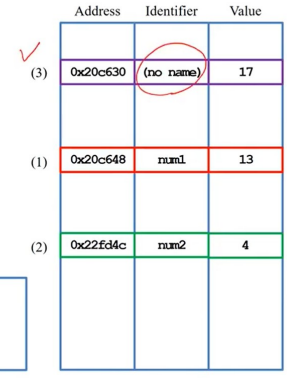
- Beside declaring a variable, we may also **assign** values to a variable.
  - **int myInt;** declares an integer variable.
  - **myInt = 10;** **assigns 10 to myInt.**
- We may do these together:

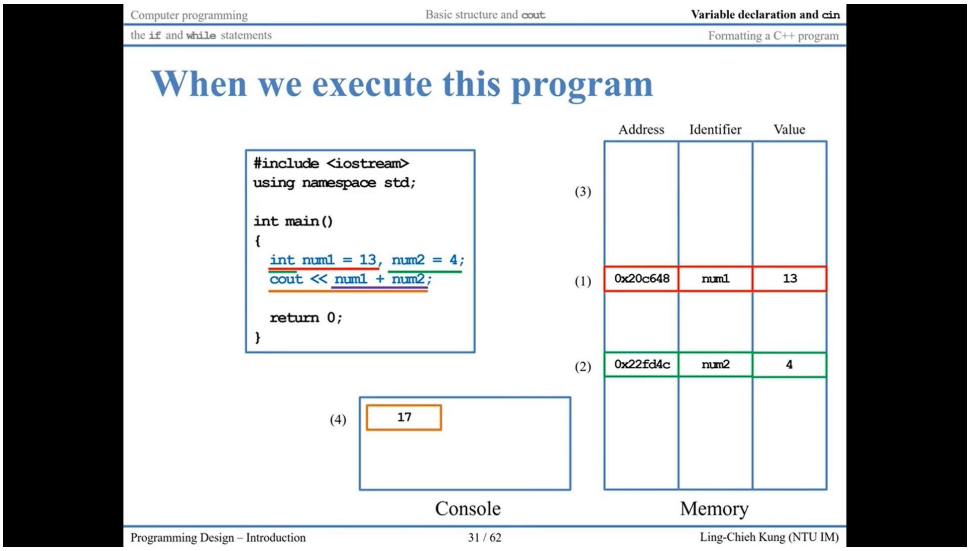
  type variable name = initial value;

  - **int yourInt = 5;** declares an integer variable **yourInt** and assigns 5 to it.
  - The assignment is called **initialization** if it is done with declaration.
- Without initialization, the variable may be of **any value** (depending on what was left since the last time this space is used)!

When we execute this program

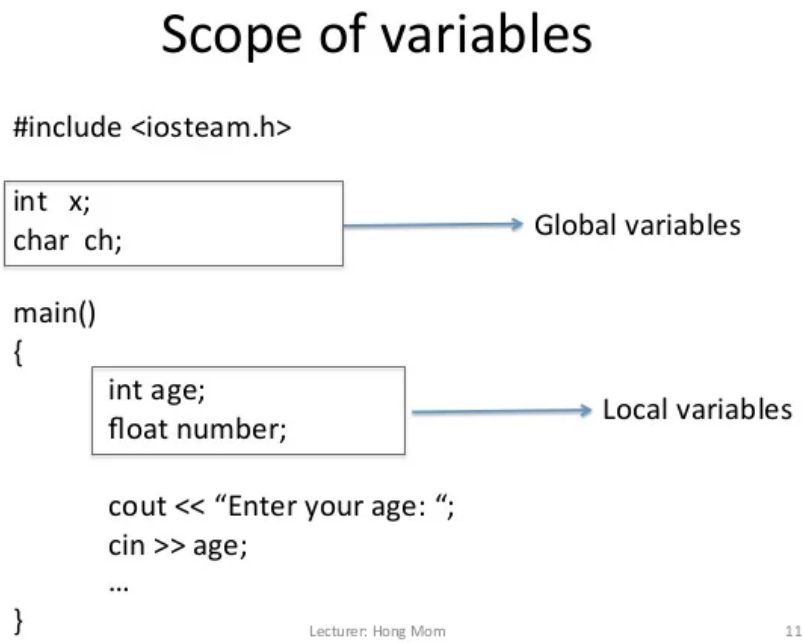
```
#include <iostream>
using namespace std;

int main()
{
  int num1 = 13, num2 = 4;
  cout << num1 + num2;

  return 0;
}
```



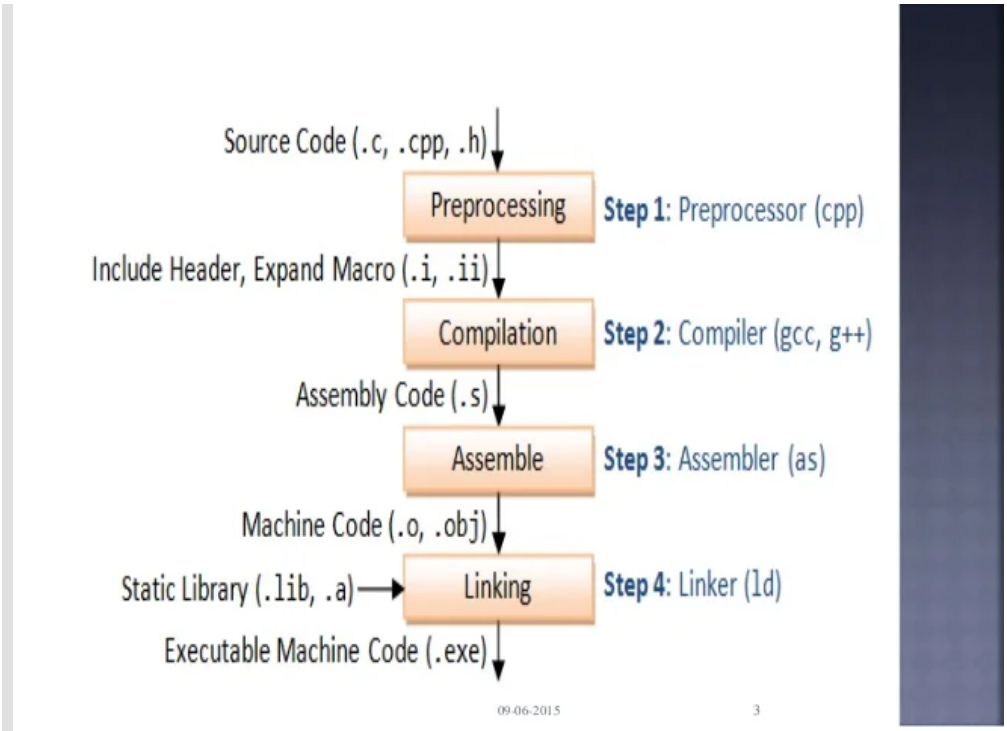


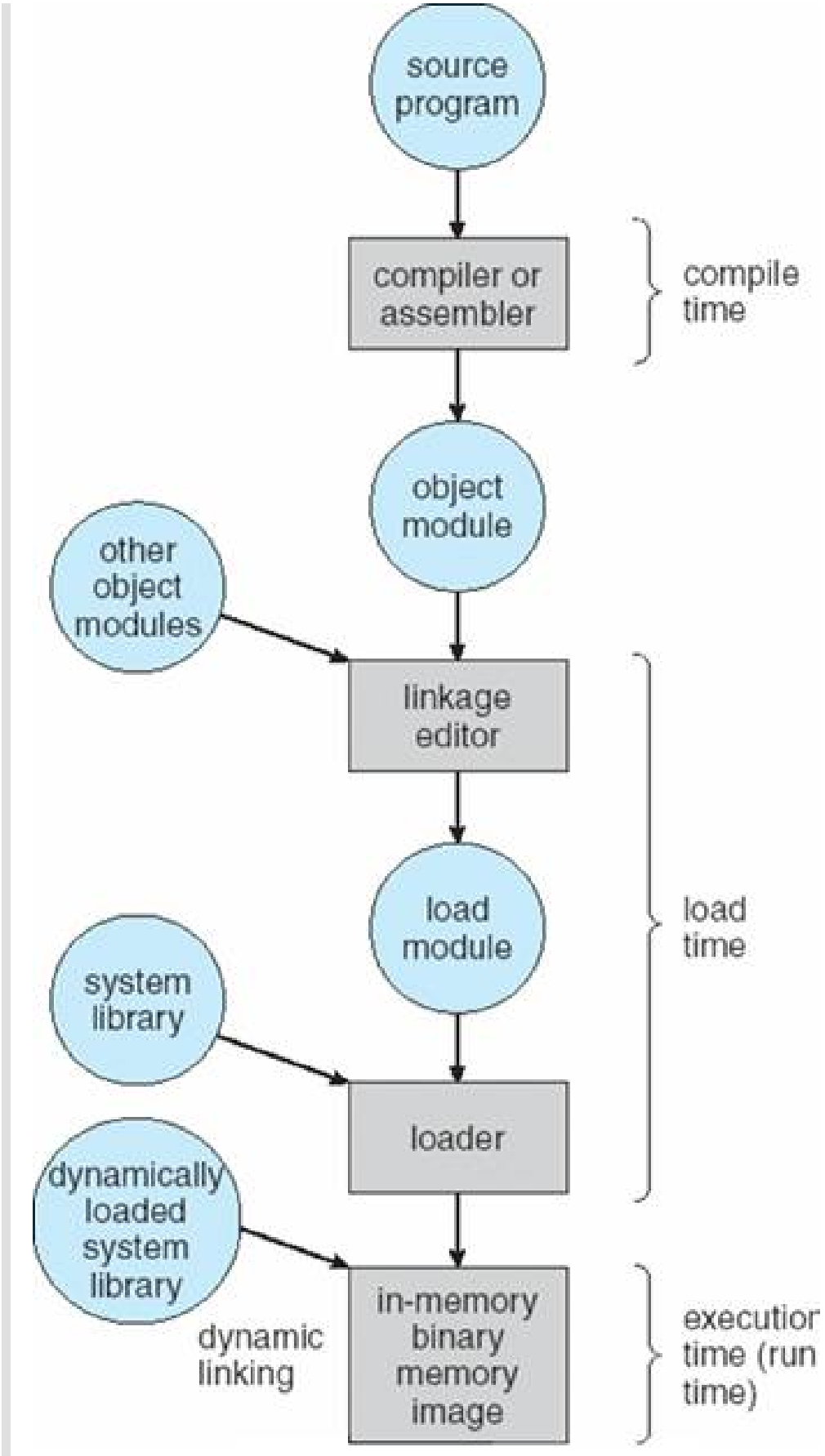
Storage Class	Specifier (Keyword)	Visibility (Scope)	Lifetime
Automatic	None (or auto)	Function	Function
Register	register	Function	Function
Local static	static	Function	Program
External (definition)	None	File (can be declared in other files)	Program
External (declaration)	extern	File	Program
External	static	File (can't be declared in other files)	Program

Scope of Variables

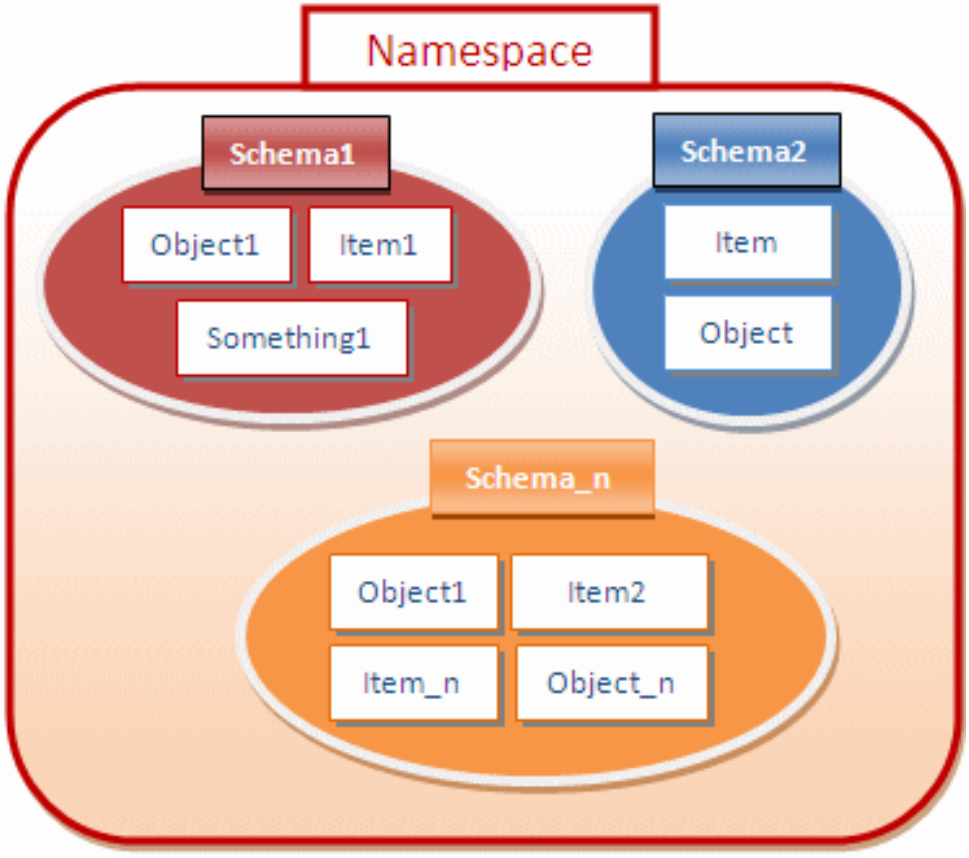


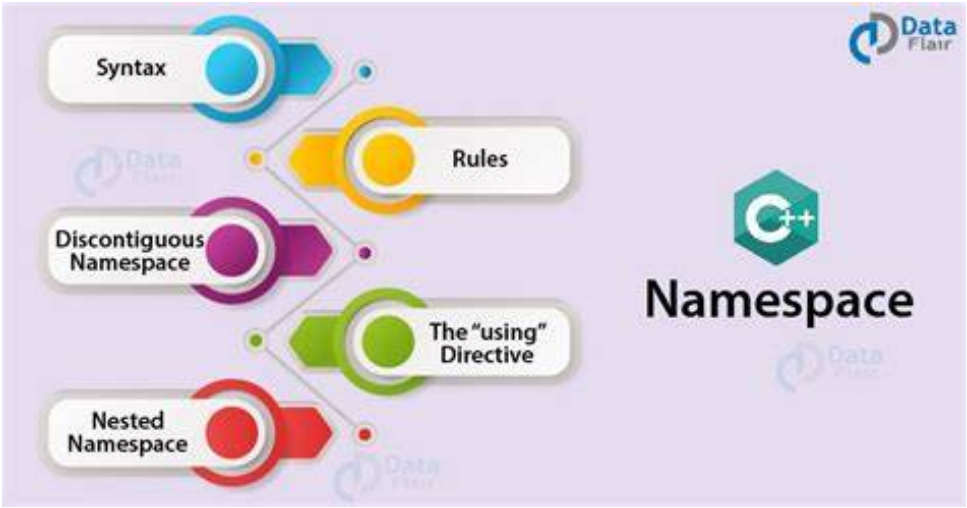
Preprocessors + Compiler + Assembler + Linker + Loader


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)
09-06-2015
3


source program
compiler or assembler
compile time
object module
other object modules
linkage editor
load module
load time
system library
loader
dynamically loaded system library
dynamic linking
in-memory binary memory image
executior time (run time)

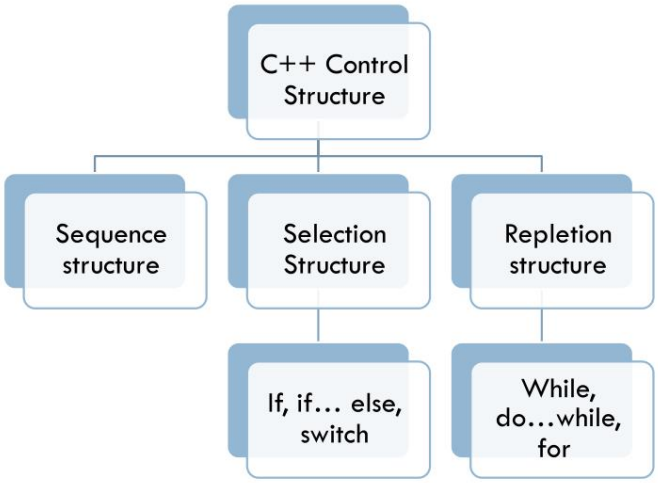
Namespace





Selection and Repetition

Control structures in C++



3. Control Structures

Bohm, Jacopini (1966) showed that only the three following control structures are needed for representing an algorithm:

Sequence

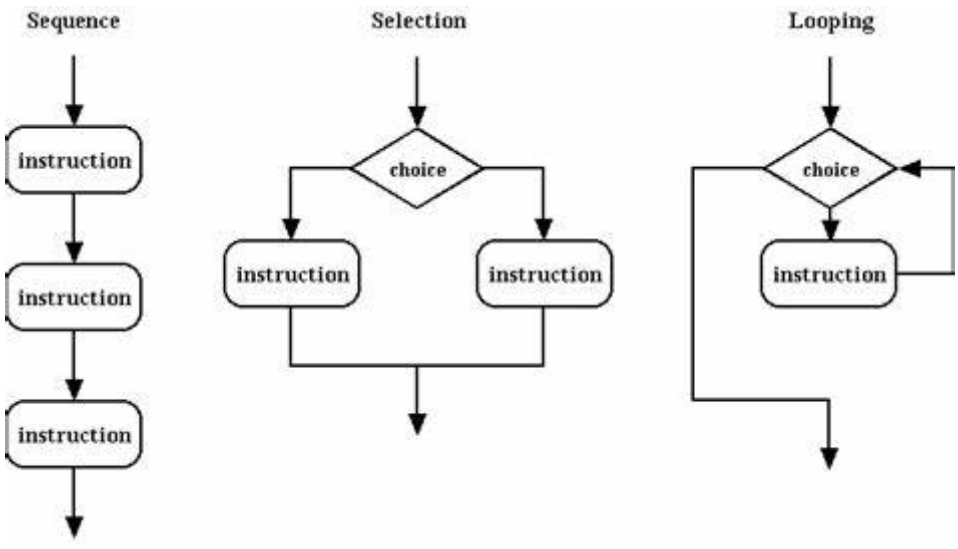
2.1. Selection
Choosing one of several courses of action according to some condition

2.2. Repetition
Repetition of actions or sequences of actions according to some condition

Syntax Constructs:

built in	**if** **if/else** **? :** **switch**	**while** **for** **do/while**

09/15/08 MET CS 563 -Fall 2008 3. Control Structures 1



Selection

- A Selection control structure is used to choose among alternative courses of action.
- There must be some *condition* that determines whether or not an action occurs.
- C++ has a number of selection control structures:
  - If
  - if/else
  - switch

Repetition

- Carries out a particular action over and over again until the condition is met.
- A loop is created to return the program to where the repetition has started for as long as it takes until the condition is met.
- There are two ways of testing to see if the end condition is met: 1) pre-test loops 2) post-test loops.
- In pre-test loops the condition is tested at the start. If the condition is false the first time, the processes will never carry out. Pre-test loops end when the condition Is false. Uses WHILE ... ENDWHILE.
- Counted loops are special types of pre-test loops. They are used when a known number of repetitions will occur.
- Pre-test loops are also known as guarded loops because the loop is only operated when the condition is met.

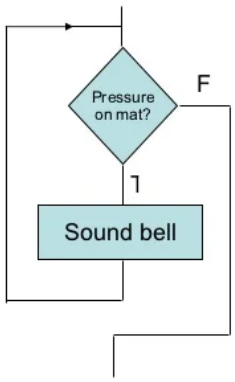
Pseudocode

WHILE there is pressure on the mat

Sound the bell

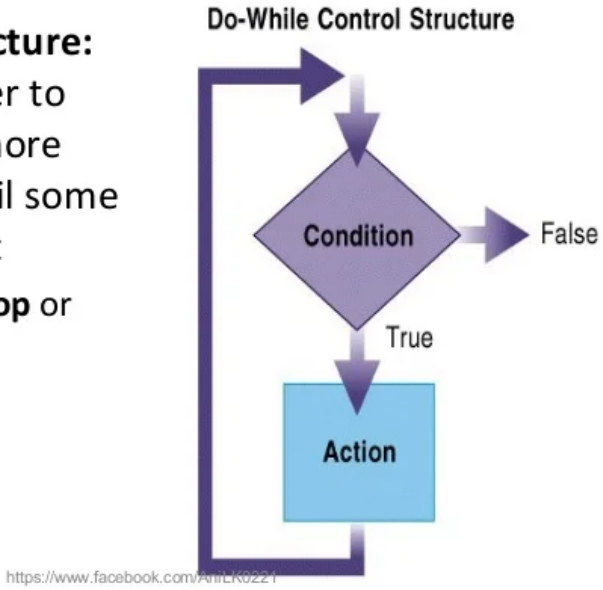
ENDWHILE

Flowchart



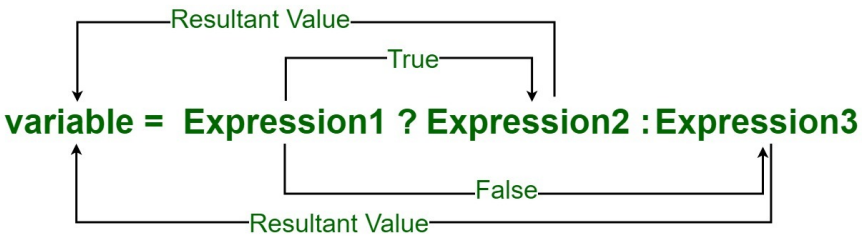
Repetition (Loop)

- **Repetition structure:** directs computer to repeat one or more instructions until some condition is met
  - Also called a **loop** or **iteration**

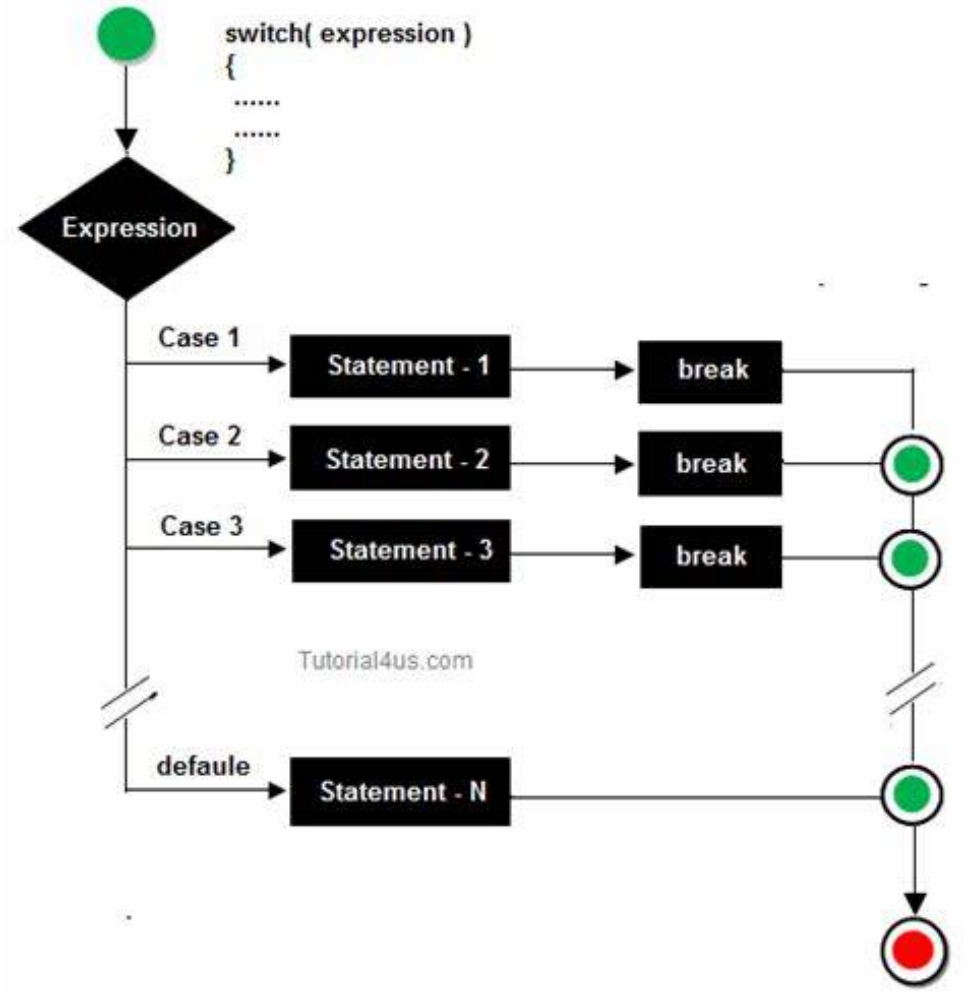


Ternary If

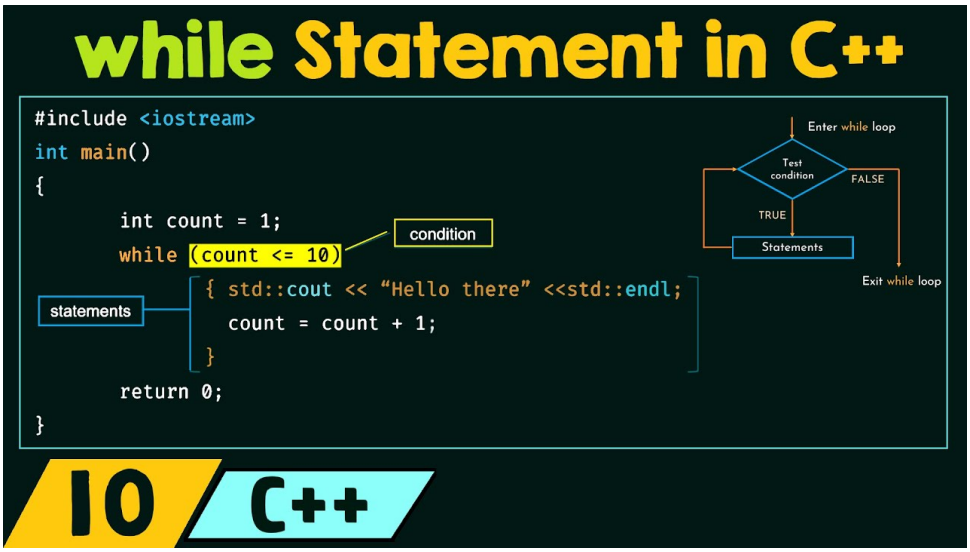
Conditional or Ternary Operator (?:) in C/C++



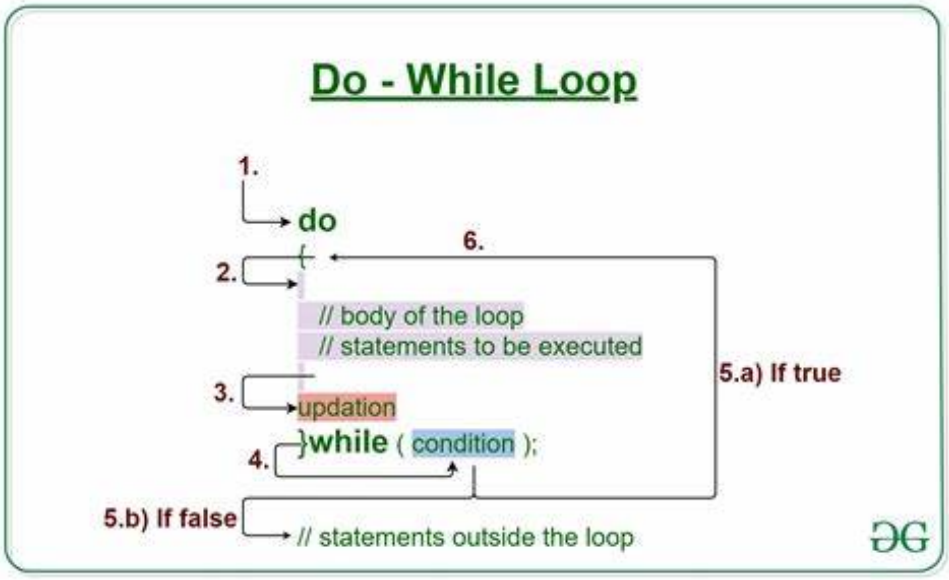
Switch Case



While



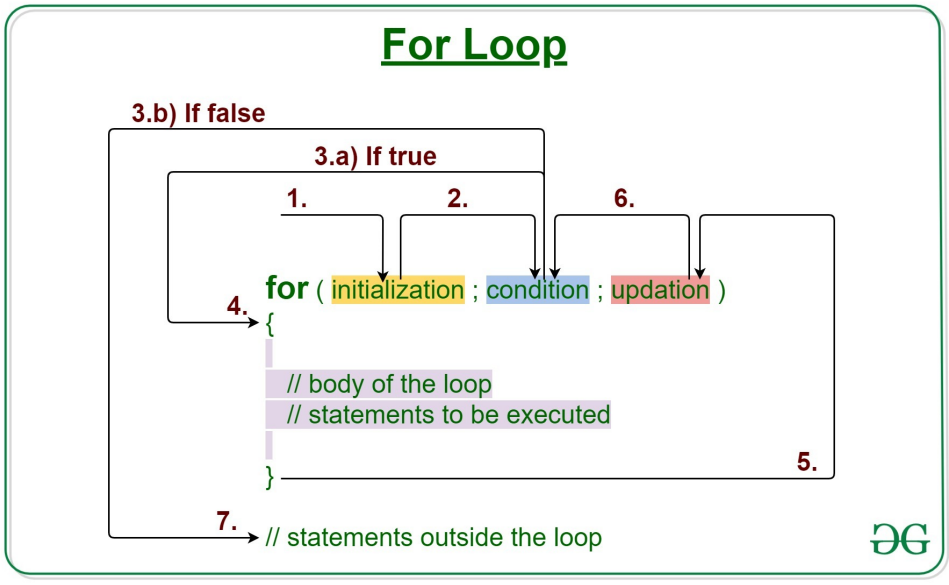
Do-While



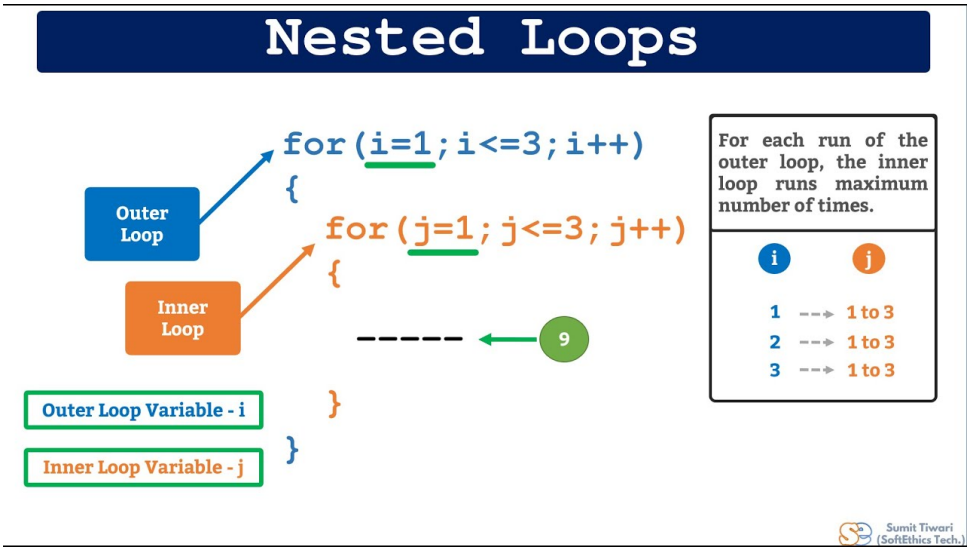
Increment and Decrement Operators

OPERATOR	MEANING
++a	Increment a by 1, then use new value of a
a++	Use value of a, then increment a by 1
--b	Decrement a by 1, then use new value of a
b--	Use value of a, then decrement a by 1

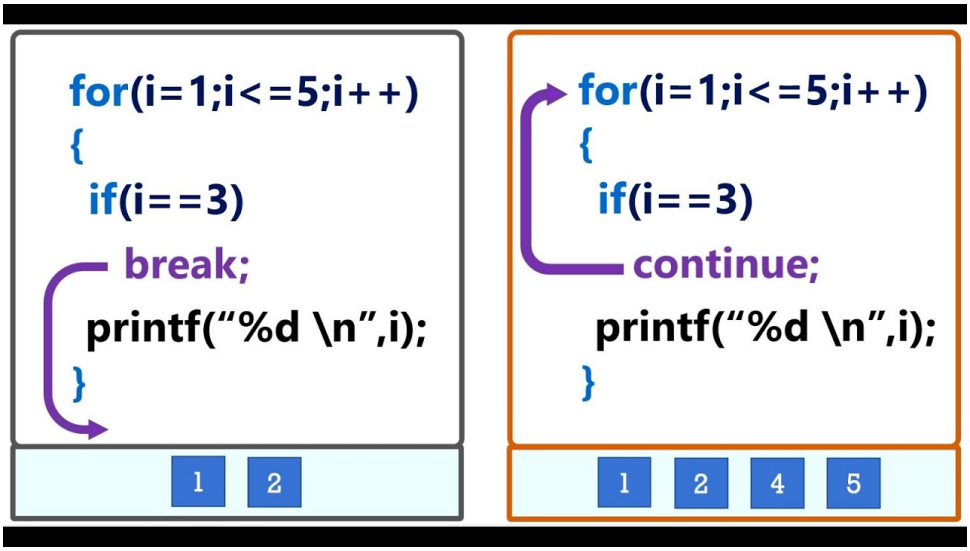
For Loop



Nested Loop



Break and Continue



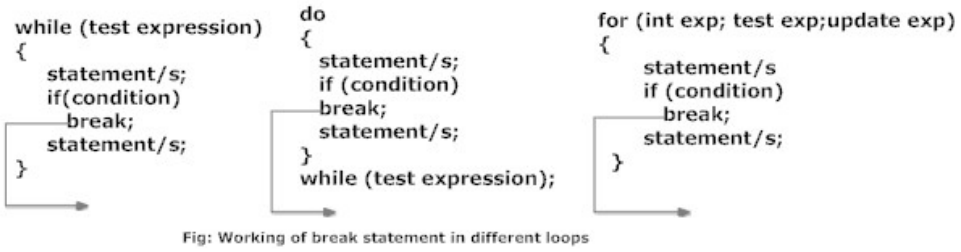


Fig: Working of break statement in different loops

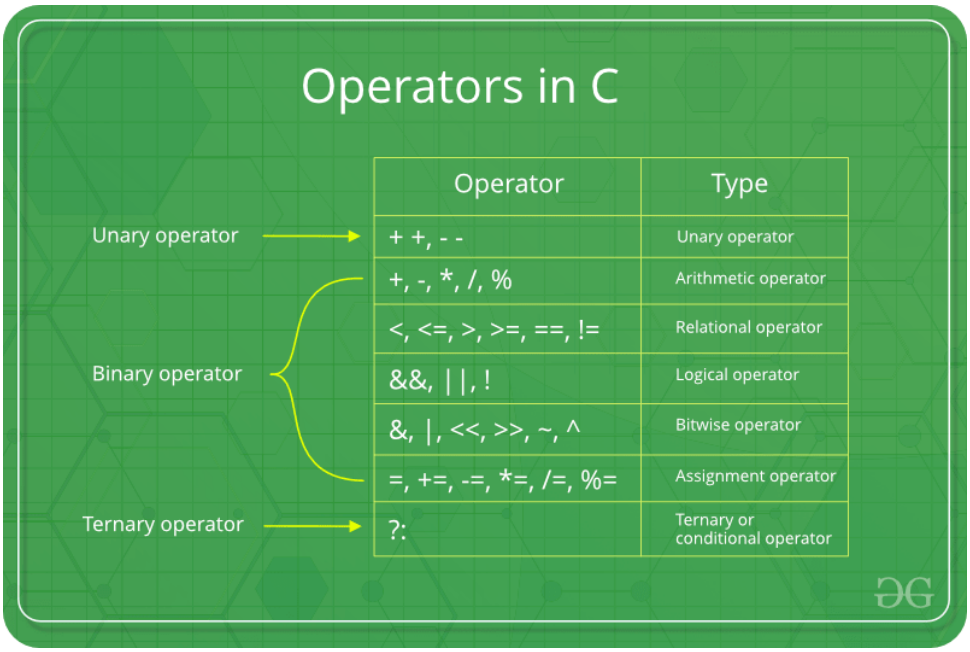
```
while (test expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statements/
}
```

NOTE: The continue statement may also be used inside body of else statement.

Operations in C

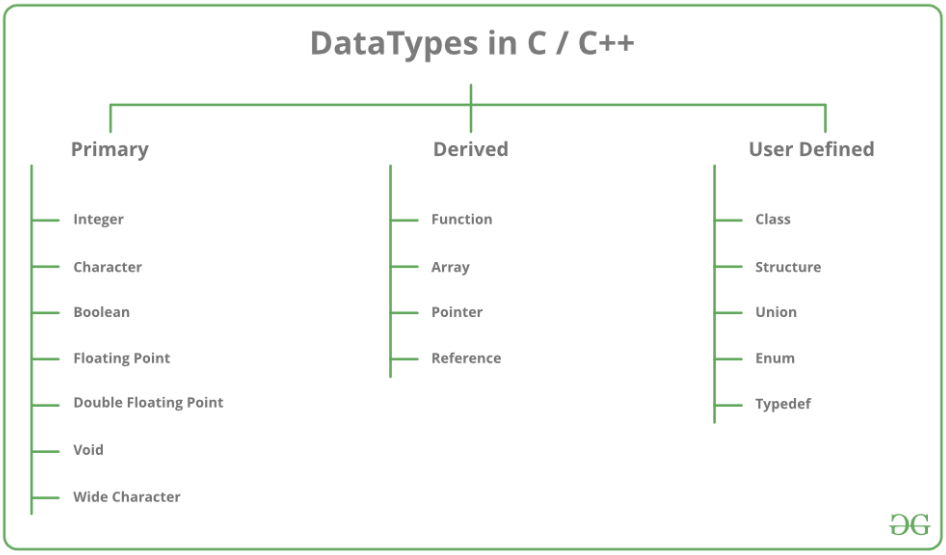


Priority of Operations

	Operator Precedence
1	! Logical not (Highest)
2	( ) Parenthesis
3	*, /, %
4	+, -
5	>, >=, <, <=
6	==, !=
7	&& (AND)
8	|| (OR)
9	= (Lowest)

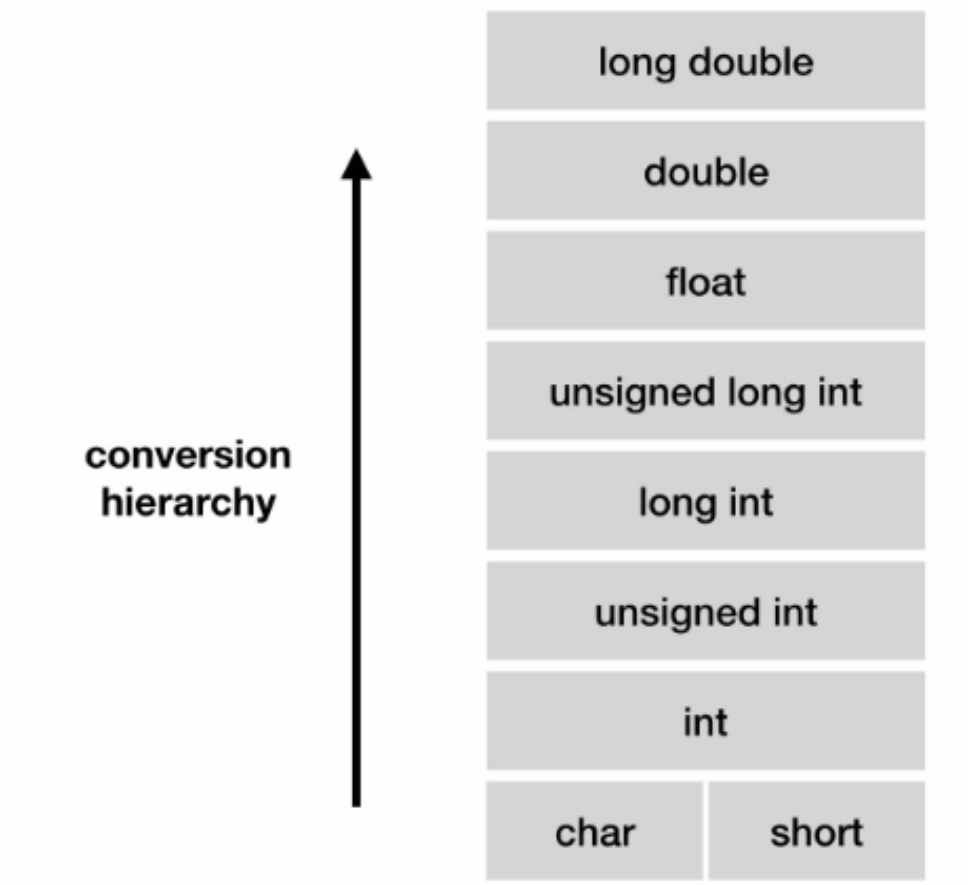
Signs of operations	Name of operation, explanation	Associativity
`() [] . ->`	Primary	From left to right
`+ - ~ ! * & ++ --` `sizeof(type)` `(type cast)`	Unary	From right to left
`* / %`	Multiplicative, arithmetical, binary	From left to right
`+ -`	Additive, arithmetical, binary	From left to right
`>> <<`	Shift	From left to right
`< > <= >=`	Relation	From left to right
`== !=`	Relation	From left to right
`&`	Bitwise "AND", logical, binary	From left to right
`^`	Bitwise XOR, logical, binary	From left to right
`|`	Bitwise logical "OR", logical, binary	From left to right
`&&`	Logical "AND", binary	From left to right
`||`	Logical "OR", binary	From left to right
`?:`	Conditional, ternary	From right to left
`= *= /= %= += -=` `<<= >>= &= |= ^=`	Simple and complex assignment	From right to left
`,`	Sequential computation	From left to right

Data Type



Key word	Size in bytes	Interpretation	Possible values
bool	1	boolean	true and false
unsigned char	1	Unsigned character	0 to 255
char (or signed char)	1	Signed character	-128 to 127
wchar_t	2	Wide character (in windows, same as unsigned short)	0 to 2^{16}-1
short (or signed short)	2	Signed integer	-2^{15} to 2^{15} - 1
unsigned short	2	Unsigned short integer	0 to 2^{16}-1
int (or signed int)	4	Signed integer	-2^{31} to 2^{31}-1
unsigned int	4	Unsigned integer	0 to 2^{32} - 1
Long (or long int or signed long)	4	signed long integer	-2^{31} to 2^{31}-1
unsigned long	4	unsigned long integer	0 to 2^{32} -1
float	4	Signed single precision floating point (23 bits of significand, 8 bits of exponent, and 1 sign bit.)	$3.4*10^{-38}$ to $3.4*10^{38}$ (both positive and negative)
long long	8	Signed long long integer	-2^{63} to 2^{63} -1
unsigned long long	8	Unsigned long long integer	0 to 2^{64} -1
double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	$1.7*10^{-308}$ to $1.7*10^{308}$ (both positive and negative)
long double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	$1.7*10^{-308}$ to $1.7*10^{308}$ (both positive and negative)

Casting



Introduction to C++

C++ Casting Operators

- **static_cast<type> (expr):**
- **static_cast operator** is used to convert a given expression to the specified type. For example, it can be used to cast a base class pointer into a derived class pointer.

```
int main() {
    int a = 31;
    int b = 3;
    float x = a/b;
    float y = static_cast<float>(a)/b;
    cout << "Output without static_cast = " << x << endl;
    cout << "Output with static_cast = " << y << endl;
}
```

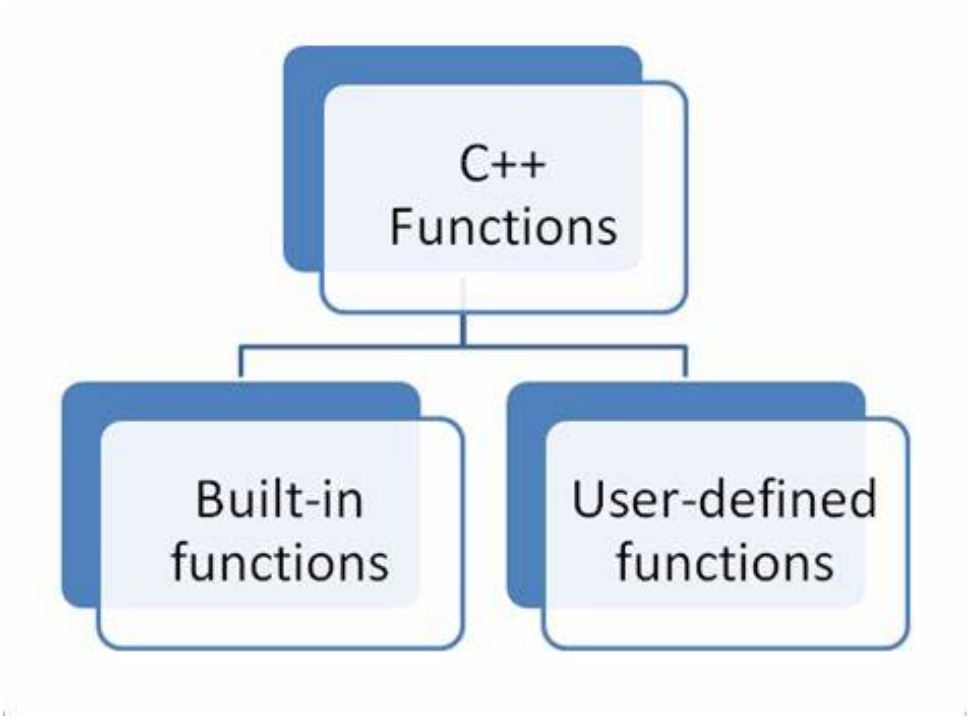
In this type casting example, using the static_cast to an integer as "float" returns a float value.

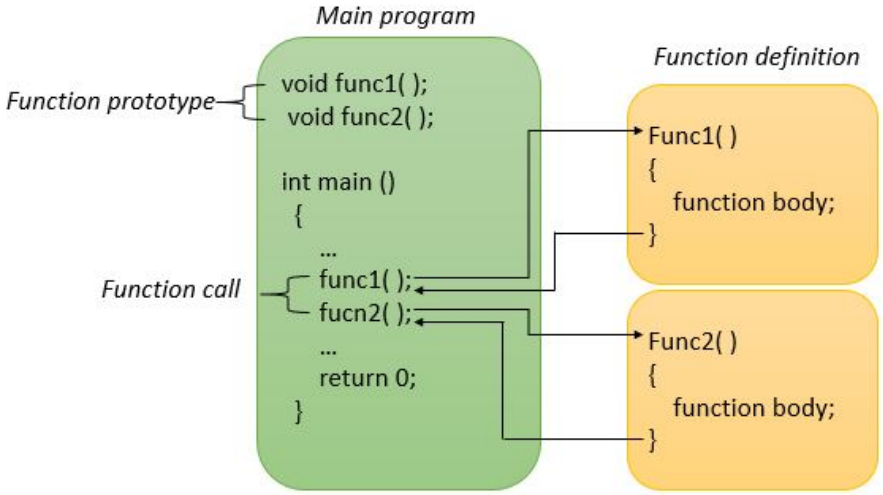
Lecture Slides By Adil Aslam

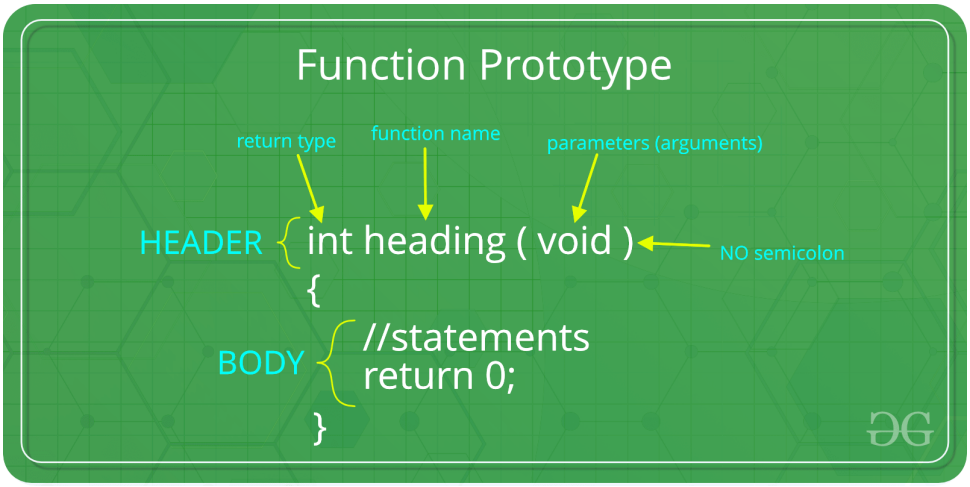
Explicit type conversion

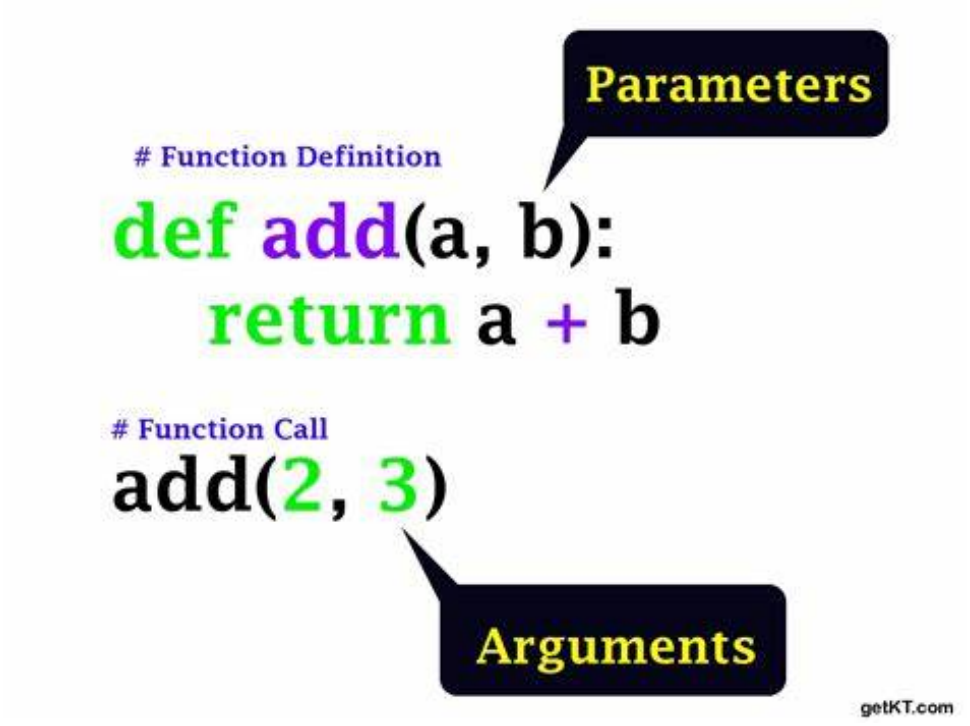
- **C++ casts**
  - **`static_cast` between 2 related types (int/float, int/enum, 2 pointers in class hierarchy)**
  - **`reinterpret_cast` between 2 unrelated types (int/ptr, pointers to 2 unrelated classes)**
  - **`const_cast` cast away constness**
  - **`dynamic_cast` used for polymorphic types Run-time type info (RTTI)**
- **Avoid casts, but use these instead of C casts**
  - **e.g., compiler can perform minimal checking for `static_cast`, none for `reinterpret_cast`**

Function

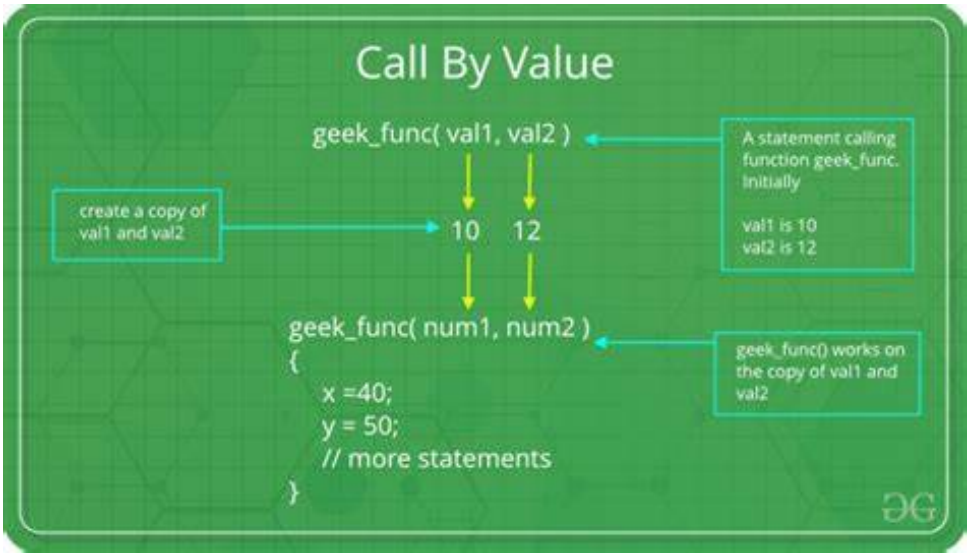


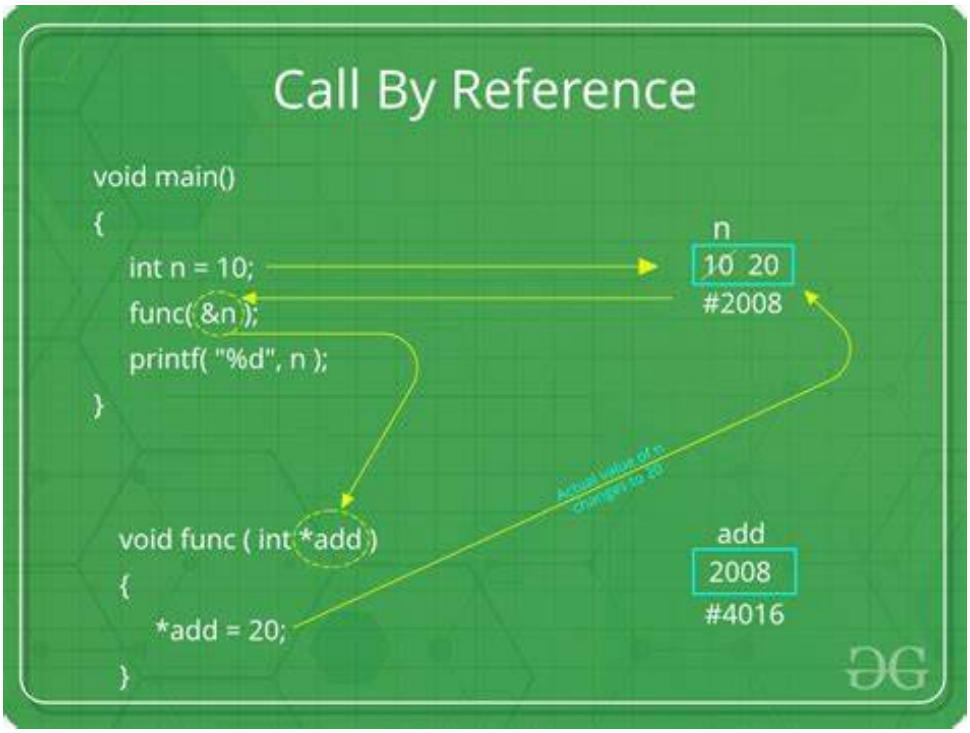






Call by Value and Call by Reference





Programming in C –Call By

CALL BY VALUE

The call to the function passes either the values or the normal variables to the called function. The actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

```
void swap(int a, int b)
{
 int t=a; a=b; b=t;
 }
void main( )
{
 int a=5, b=10 ;
 printf("Before swap: a=%d,b=%d",a,b);
 swap(a,b); /*calling swap function*/
 printf("After swap: a=%d,b=%d",a,b);
}
```

Output:
Before swap: a=5,b=10
After swap: a=5,b=10
Because variable declared 'a', 'b' in main() is different from variable 'a', 'b' in swap(). Only variable names are similar but their memory address are different and stored in different memory locations.

CALL BY REFERENCE

The call to the function passes base address to the called function. The actual arguments are not copied to the formal arguments, only referencing is made. Hence any operation performed by function on formal arguments / parameters affects actual parameters.

```
void change(int b[ ])
{
 b[0]=10; b[1]=20; b[2]=30;
 }
void main( )
{
 int a[3]={ 5, 15, 25 } ;
printf("BeforeChange:%d,%d,%d",a[0],a[1],a[2]);
 change(a); /*calling swap function*/
printf("AfterChange:%d,%d,%d",a[0],a[1],a[2]);
}
```

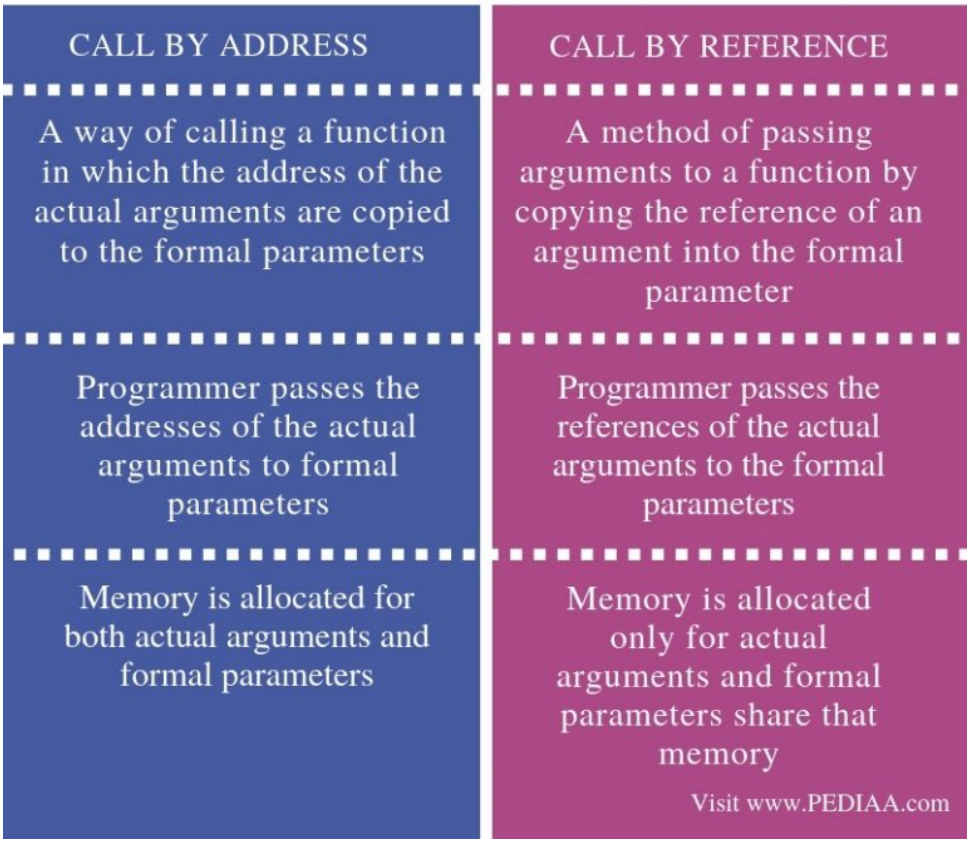
Output:
BeforeChange: 5,15,25
AfterChange: 10,20,30
Because array variable declared 'b' in change() is referencing/ pointing to array variable 'a' in main(). Only variable name is different but both are pointing / referencing to same memory address locations.

CALL BY ADDRESS

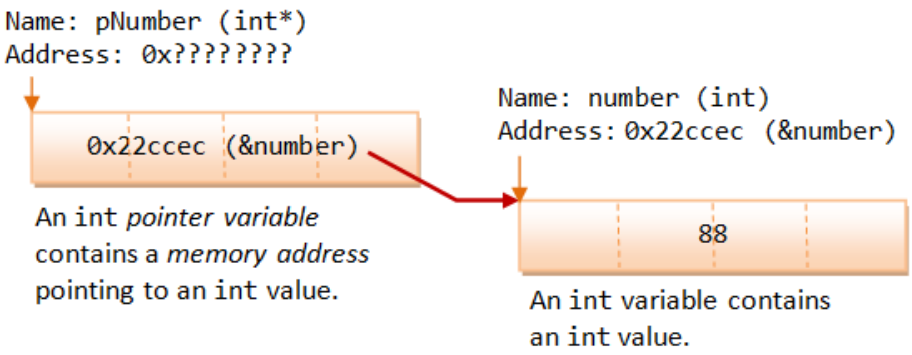
The call to the function passes variable's address to the called function. The actual arguments are not copied to the formal arguments, the addresses of actual arguments (or parameters) are passed to the formal parameters. Hence any operation performed by function on formal arguments / parameters affects actual parameters.

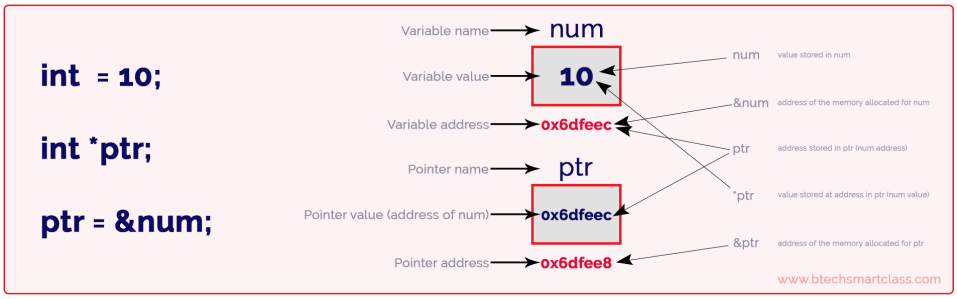
```
void swap(int *x, int *y)
{ int t=*x; *x=*y; *y=t;
 }
void main( ) {
 int a=5, b=10 ;
 printf("Before swap: a=%d,b=%d",a,b);
 swap(&a,&b); /*calling swap function*/
 printf("After swap: a= %d,b=%d",a,b);
}
```

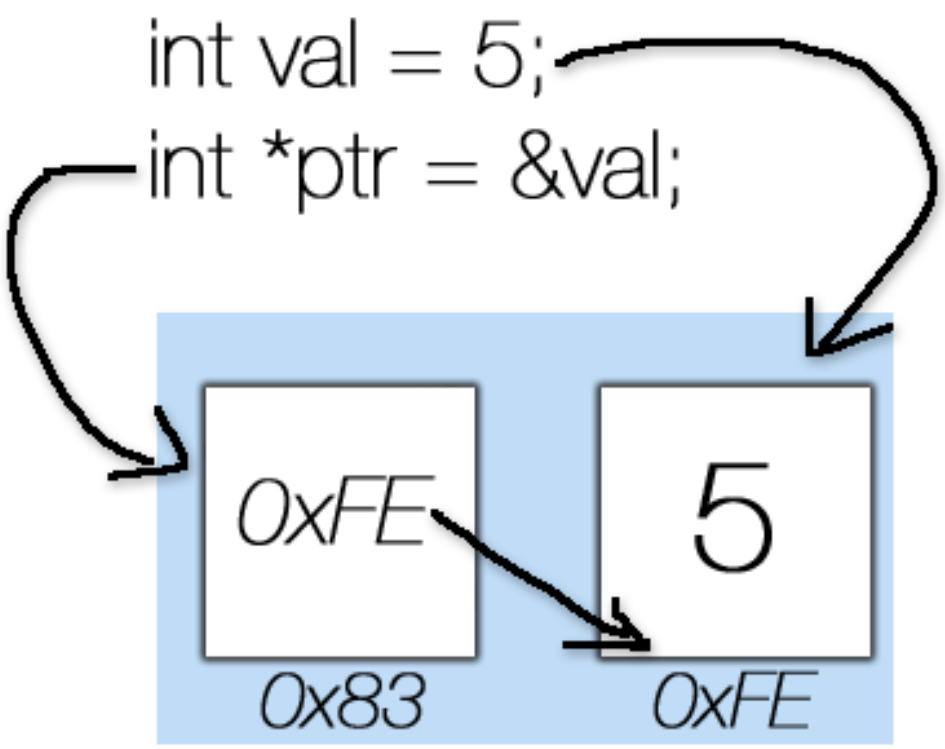
Output:
Before swap: a=5,b=10
After swap: a=10,b=5
Because variable declared 'a', 'b' in main() is different from variable 'x', 'y' in swap(). Only variable names are different but both a and x, b and y point to the same memory address locations

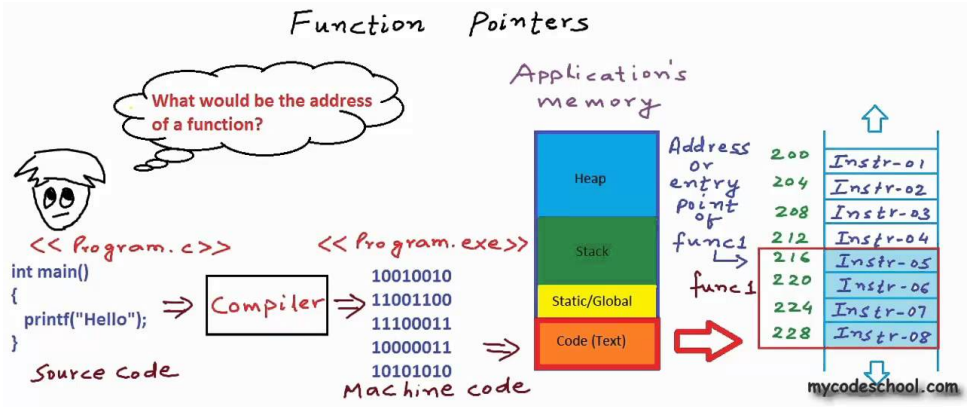

CALL BY ADDRESS
VERSUS
CALL BY REFERENCE
CALL BY ADDRESS
A way of calling a function in which the address of the actual arguments are copied to the formal parameters
Programmer passes the addresses of the actual arguments to formal parameters
Memory is allocated for both actual arguments and formal parameters
CALL BY REFERENCE
A method of passing arguments to a function by copying the reference of an argument into the formal parameter
Programmer passes the references of the actual arguments to the formal parameters
Memory is allocated only for actual arguments and formal parameters share that memory
Visit www.PEDIAA.com

Pointer


Name: pNumber (int*)
Address: 0x????????
0x22ccec (&number)
An int pointer variable contains a memory address pointing to an int value.
Name: number (int)
Address: 0x22ccec (&number)
88
An int variable contains an int value.


int = 10;
int *ptr;
ptr = #
Variable name
num
Variable value
10
Variable address
0x6dfeec
Pointer name
ptr
Pointer value (address of num)
0x6dfeec
Pointer address
0x6dfee8
num
value stored in num
&num
address of the memory allocated for num
ptr
address stored in ptr (num address)
*ptr
value stored at address in ptr (num value)
&ptr
address of the memory allocated for ptr
www.btechsmartclass.com





Address Operation

Introduction to C++

C++ Address of Operator (&)

- The & is a unary operator means it requires only one operand.
- The Address of Operator returns the memory address of its operand.
- Address of operator has the same precedence and right-to-left associativity as that of other unary operators.
- Symbol for the bitwise AND and the address of operator are the same but they don't have any connection between them.

Lecture Slides By Adil Aslam

Introduction to C++

C++ Indirection/ Deference Operator *

- Getting the address of a variable isn't very useful by itself.
- The dereference operator (*) allows us to get the value at a particular address:
- The Indirection operator * is a unary operator means it requires only one operand.
- Indirection Operator (*) is the complement of Address of Operator (&).
- Indirection Operator returns the value of the variable located at the address specified by its operand.

Lecture Slides By Adil Aslam

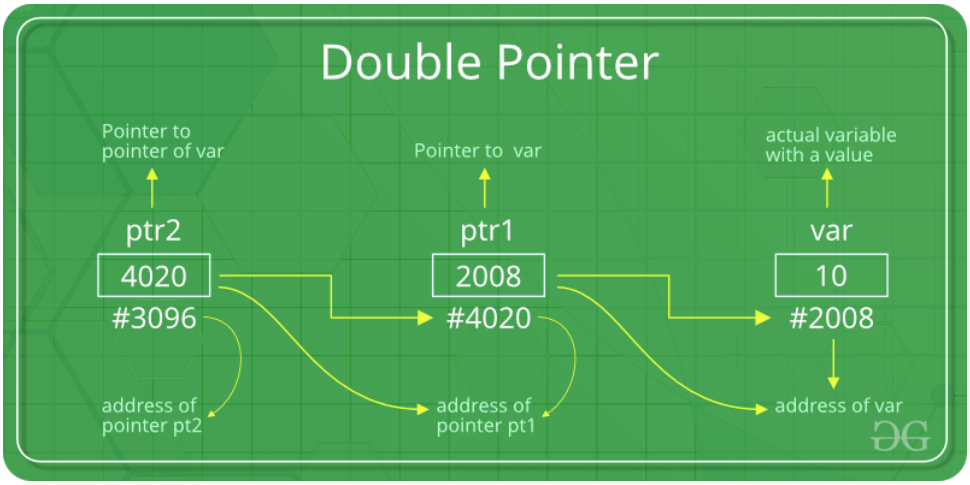
Introduction to C++

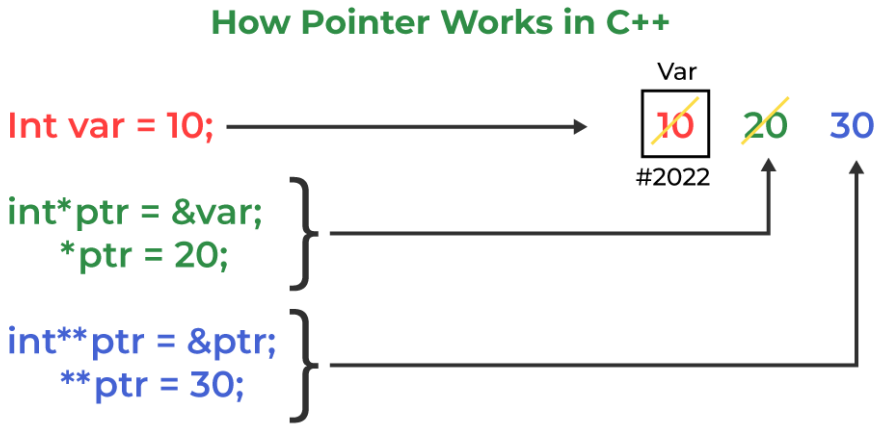
Reference Operator (&) and Deference Operator (*)

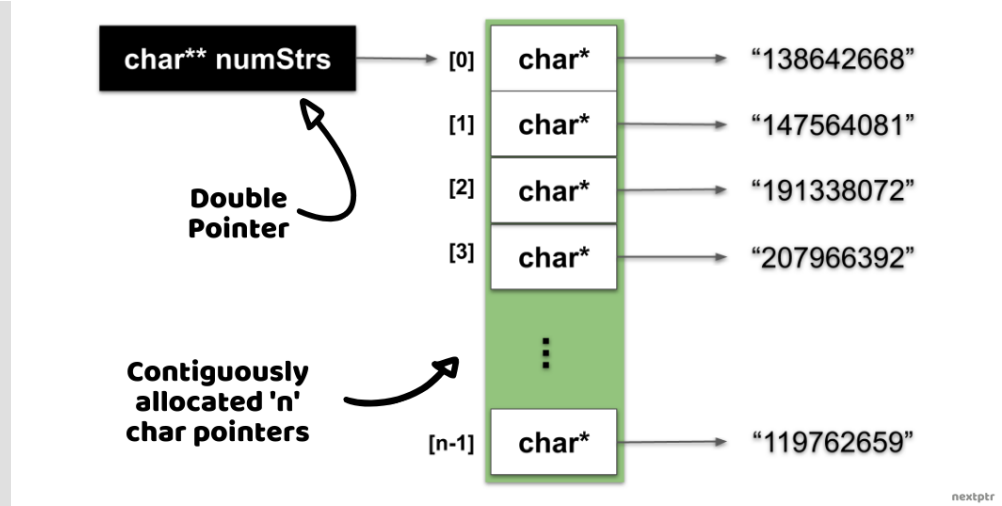
- Reference operator (&) as already discussed, gives the address of a variable.
- To get the value stored in the memory address, we use the dereference operator (*).
- **For Example:** If a number variable is stored in the memory address 0x123, and it contains a value 5.
- The reference (&) operator gives the value 0x123, while the dereference (*) operator gives the value 5.
- **Note:** The (*) sign used in the declaration of C++ pointer is not the dereference pointer. It is just a similar notation that creates a pointer.

Lecture Slides By Adil Aslam

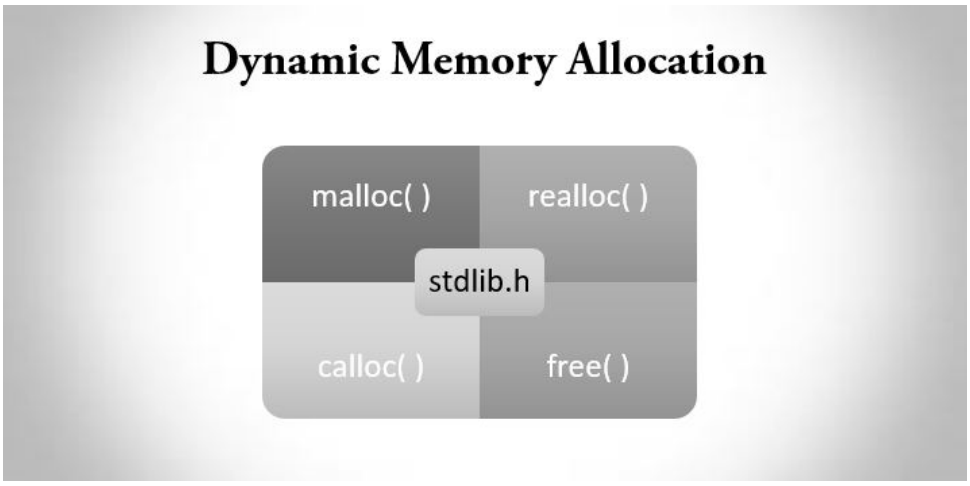
Pointer to Pointer

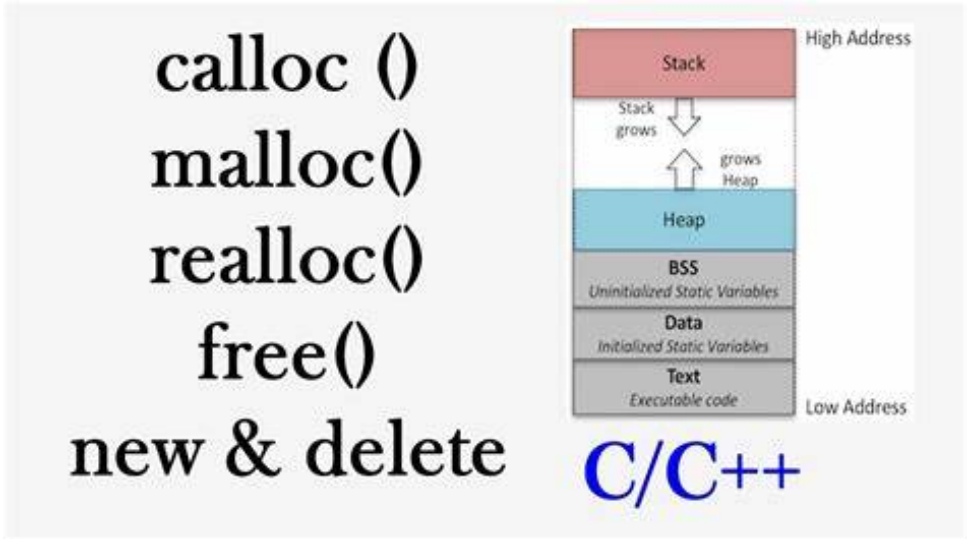






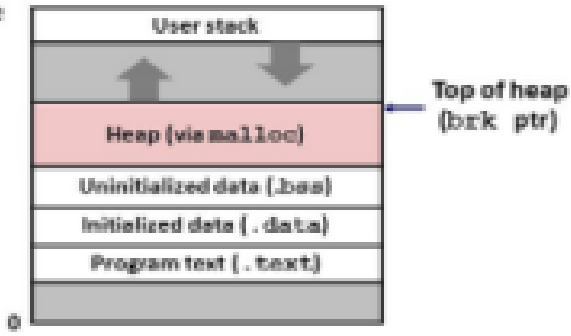
DMA(Dynamic Memory Allocation)





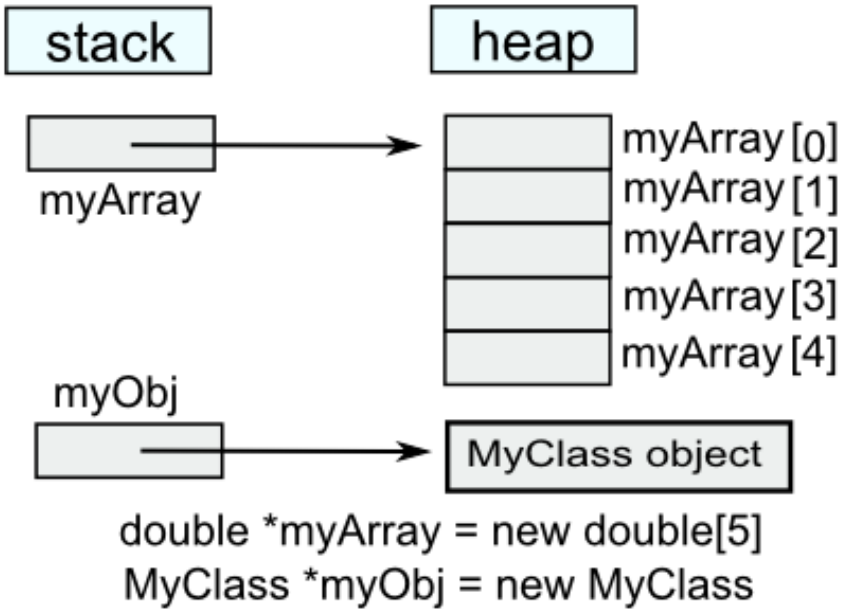
Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
  - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.

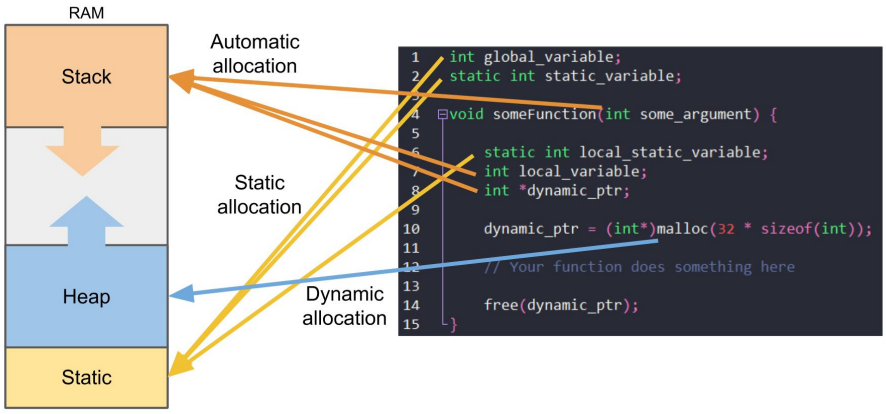


DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

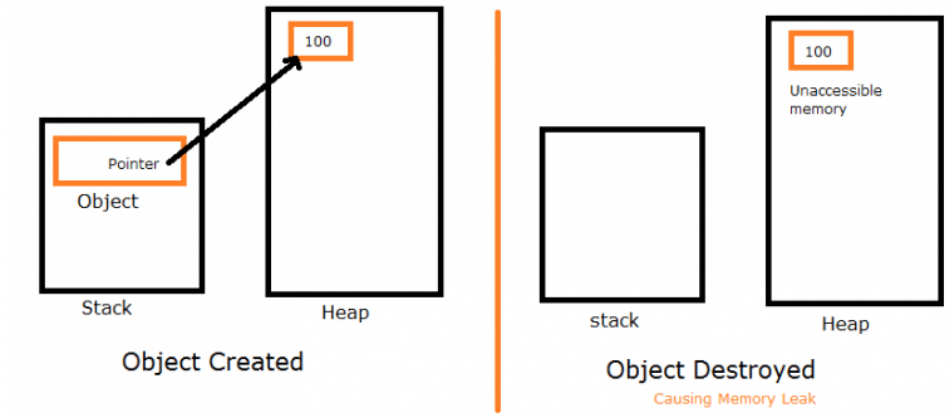
malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));



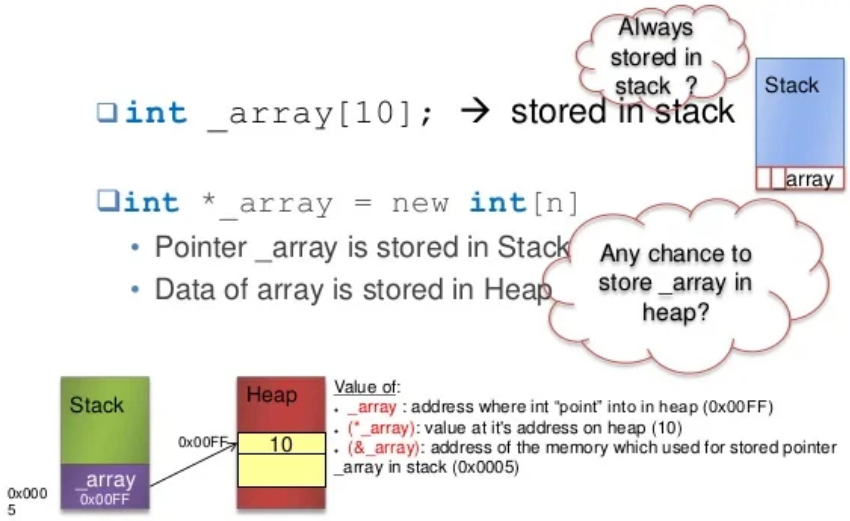
Memory Allocation



Memory Leak

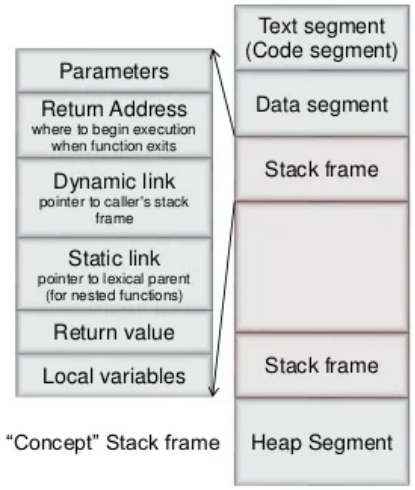


Heap vs Stack



8

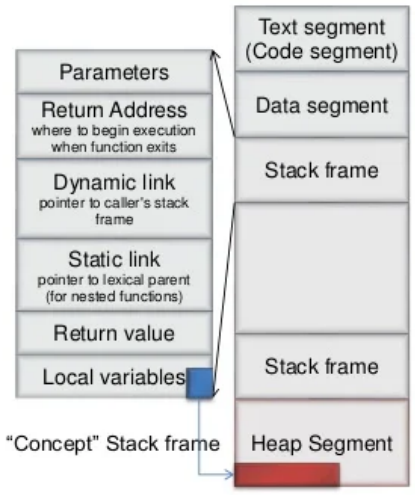
Stack



- Where parameters and local variables are allocated
- Limited size → Stack overflow
- Memory use in stack is temporary and auto release
- Fast processing/low size

6

Heap



- Large pool of memory
- Dynamic allocation
- Stays allocated until specifically deallocated → leak !
- Must be accessed through a pointer
- Large arrays, structures, or classes should be stored Heap → why?
- Large & dynamic

7

Basics of pointers | Using pointers in functions
Dynamic memory allocation (DMA) | Arrays and pointer arithmetic

Releasing space manually

- The **delete** operator will release a dynamically-allocated space.

```
int* a = new int;
delete a; // release 4 bytes
int* b = new int[5];
delete b; // release only 4 bytes!
          // Unpredictable results may happen
delete [] b; // release all 20 bytes
```

- The **delete** operator will do nothing to the pointer. To avoid reusing the released space, set the pointer to **nullptr**.

```
int* a = new int;
delete a;    // a is still pointing to the address
a = nullptr; // now a points to nothing
int* b = new int[5];
delete [] b; // b is still pointing to the address
b = nullptr; // now b points to nothing
```

Programming Design – Pointers 58 / 87 Ling-Chieh Kung (NTU IM)

Basics of pointers | Using pointers in functions
Dynamic memory allocation (DMA) | Arrays and pointer arithmetic

Good programming style

- Use DMA for arrays with **no predetermined** length.
  - Even though Dev-C++ (and some other compilers) converts

```
int a = 10;
int b[a];
```

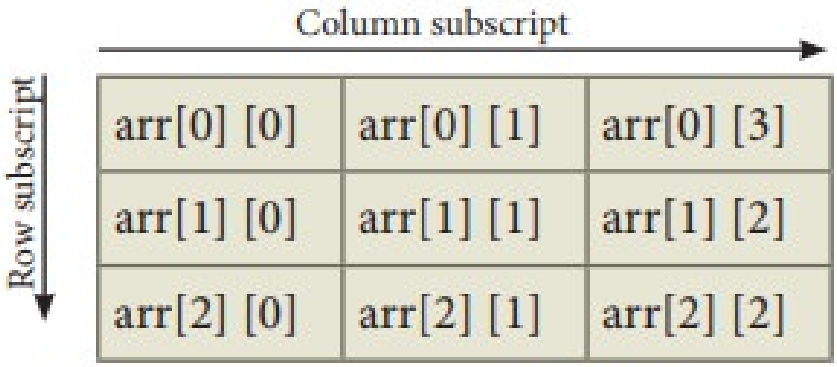
to

```
int a = 10;
int* b = new int[a];
// ...
delete [] b;
```

- To avoid memory leak:
  - Whenever you write a **new** statement, add a **delete** statement below immediately (unless you know you really do not need it).
  - Whenever you want to change the value of a pointer, check whether memory leak occurs.
  - Whenever you write a **delete** statement, set the pointer to **nullptr**.

Programming Design – Pointers 59 / 87 Ling-Chieh Kung (NTU IM)

Array



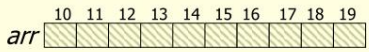
The array arr can be coneptually viewed in matrix form with 3 rows and coloumns. point to be noted here is since the subscript starts with 0 arr [0][0] represents the first element.

Figure 12.4

Array Declaration

- An *array* is a collection of objects with same type stored consecutively in memory
- Declaring an array

```
IntCell arr[10]; //an array consisting of 10 IntCell
objects
```



  - The size of the array must be known at compile time.
  - *arr* actually is a constant pointer. The value of *arr* **cannot** be changed.

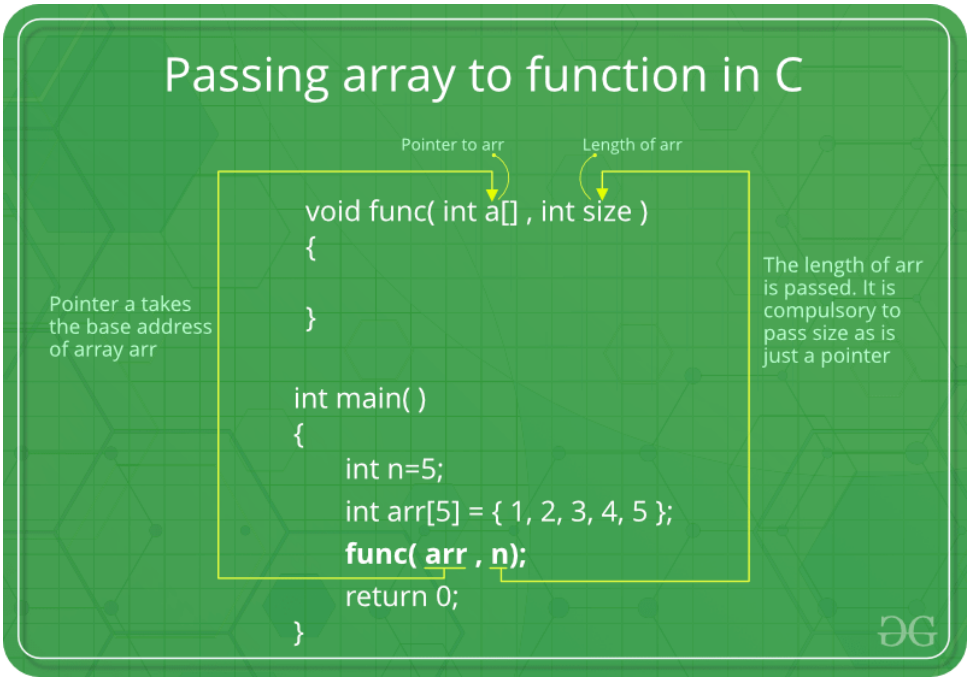
```
IntCell * p = new IntCell[10];
arr = p; // invalid
```

  - The *(i+1)*-st object in the array *arr* can be accessed either by using *arr[i],* or by **(arr+i).*
- There is no index range checking for arrays in C++
- Cannot be copied with =
- Arrays are *not* passed by copy. Instead, the address of the first element is passed to the function

```
int sumOfArray( int values[], int numValues )
```

21

Passing Array to Function

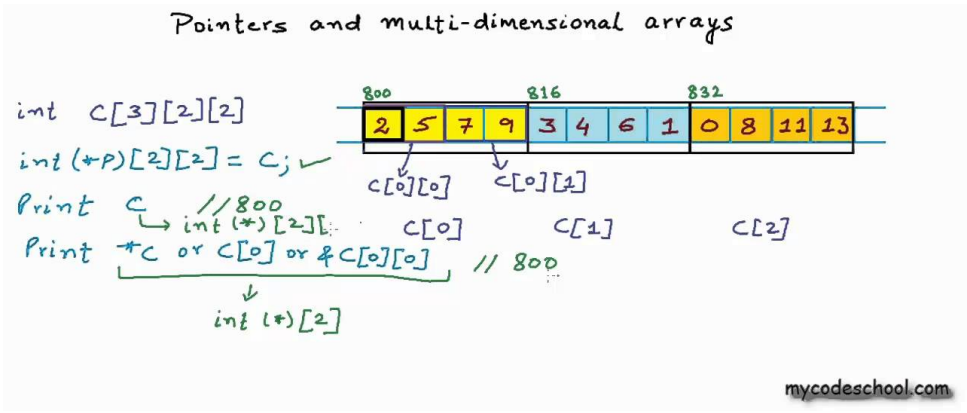


Container Library Array

language built-in array	container library array

`#include <iostream>`
 	
`using namespace std;`
 	
`int main()`
`{`
`  int myarray[3] = {10,20,30};`
 	
`  for (int i=0; i<3; ++i)`
`  ++myarray[i];`
 	
`  for (int elem : myarray)`
`  cout << elem << '\n';`
`}`
	`#include <iostream>`
	`#include <array>`
	`using namespace std;`
	
	`int main()`
	`{`
	`  array<int,3> myarray {10,20,30};`
	
	`  for (int i=0; i<myarray.size(); ++i)`
	`  ++myarray[i];`
	
	`  for (int elem : myarray)`
	`  cout << elem << '\n';`
	`}`

Multi-Dimesion Array



```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```

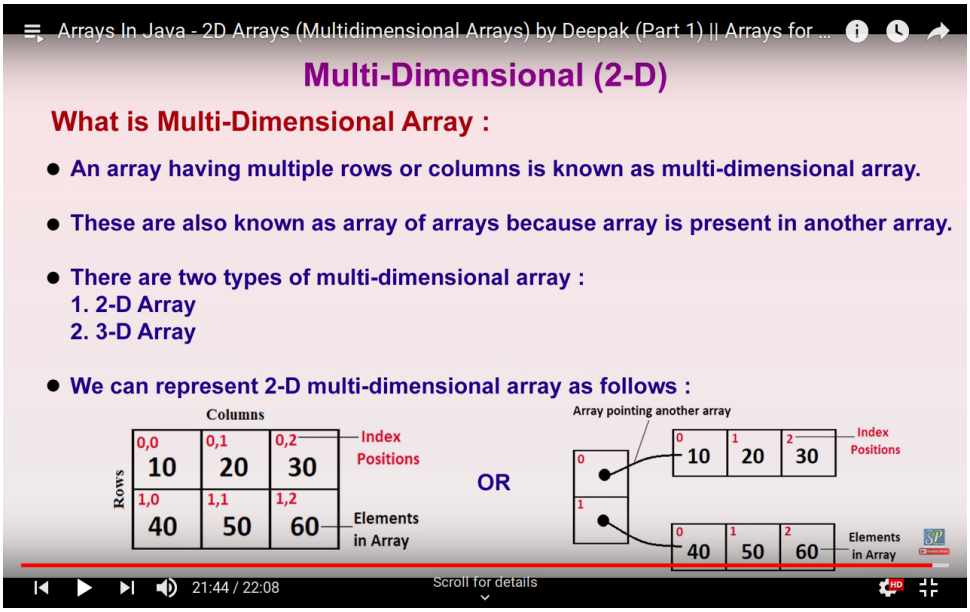
CLASSROOM

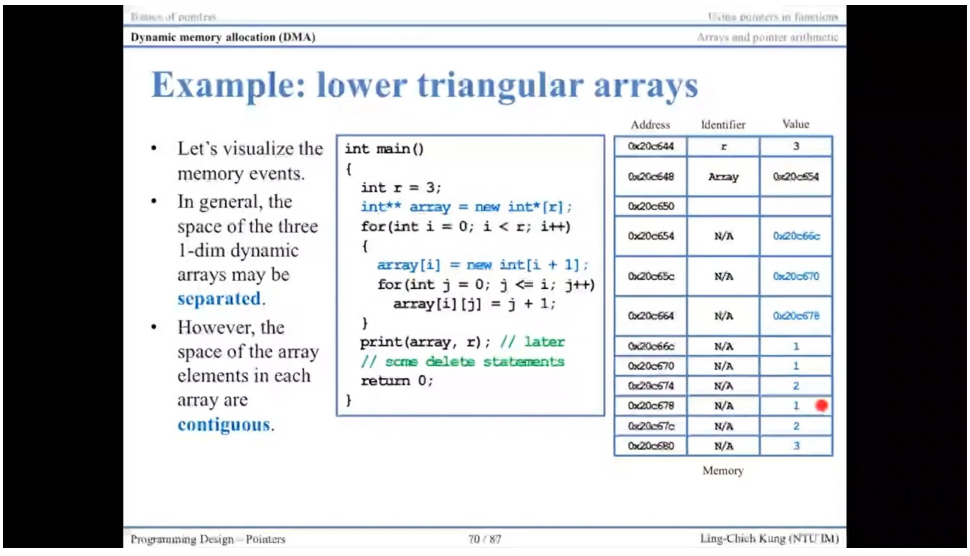
row-wise memory allocation

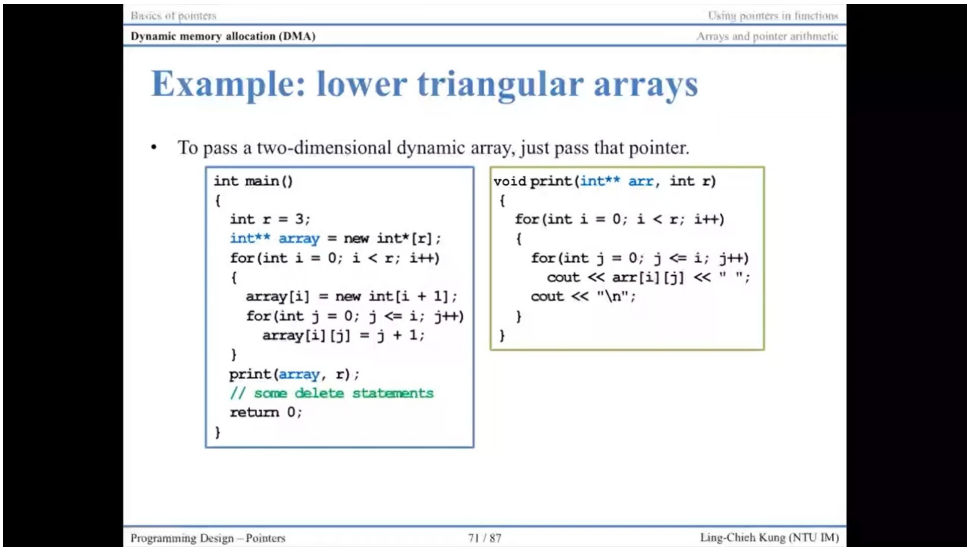
	<—— row 0 ——>				<—— row 1 ——>				<—— row 2 ——>			
value	1	2	3	4	5	6	7	8	9	10	11	12
address	1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1022

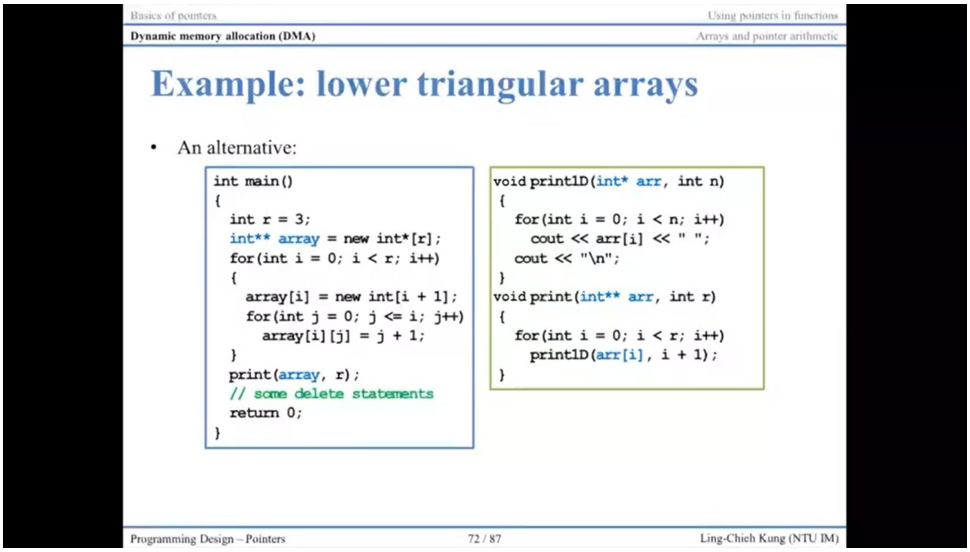
first element of the array num

dyclassroom.com

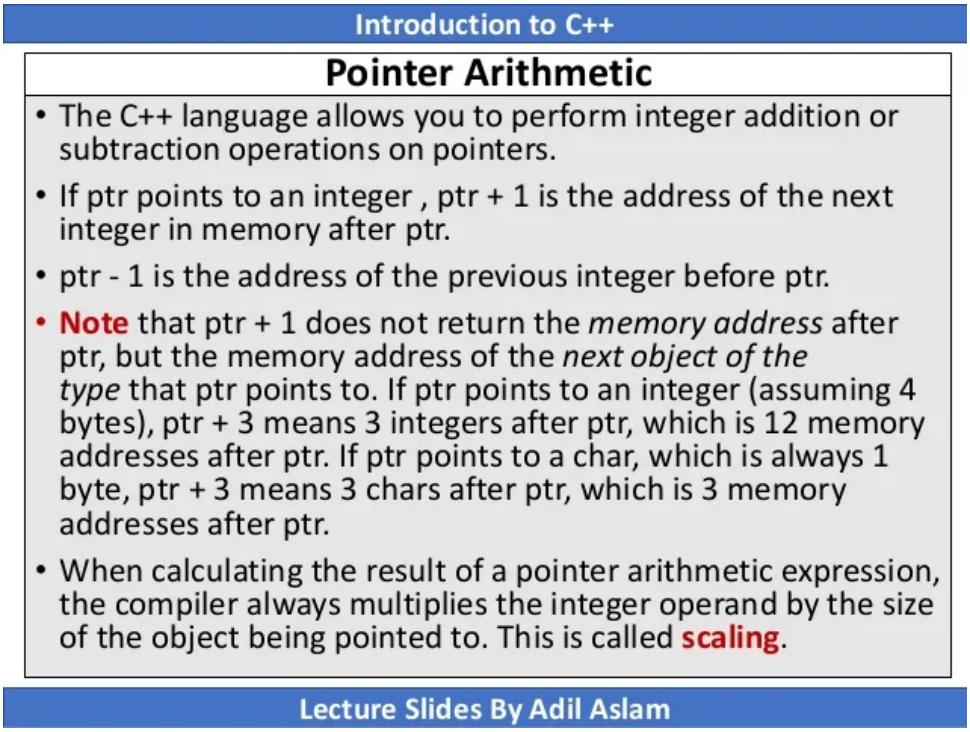






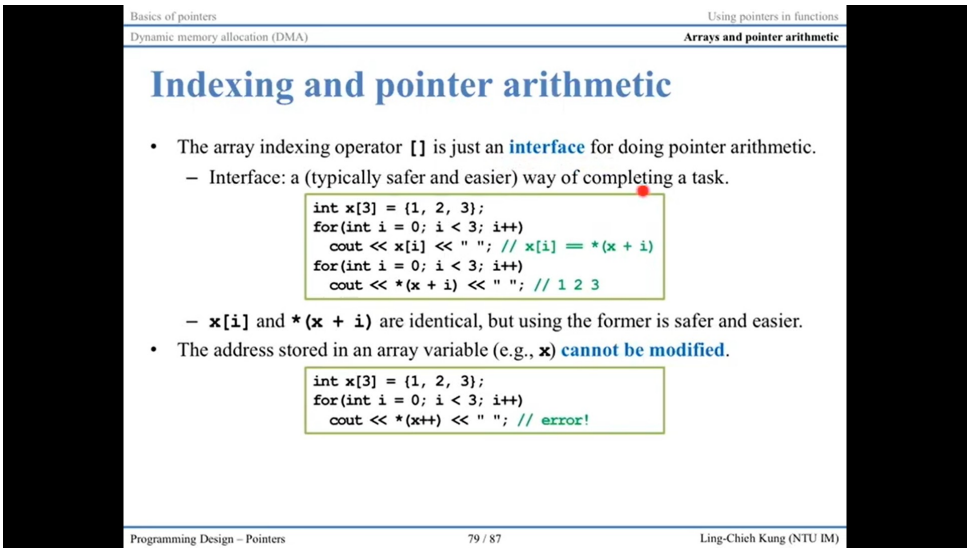


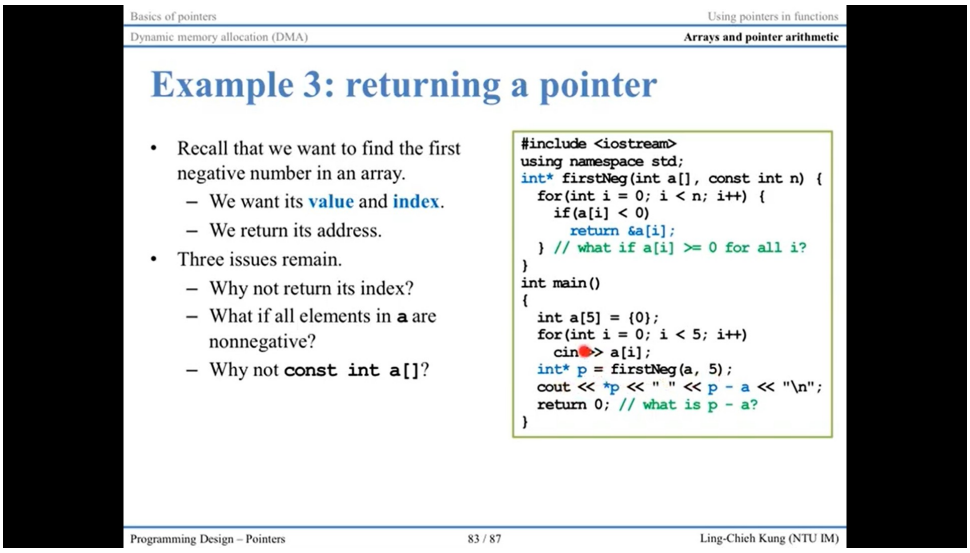
Pointer Arithmetic

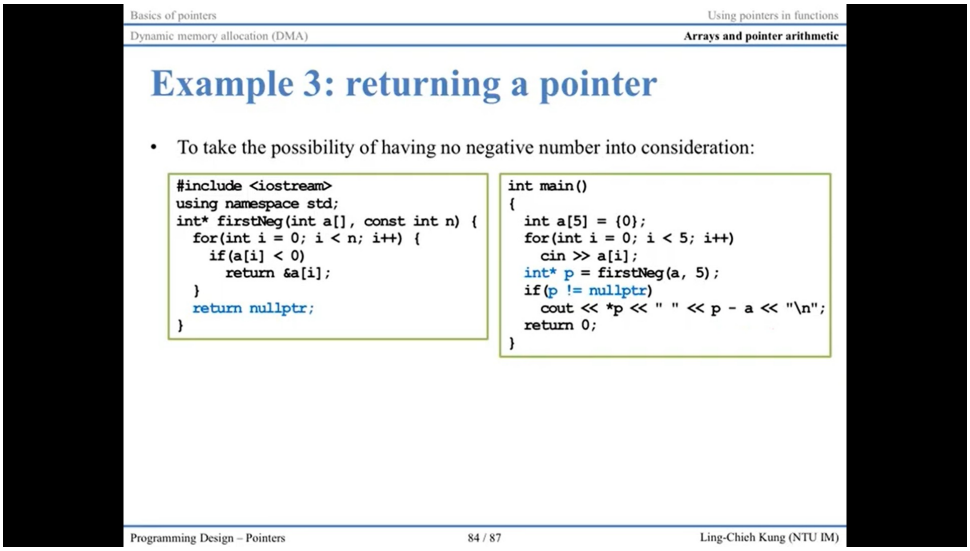


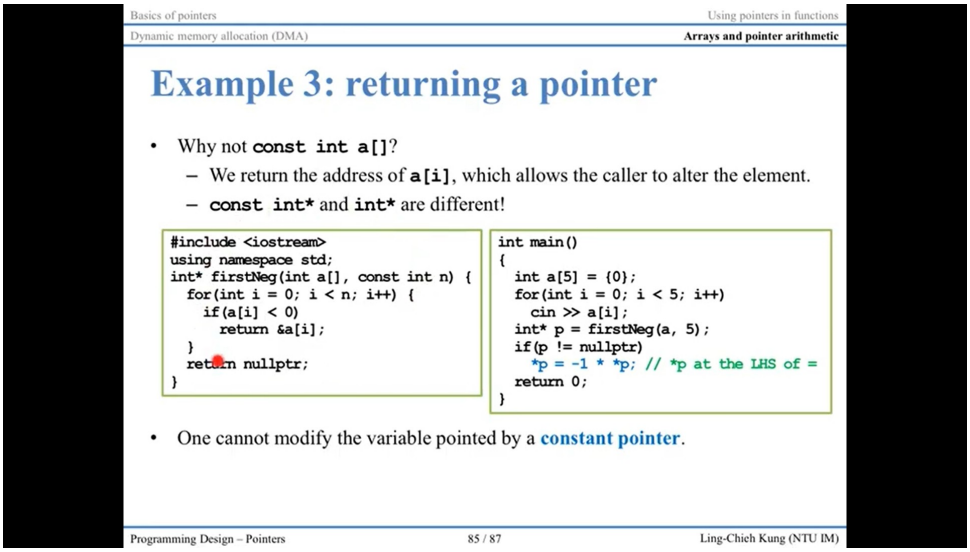
Pointer Arithmetic

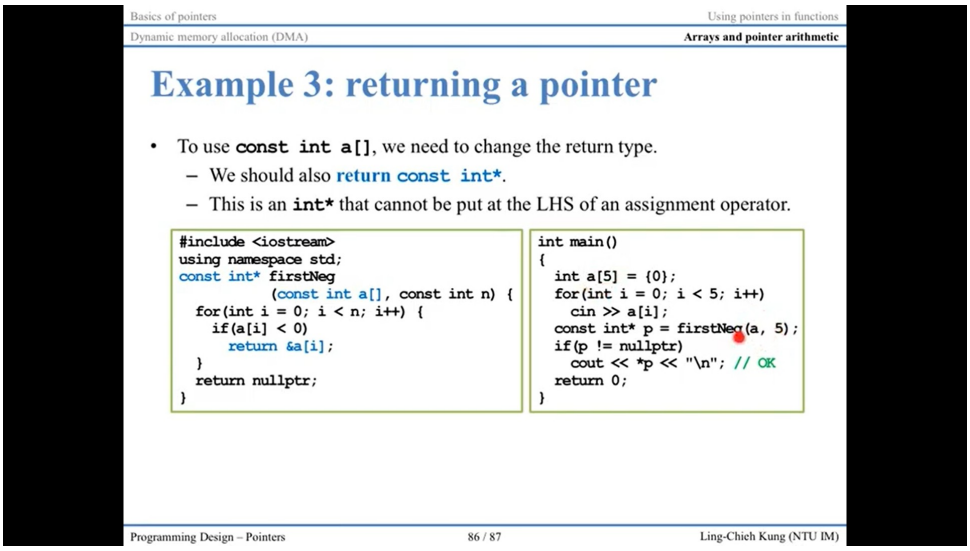
- `int *p, *q;`
  `q = p + 1;`
  - Construct a pointer to the next *integer* after `*p` and assign it to `q`
- `double *p, *r;`
  `int n;`
  `r = p + n;`
  - Construct a pointer to a *double* that is `n` *doubles* beyond `*p`, and assign it to `r`
  - `n` may be negative





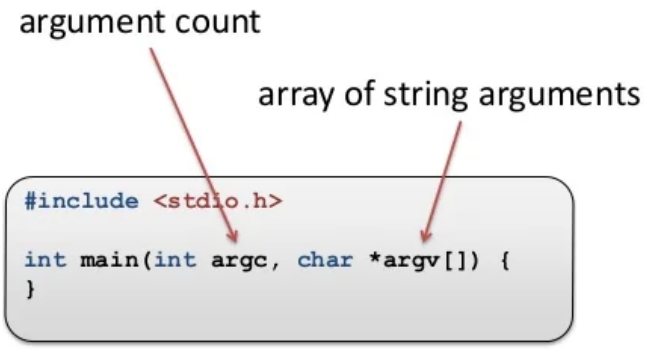






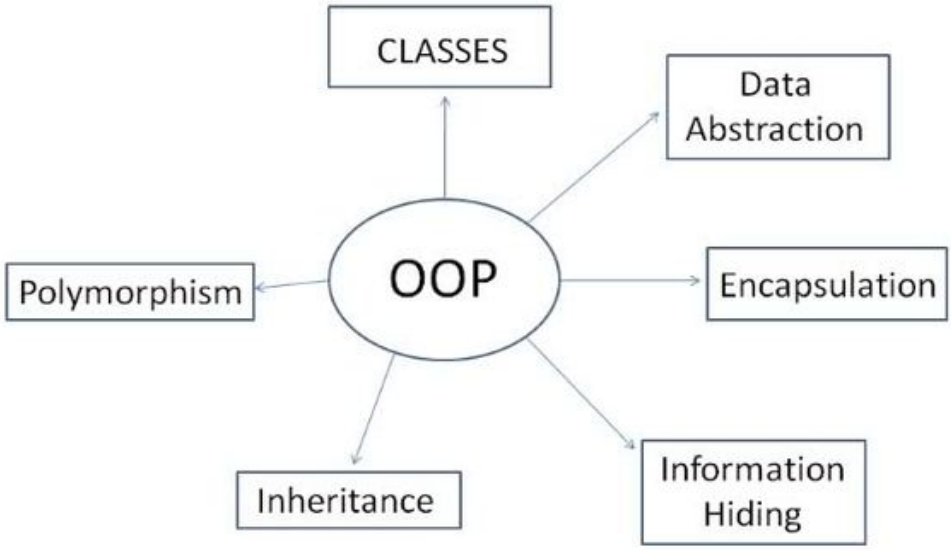
Main Function Arguments

Command line arguments



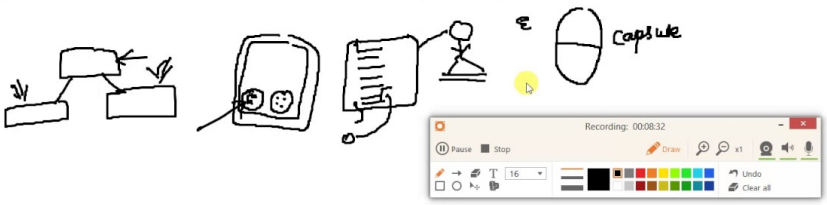
argv[0] is the program name with full path
argv[1] is the first argument
argv[2] is the second argument, etc.

OOP

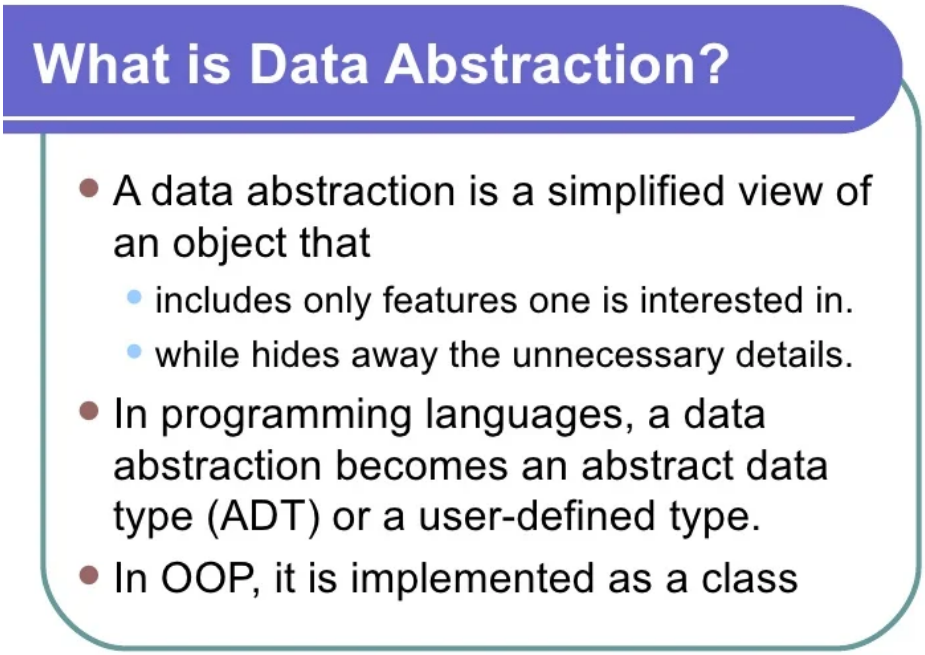


Four Pillars of Object Oriented Programming.

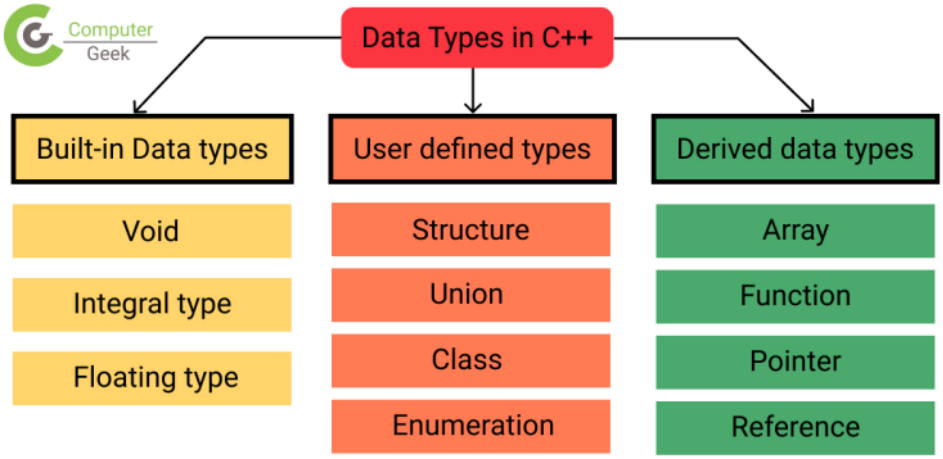
- **Inheritance:** Process of creating new classes from existing classes.
- **Abstraction:** Refers to providing only essential information and hiding the background details from the end user.
- **Encapsulation:** Binding together of data and functions that manipulate that data and keep both safe from outside interference and misuse.
- **Polymorphism:** Refers to one interface and many forms.

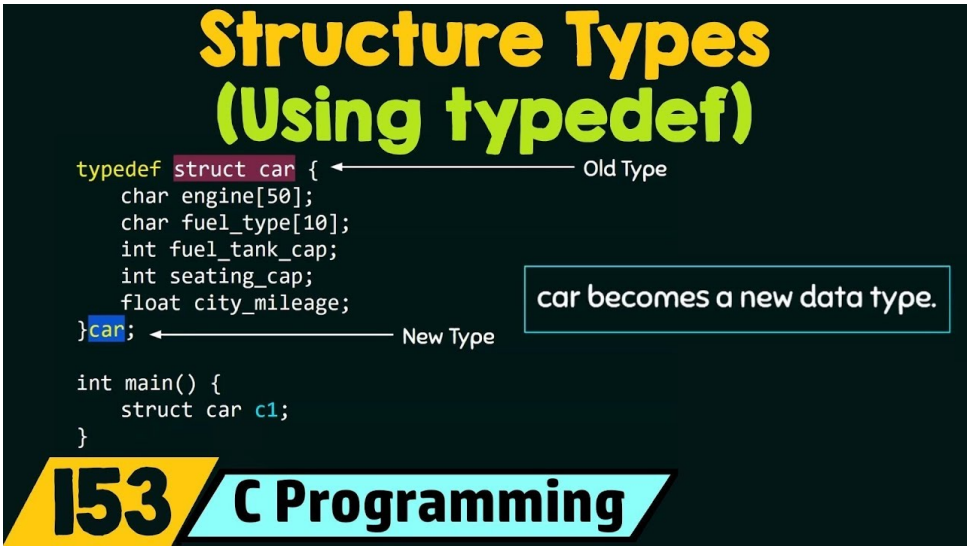


Data Abstraction



Struct in C and Class in C++



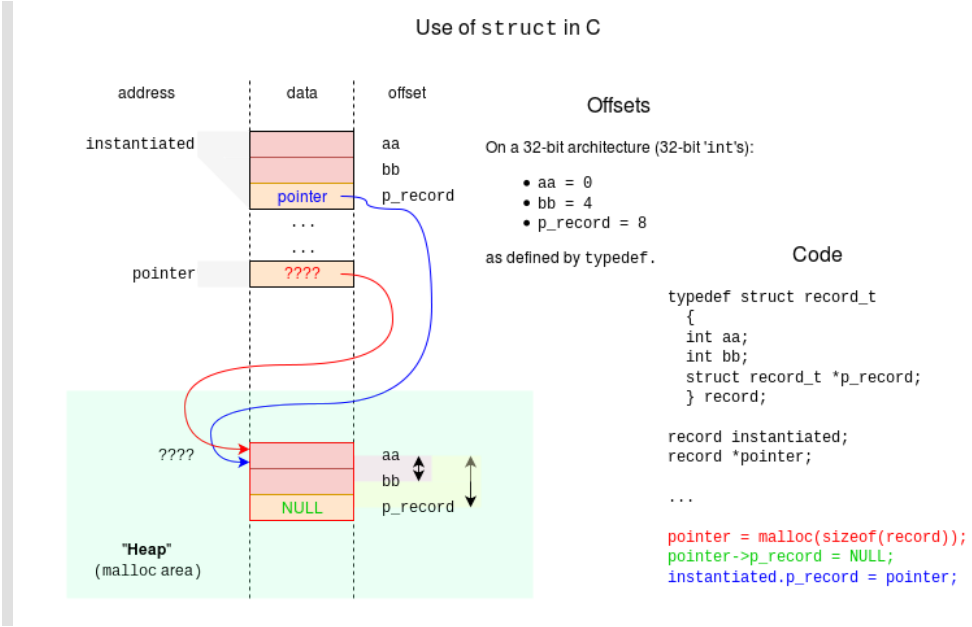


Structure

Structures within Structures

1. C define a variable of structure type as a member of other structure type.
2. The syntax to define the structure within structure is

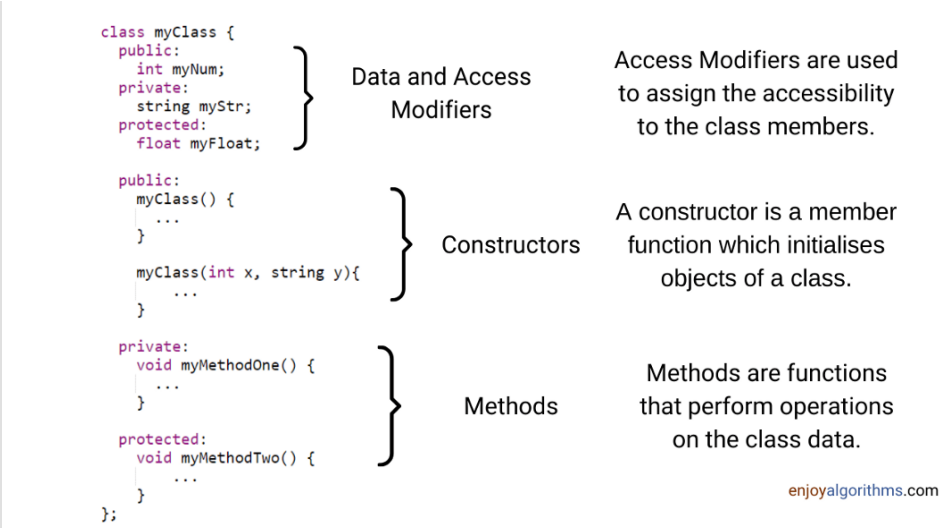
```
struct <struct_name>{
   <data_type> <variable_name>;
   struct <struct_name>
            { <data_type> <variable_name>;
            ……..}<struct_variable>;
   <data_type> <variable_name>;
};
```

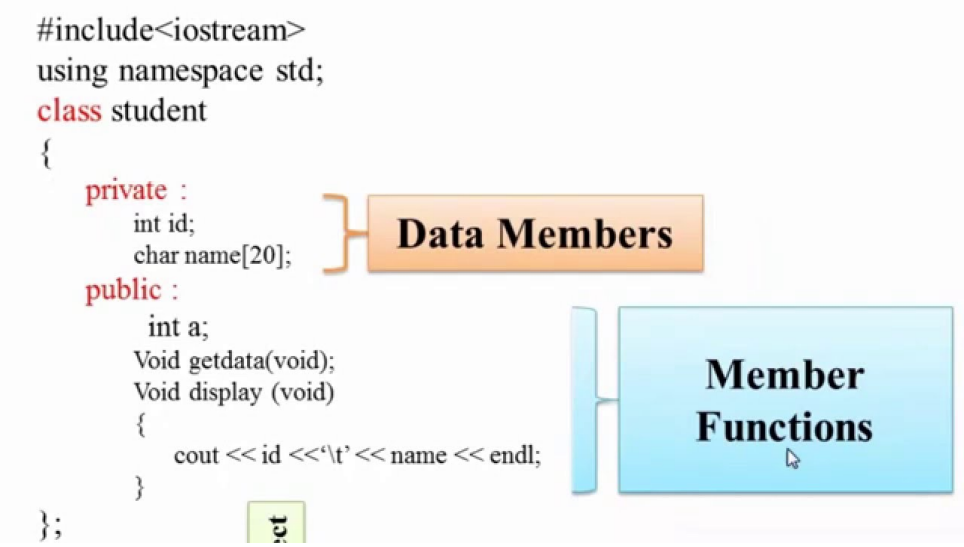


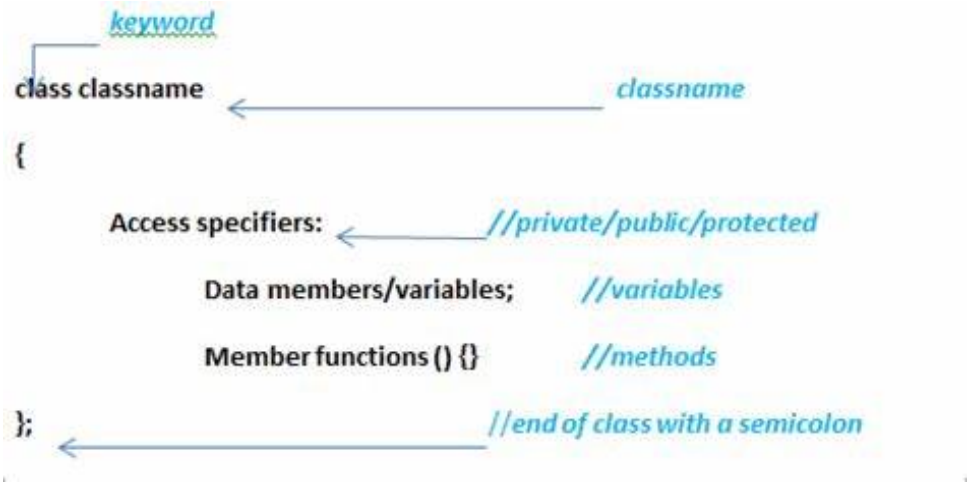
In [ ]:

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

struct MyVector {
    int n;
    int* m;
    void init(int dim);
    void print();
};

void MyVector::init(int dim){
    n = dim;
    m = new int[n];

    for(int i = 0; i < n; i++){
        m[i] = 0;
    }
}

void MyVector::print(){
    cout << "(";
    for(int i = 0; i < n-1; i++){
        cout << m[i] << ", ";
    }
    cout << m[n-1] << ")" << endl;
}


int main(){
    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: OOP - Class" << endl << endl;

    MyVector v;
    v.init(3);
    v.m[0] = 3;
    v.print();

    delete [] v.m;

    return 0;
}
```

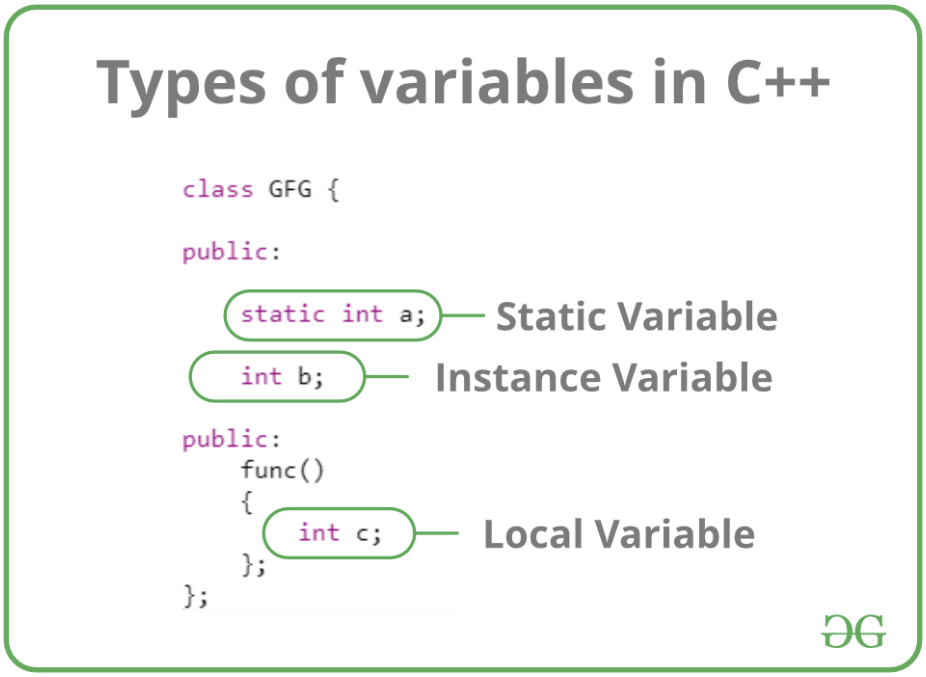
Class in C++







Variable Scope in Class



Struct vs Class

Structure	Class
It is a value type.	It is a reference type.
Its object is created on the stack memory.	Its object is created on the heap memory.
It does not support inheritance.	It supports inheritance.
The member variable of structure cannot be initialized directly.	The member variable of class can be initialized directly.
It can have only parameterized constructor.	It can have all the types of constructor and destructor.

Encapsulation

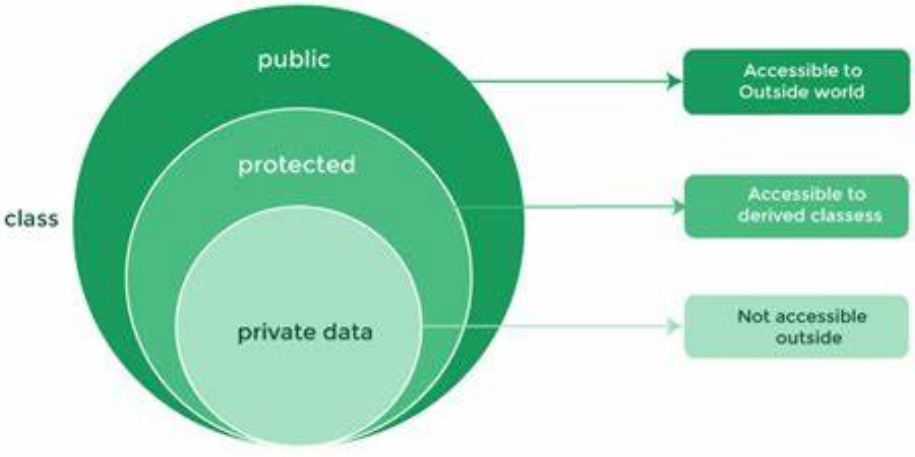
Access Control

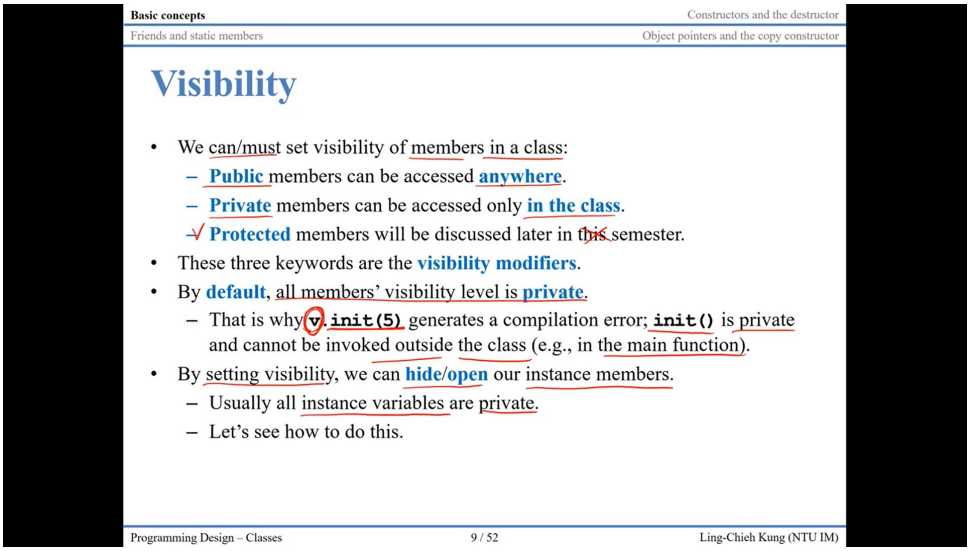
- We use the **private section** of the struct to **encapsulate** our data type as in our ADTs in C
- But if the struct is defined in the header file, doesn't this **break encapsulation**?
  - No! Encapsulation is not about *hiding information* from the user, it's about ensuring that he uses our ADT **in a safe way**, which we achieve by enabling him to use only its public interface
- However, unlike ADTs in C style, we do not get **compile-time** encapsulation
  - In other words, changing the private part of a struct will require **recompiling files** that use the ADT

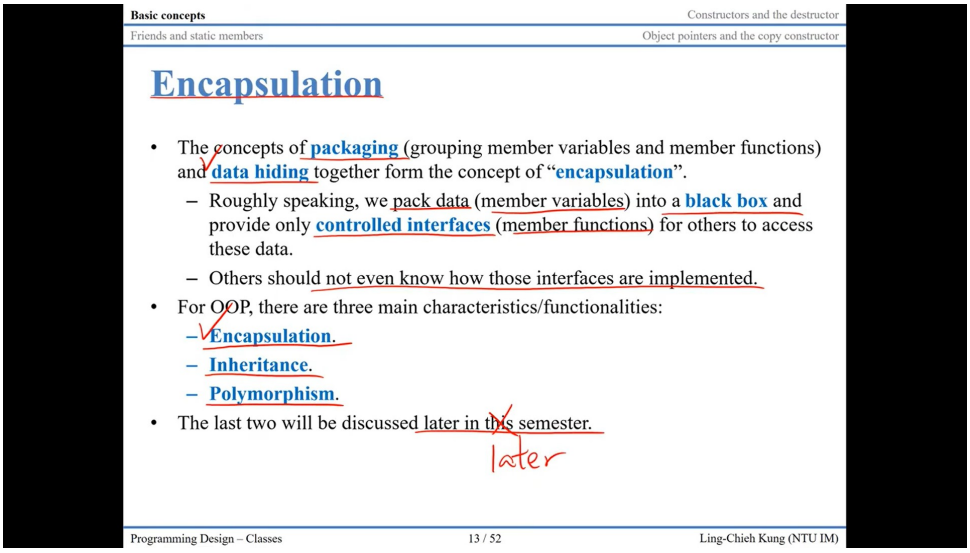
Encapsulation

- **Encapsulation** is the packing of data and functions into a single component. The features of **encapsulation** are supported using classes in most object-oriented programming languages, although other alternatives also exist.
- Encapsulation is:
  - A language mechanism for restricting access to some of the object's components. (public, private, protected)
  - A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

Visibility in Class







Class Constructor and Deconstructor

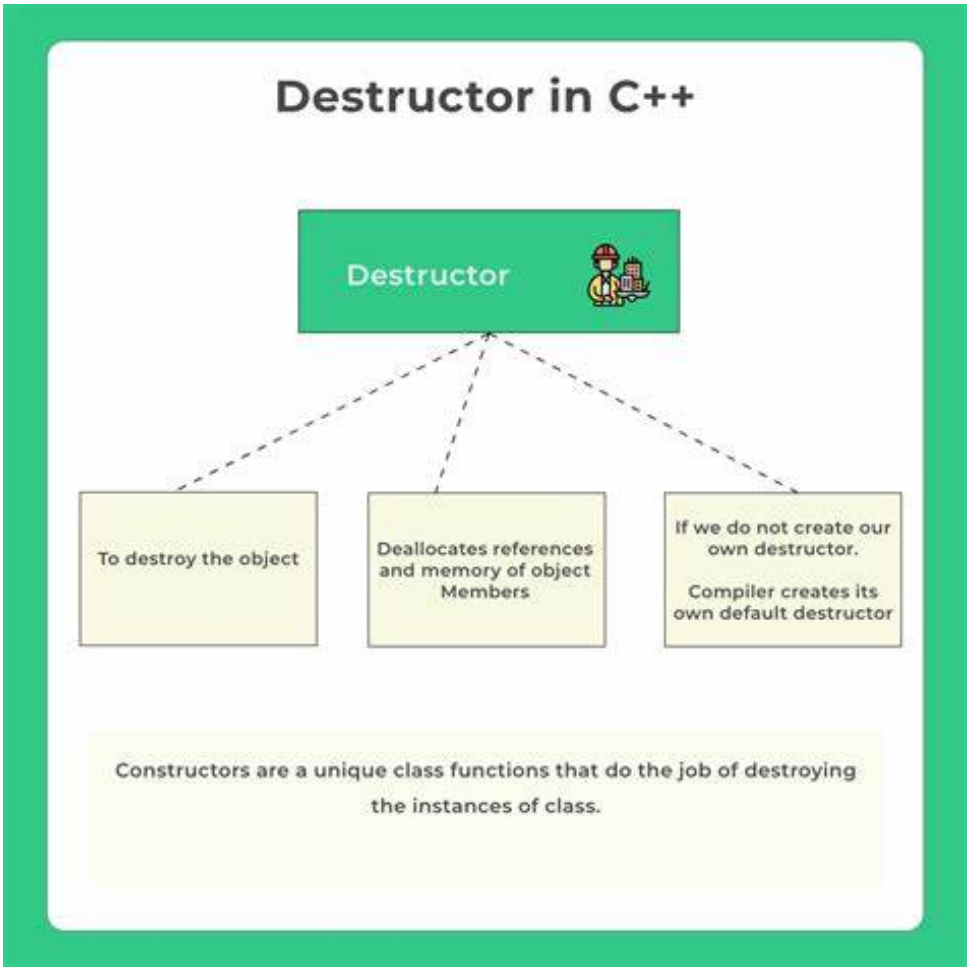
Data Abstraction in C++ (continued)

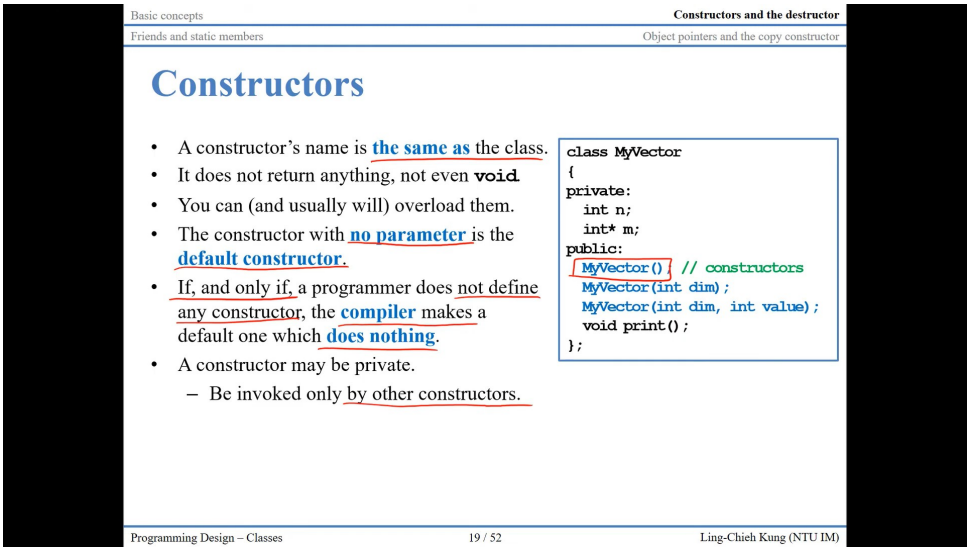
(11.14)

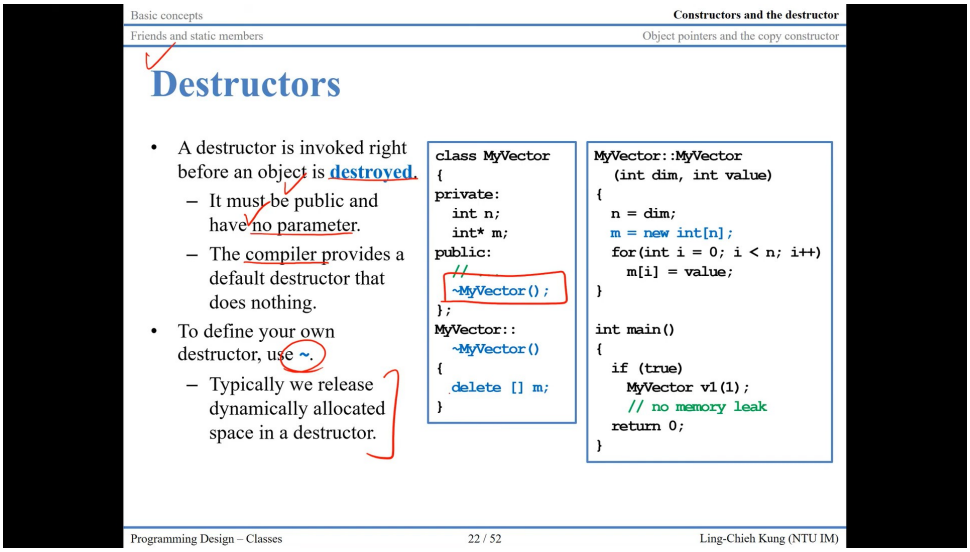
- **Constructors**
  - functions to initialize the data members of instances
  - may also allocate storage if part of the object is heap-dynamic
  - can include parameters to provide parameterization of the objects
  - implicitly called when an instance is created
    - » can be explicitly called
  - name is the same as the class name
- **Destructors**
  - functions to cleanup after an instance is destroyed; usually just to reclaim heap storage
  - implicitly called when the object's lifetime ends
    - » can be explicitly called
  - name is the class name, preceded by a tilda (~)

Class Constructors

- A class constructor is a member function whose purpose is to initialize the private data members of a class object
- The name of a constructor is always the name of the class, and there is no return type for the constructor
- A class may have several constructors with different parameter lists. A constructor with no parameters is the default constructor
- A constructor is implicitly and automaticly invoked when a class object is declared--if there are parameters, their values are listed in parentheses in the declaration







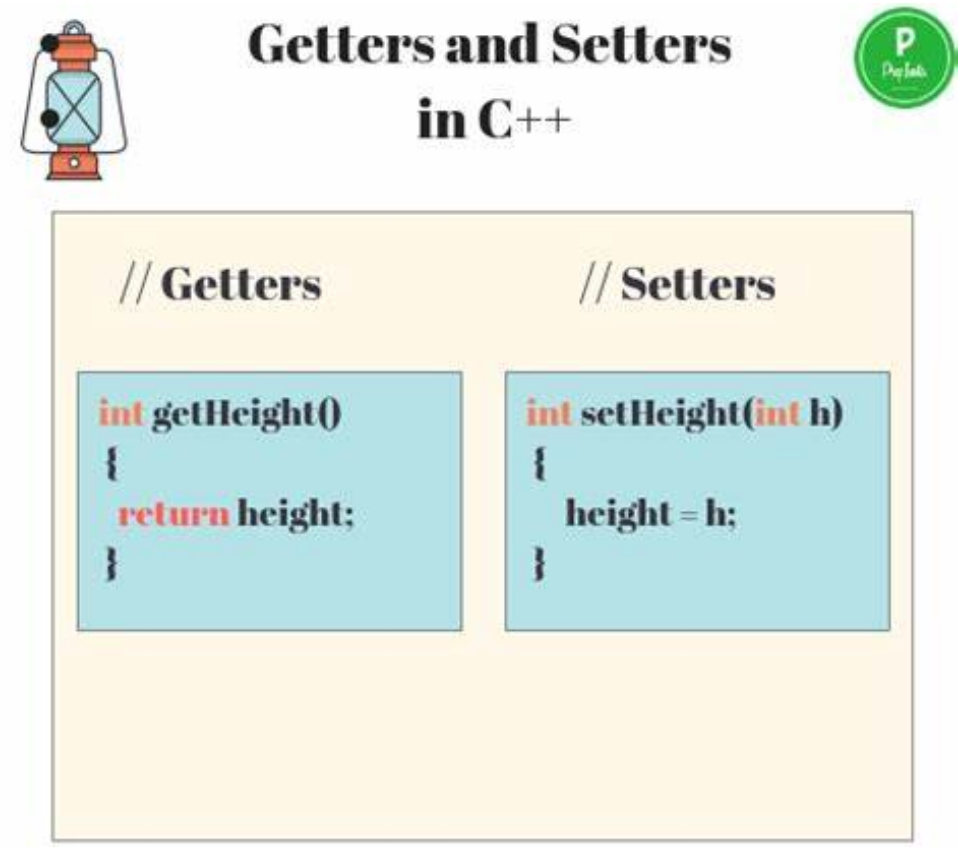
In [ ]:
```
#include <iostream>
using namespace std;


class MyVector {
private:
    int n;
    int* m;
public:
    MyVector();
    MyVector(int dim, int value = 0);
    ~MyVector();
    void print();
};

MyVector::MyVector(){
    n = 0;
    m = nullptr;
}

MyVector::MyVector(int dim, int value){
    n = dim;
    m = new int[n];
    for(int i =0; i < n; i++){
        m[i] = value;
    }
```

```
}

MyVector::~MyVector(){
    delete [] m;
}


void MyVector::print(){
    cout << "(";
    for(int i = 0; i < n-1; i++){
        cout << m[i] << ", ";
    }
    cout << m[n-1] << ")" << endl;
}


int main(){
    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: OOP - Class" << endl << endl;

    MyVector v1(1);
    MyVector v2(3, 8);

    v1.print();
    v2.print();

    return 0;
}
```

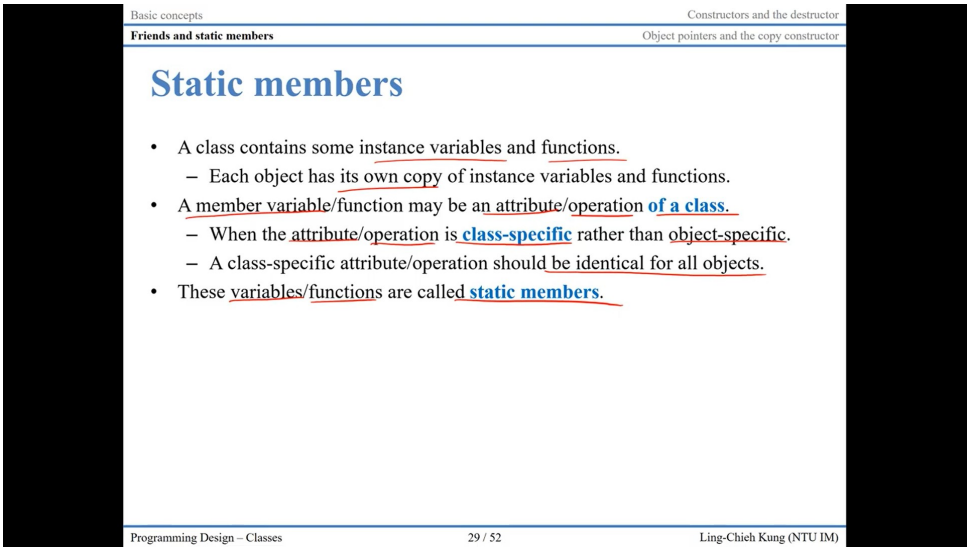
In [ ]:

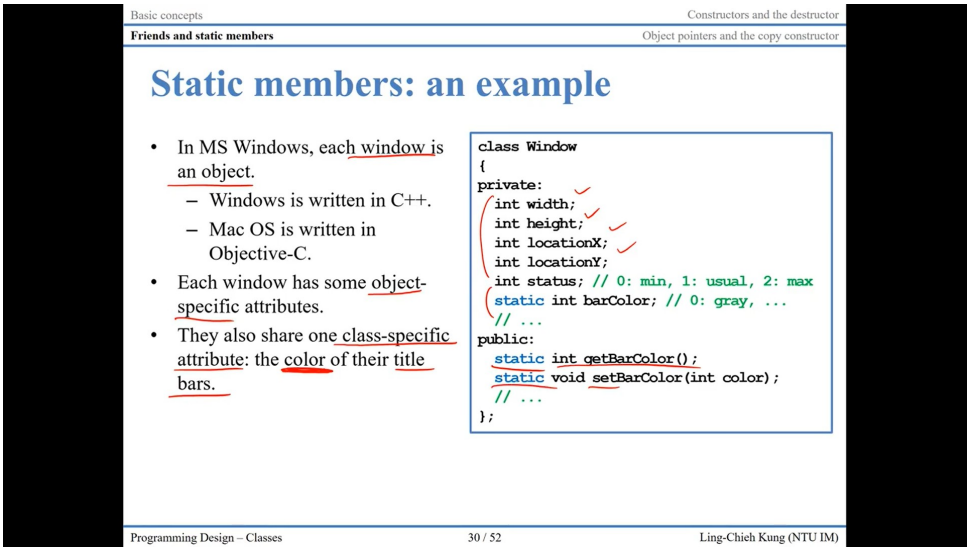
```
#include <iostream>
using namespace std;


class A {
public:
    A() { cout << "A\n"; }
    ~A() { cout << "~A\n"; }
};

class B {
private:
    A a;
public:
    B() { cout << "B\n"; }
    ~B() { cout << "~B\n"; }
};


int main(){
    B b;

    return 0;
}
```

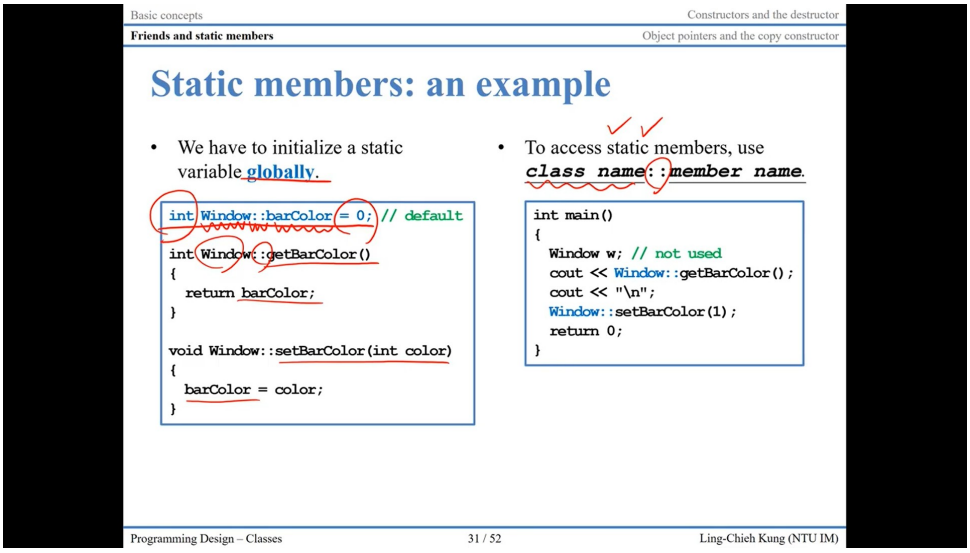
Getter and Setter

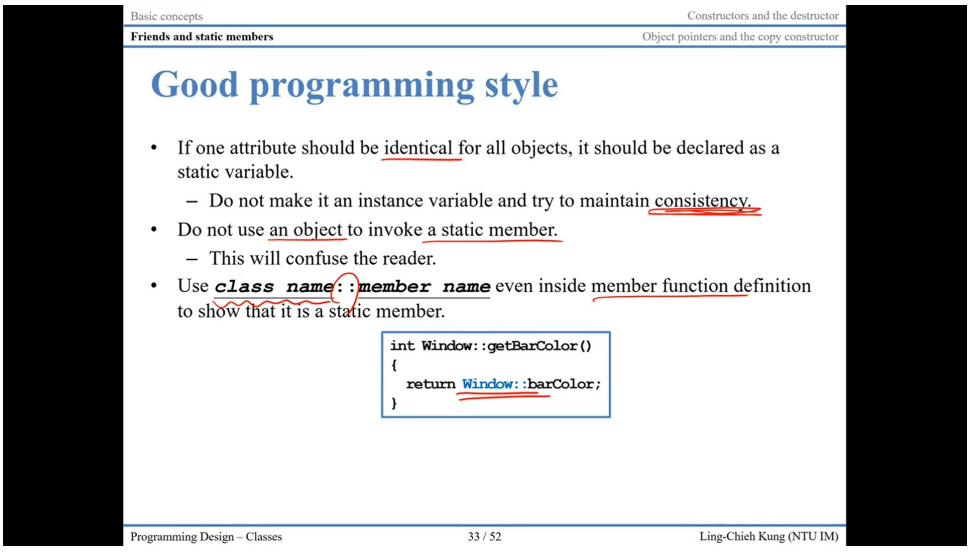


Static Member

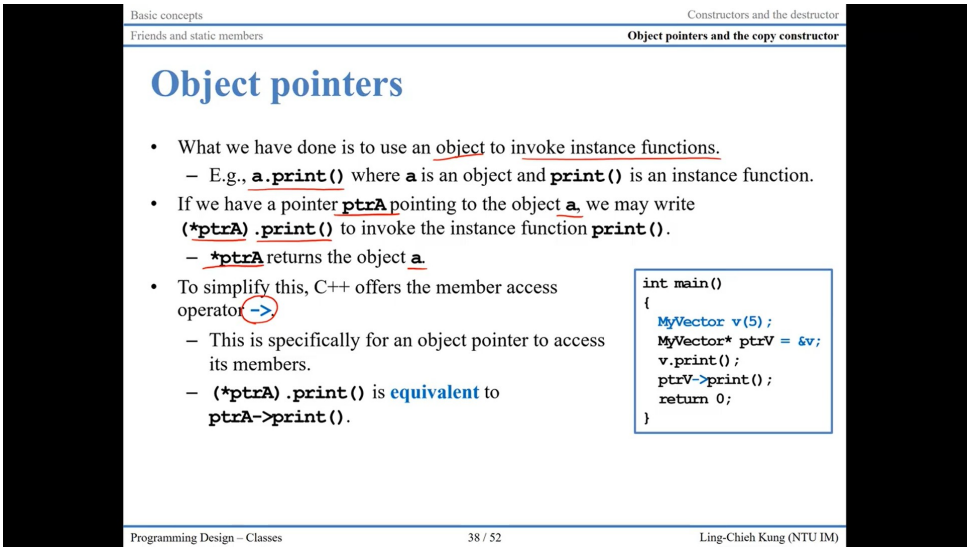


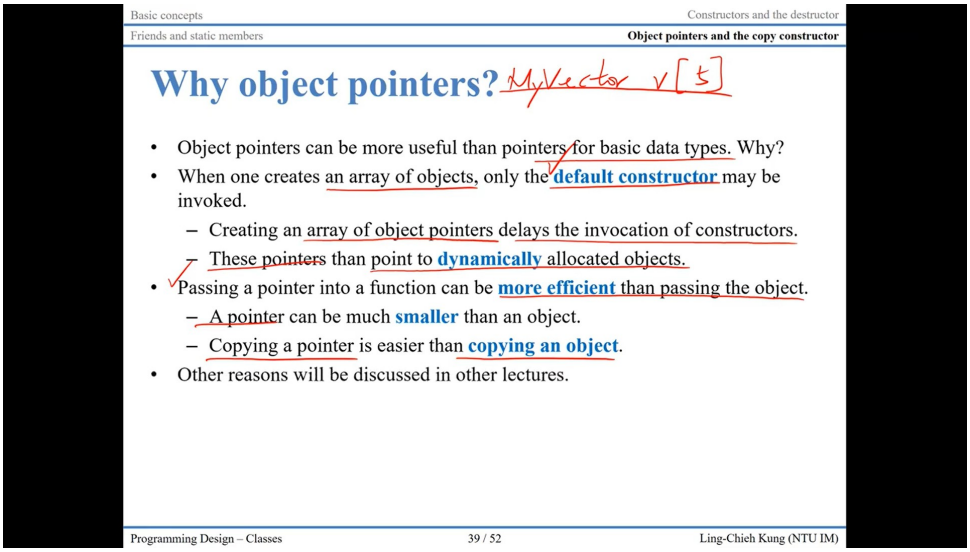


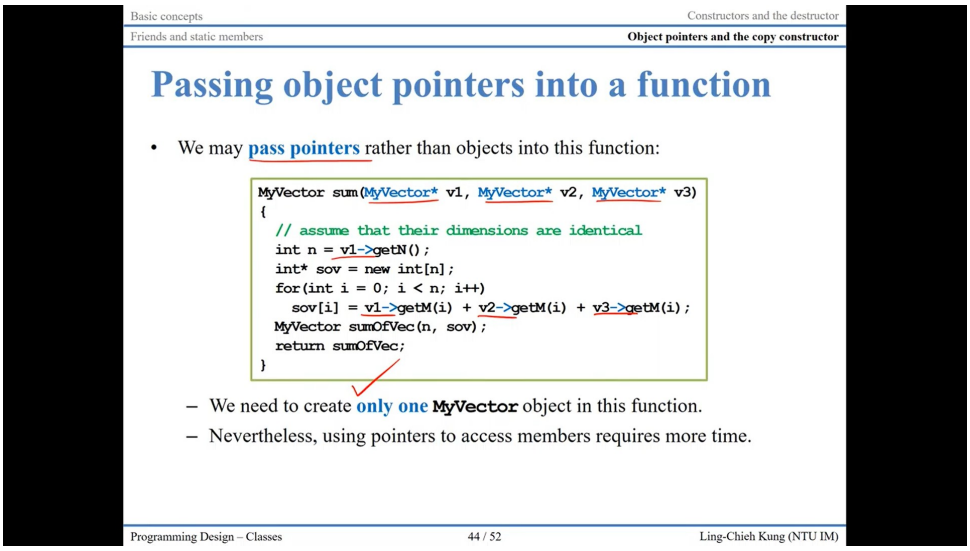




Object Pointer







Copy Constructor

Copying an object

- Creating an object by "copying" an object is a special operation.
  - When we pass an object into a function using the call-by-value mechanism.
  - When we assign an object to another object.
  - When we create an object with another object as the argument of the constructor.

```
f(a1, a2, a3);
```

```
A a4 = a1;
```

```
A a5(a1);
```

- When this happens, the **copy constructor** will be invoked.
  - If the programmer does not define one, the compiler adds a **default copy constructor** (which of course does not print out anything) into the class.
  - The default copy constructor simply copies all member variables one by one, regardless of the variable types.

Copy constructors

- We may implement our own copy constructor.
  - In the C++ standard, the parameter must be a **constant reference**.
  - If calling by value, it will invoke itself infinitely many times.

```
class A
{
private:
  int i;
public:
  A() { cout << "A"; }
  A(const A& a) { cout << "a"; }
};
```

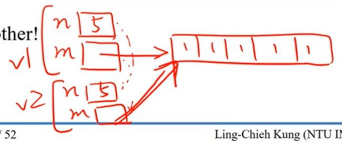
```
void f(A a1, A a2, A a3)
{
  A a4;
}
int main()
{
  A a1, a2, a3; // AAA
  cout << "\n===\n";
  f(a1, a2, a3); // aaaA
  return 0;
}
```

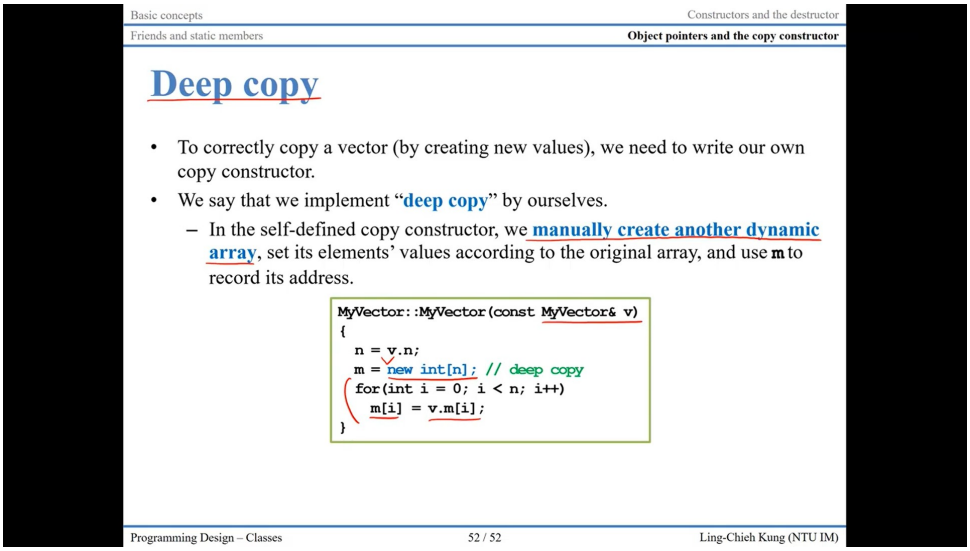

Copy constructors for MyVector

- For **MyVector**, one way to implement a copy constructor is
  - This has nothing different from the default copy constructor.
  - If no member is an array/pointer, the default copy constructor is fine.
- If there is any array or pointer member variable, the default copy constructor does "**shallow copy**".
  - And two different vectors may share the same space for values.
  - Modifying one vector affects the other!

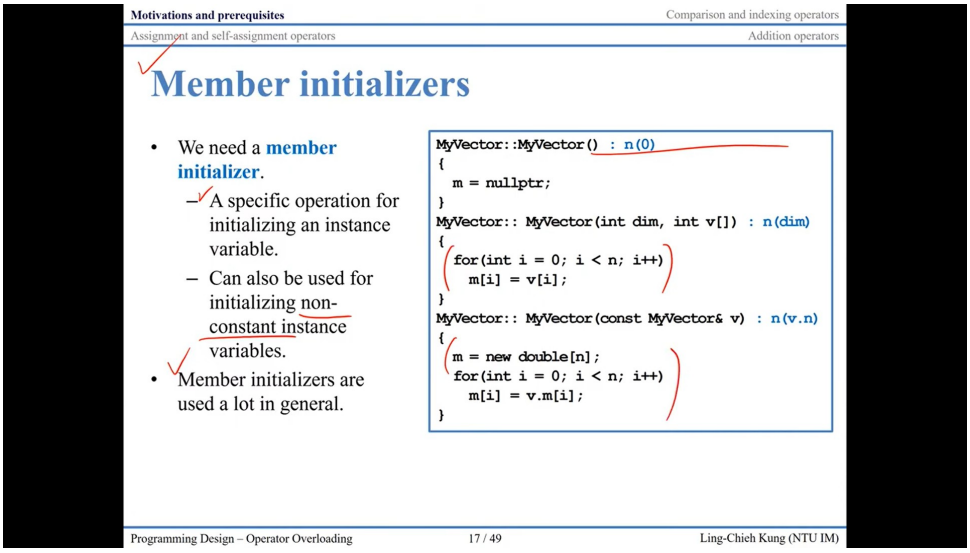
```
MyVector::MyVector
  (const MyVector& v)
{
  n = v.n;
  m = v.m;
}
```

```
int main()
{
  MyVector v1(5, 1);
  MyVector v2(v1); // what is bad?
}
```

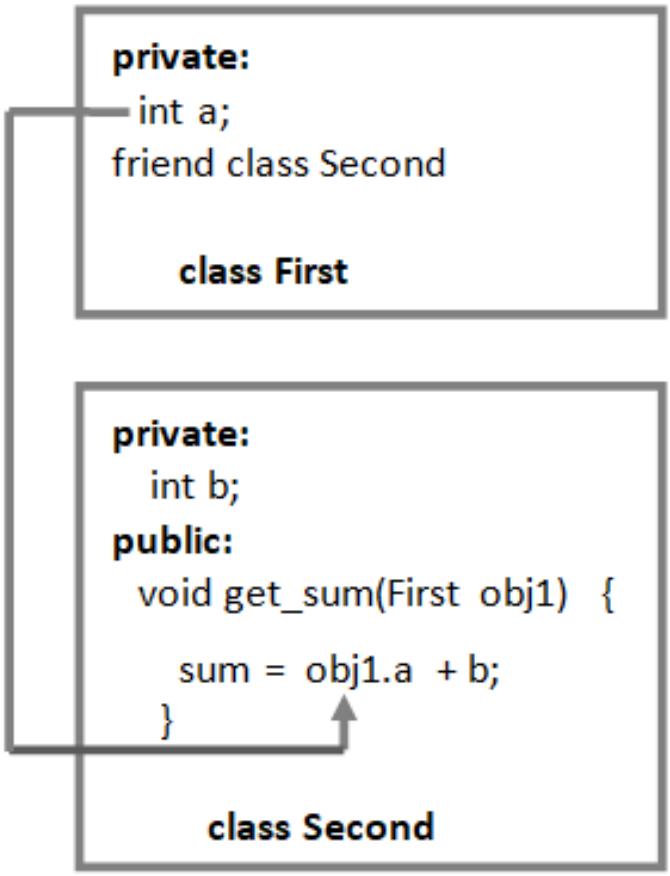




Member Initilizers

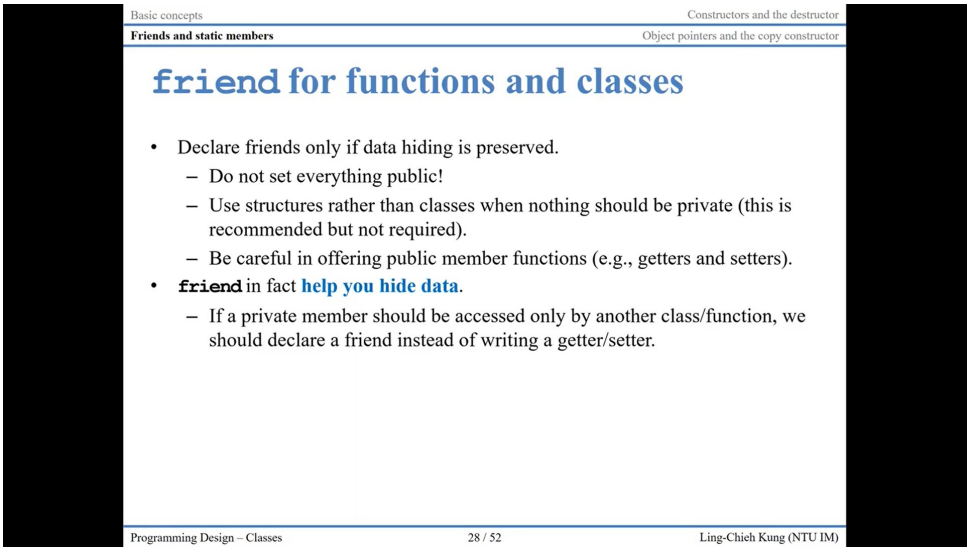


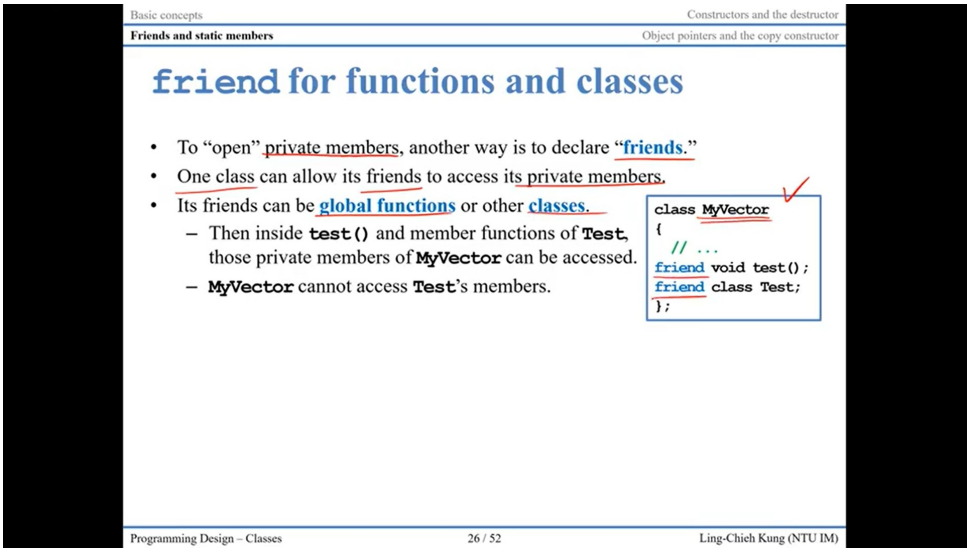
Friend



Friend function characterstics

- It is not in scope of class.
- It cannot be called using object of that class.
- It can be invoked like a normal function.
- It should use a dot operator for accessing members.
- It can be public or private.
- It has objects as arguments.
- Perhaps the most common use of friend functions is **overloading << and >>** for I/O.





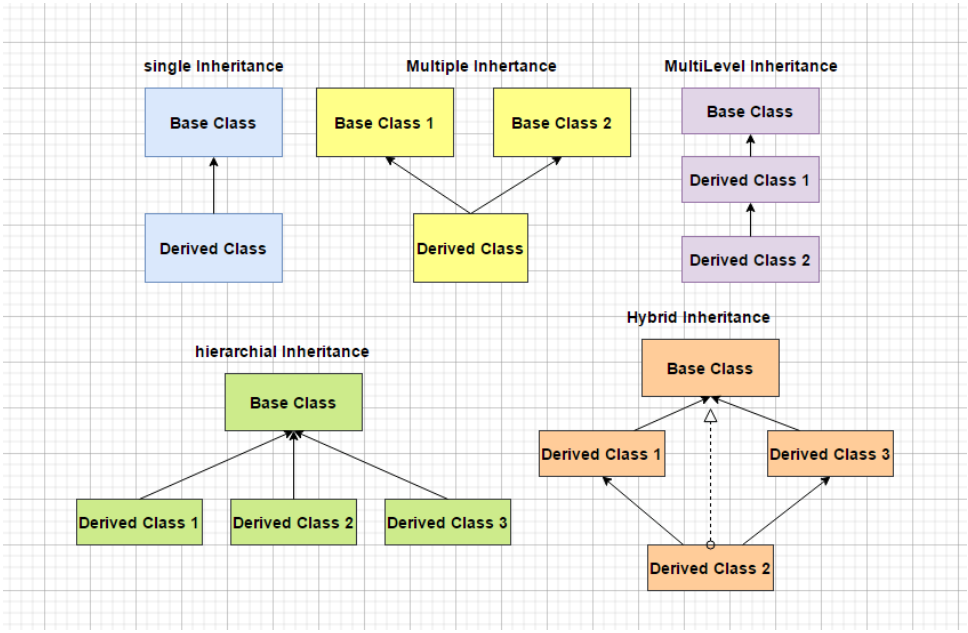
In [ ]:
```
#include <iostream>
#include <cstdlib>
using namespace std;


class MyVector {
private:
    int n;
    double* m;
public:
    MyVector();
    MyVector(int dim, double v[]);
    MyVector(const MyVector& v); //Deep Copy
    ~MyVector();
    void print() const;
    bool operator== (const MyVector& v) const;
    bool operator!= (const MyVector& v) const;
    double operator[] (int i) const;
    double& operator[] (int i);
    MyVector& operator= (const MyVector& v);
    const MyVector& operator+= (const MyVector& v);
    const MyVector operator+ (const MyVector& v);
    const MyVector operator+ (double d);
    friend const MyVector operator+ (const MyVector& v, double d);
};
```

```
MyVector::MyVector() : n(0), m(nullptr) {
}

MyVector::MyVector(int dim, double v[]) : n(dim){
    m = new double[n];
    for(int i =0; i < n; i++){
        m[i] = v[i];
    }
}

//Deep Copy
MyVector::MyVector(const MyVector& v): n(v.n){
    m = new double[n];
    for(int i = 0; i < n; i++){
        m[i] = v.m[i];
    }
}


MyVector::~MyVector(){
    delete [] m;
}

void MyVector::print() const {
    cout << "(";
    for(int i = 0; i < n - 1; i++){
        cout << m[i] << ", ";
    }
    cout << m[n - 1] << ")";
}

bool MyVector::operator== (const MyVector& v) const {
 if(n != v.n)
    return false;
 else {
    for(int i = 0; i < n; i++){
        if(m[i] != v.m[i])
            return false;
    }
 }
 return true;
}

bool MyVector::operator!= (const MyVector& v) const {
    return (*this == v);
}

double MyVector::operator[] (int i) const {
    if (i < 0 || i >=n){
        exit(1); //terminate the program! required <cstdlib>
    }
    return m[i];
}

double& MyVector::operator[] (int i){
    if (i < 0 || i >= n){
        exit(1);
    }
    return m[i];
}
```

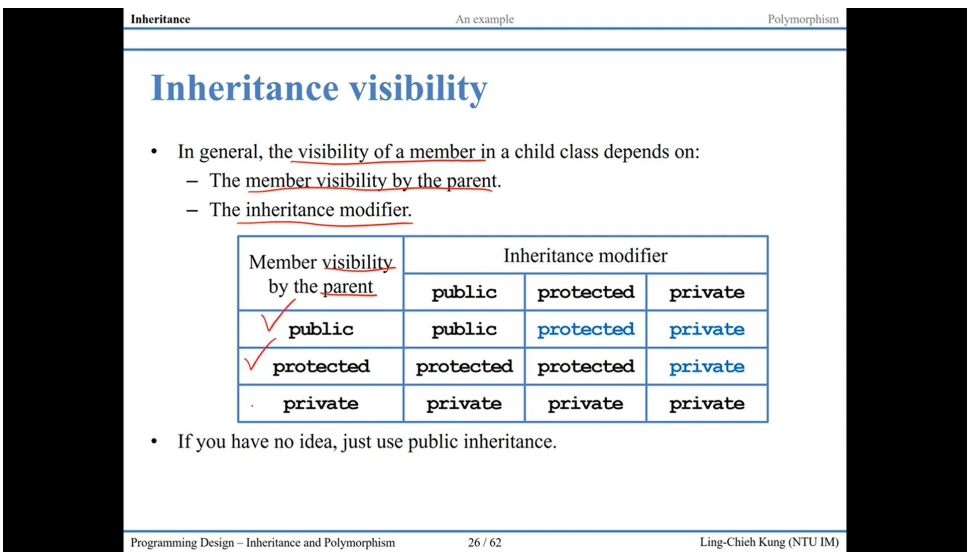
```
MyVector& MyVector::operator= (const MyVector& v){
    if(this != &v){
        if(this->n != v.n){
            delete [] this->m;
            this->n = v.n;
            this->m = new double[this->n];
        }
        for(int i = 0; i < n; i++){
            this->m[i] = v.m[i];
        }
    }
    return *this;
}

const MyVector& MyVector::operator+= (const MyVector& v){
    if(this->n == v.n){
        for(int i = 0; i < n; i++){
            this->m[i] += v.m[i];
        }
    }
    return *this;
}

const MyVector MyVector::operator+ (const MyVector& v){
    MyVector sum(*this); //MyVector(const MyVector& v)
    sum += v;
    return sum;
}

const MyVector MyVector::operator+ (double d){
    MyVector sum(*this);
    for(int i = 0; i < n; i++){
        sum[i] += d;
    }
    return sum;
}

const MyVector operator+ (const MyVector& v, double d){
    MyVector sum(v);
    for(int i = 0; i < v.n; i++){
        sum[i] += d;
    }
    return sum;
}

const MyVector operator+ (double d, const MyVector& v){
    return v + d;
}

const MyVector operator+ (const MyVector& v1, const MyVector& v2){
    MyVector sum(v1);
    sum += v2;
    return sum;
}



int main(){
    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: OOP - Class" << endl << endl;
```

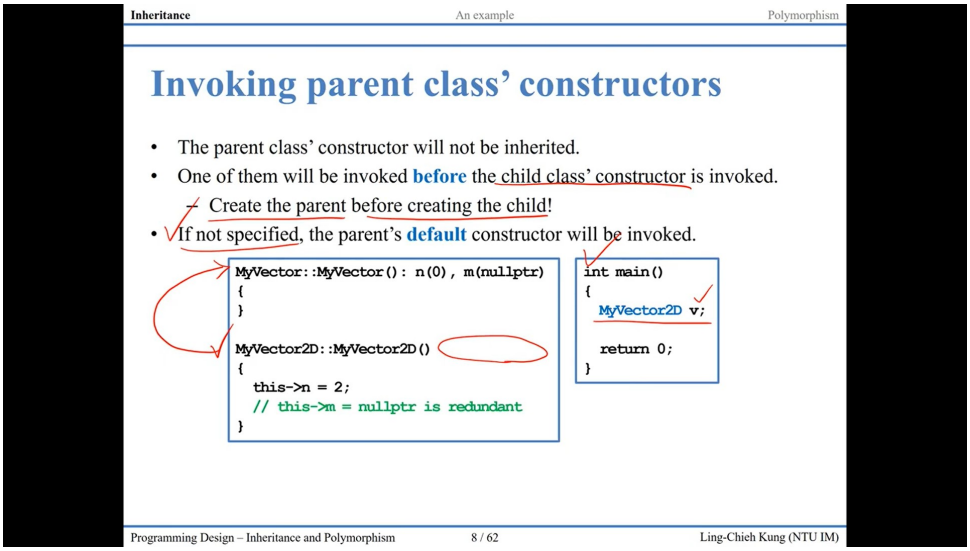
```
    double d1[5] = {1, 2, 3, 4, 5};
    MyVector a1(5, d1);

    double d2[4] = {1, 2, 3, 4};
    MyVector a2(4, d2);

    MyVector a3(a1);

    cout << (a1 == (a2) ? "Y" : "N");
    cout << endl;
    cout << (a1 == (a3) ? "Y" : "N");
    cout << endl;

    cout << "a1[3]: "<< a1[3] << endl;
    a1[1] = 999;
    cout << "a1[1]: " << a1[1] <<endl;

    cout << "a1: ";
    a1.print();
    cout << endl;

    cout << "a2: ";
    a2.print();
    cout << endl;

    cout << "a3: ";
    a3.print();
    cout << endl;

    a1 += a3;
    cout << "a1 + a3: ";
    a1.print();
    cout << endl;

    a1 = a1 + 4.2;
    cout << "a1 + 4.2: ";
    a1.print();
    cout << endl;

    a2 = 100+ a2;
    cout << "100 + a2: ";
    a2.print();
    cout << endl;

    return 0;
}
```

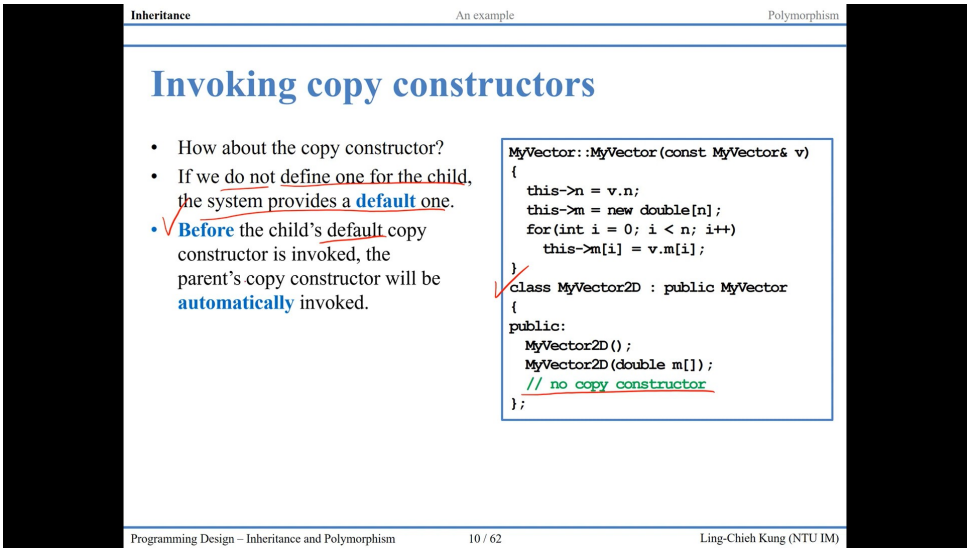
Inheritance

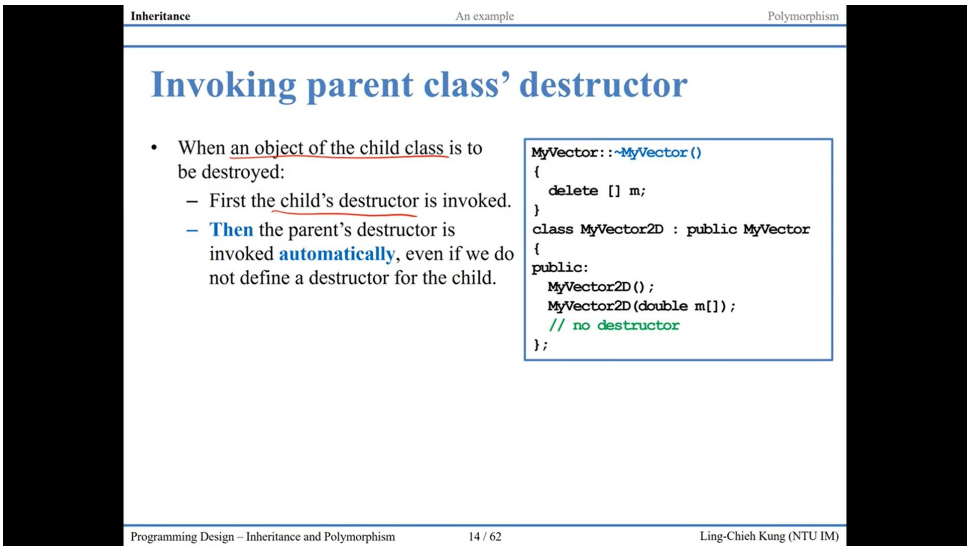


Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

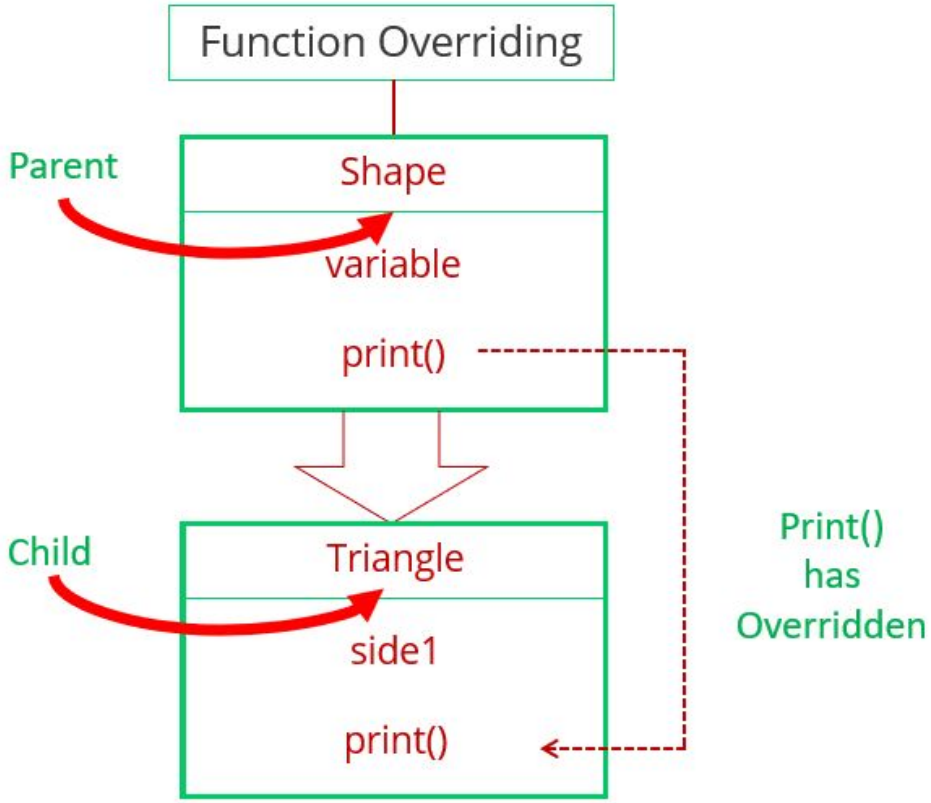






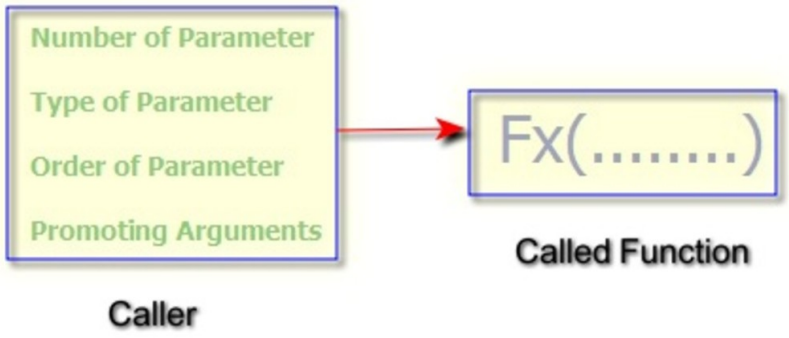


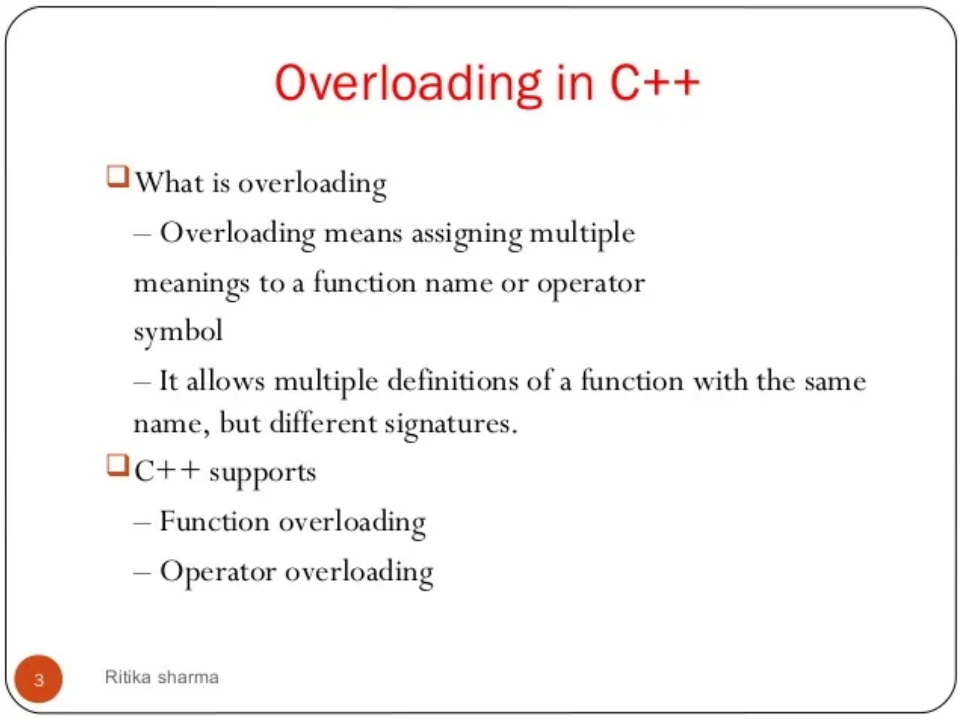
Overriding vs Overloading

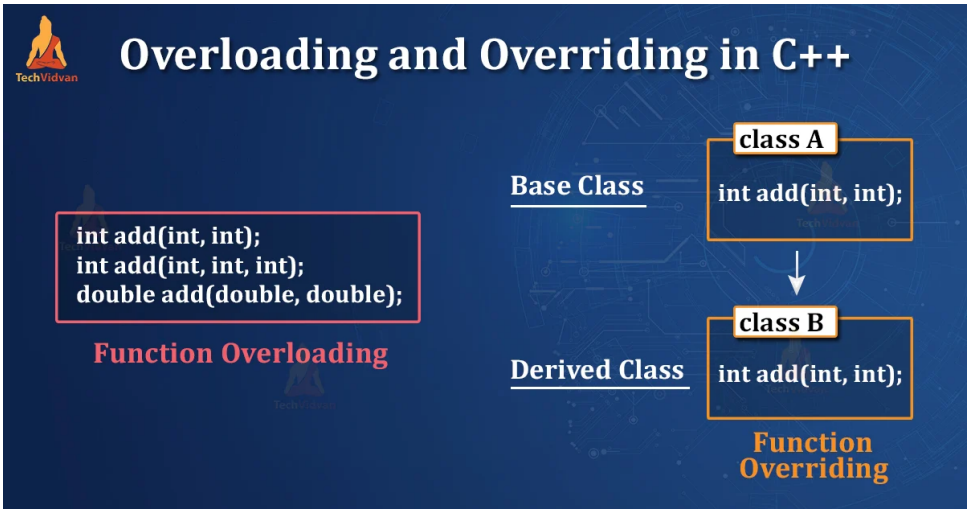


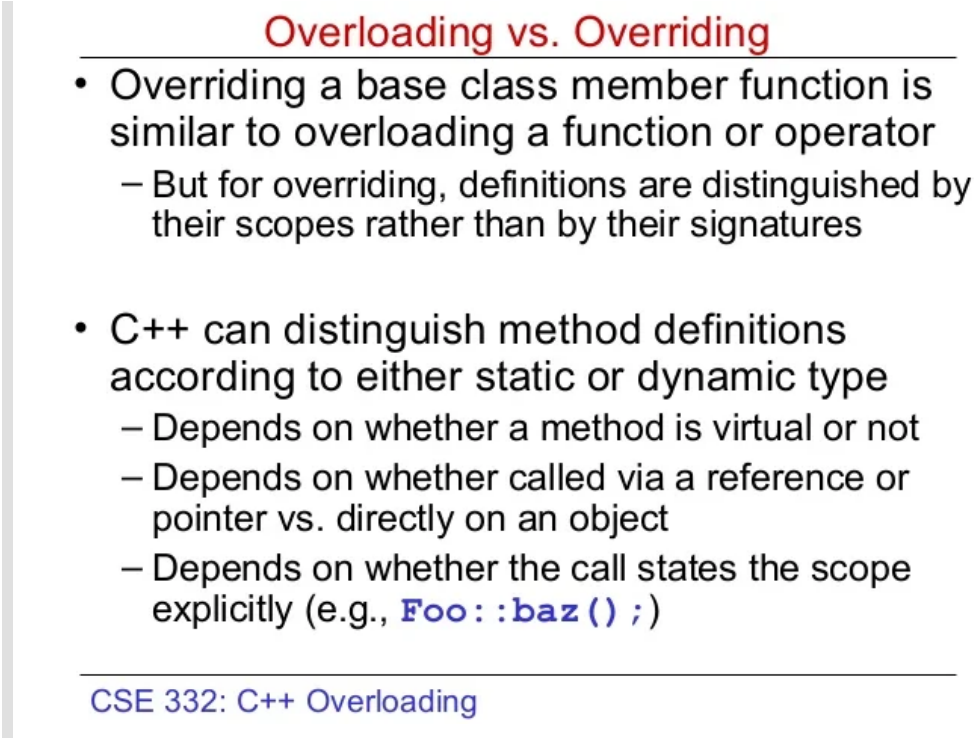
Function signature and Parameters

- **Function signature** is the combination of the function name and the parameter list.
- Variables defined in the function header are known as **formal parameters.**
- When a function is invoked, you pass a value to the parameter. This **value** is referred to as **actual parameter or argument**.

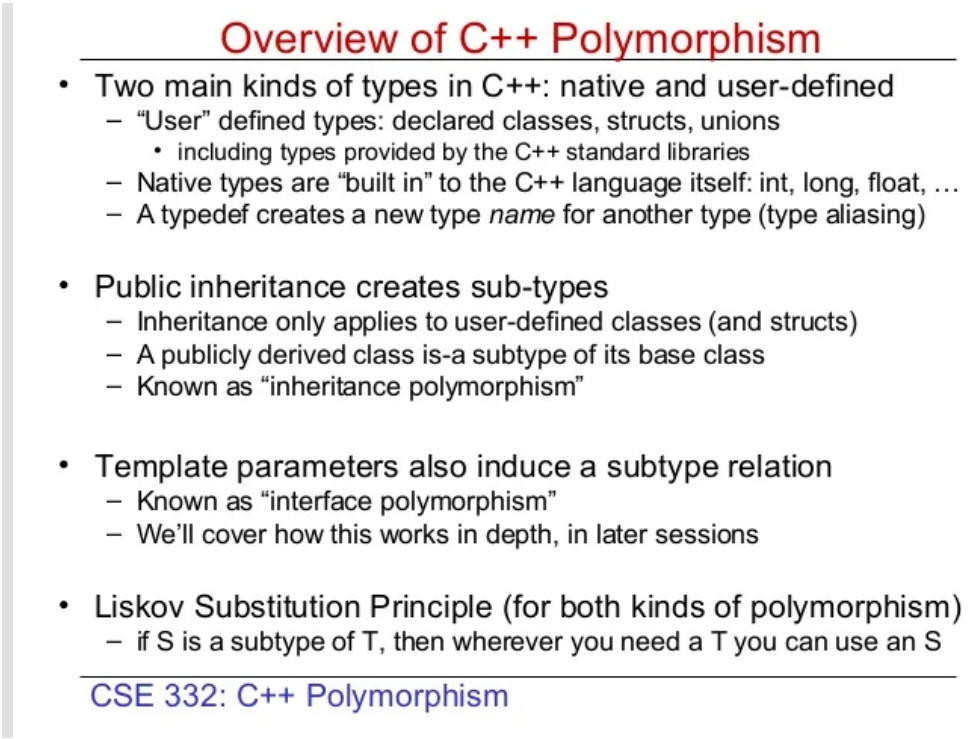

Number of Parameter
Type of Parameter
Order of Parameter
Promoting Arguments
Caller
Fx(.....)
Called Function

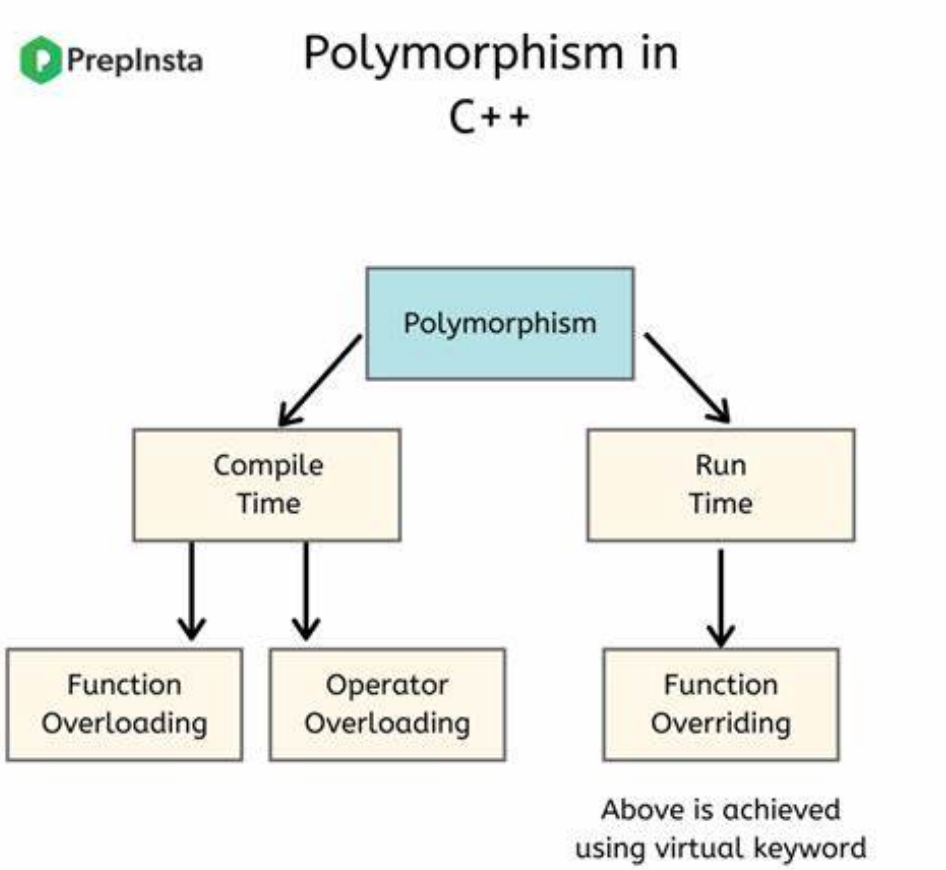

Overloading in C++
What is overloading
– Overloading means assigning multiple meanings to a function name or operator symbol
– It allows multiple definitions of a function with the same name, but different signatures.
C++ supports
– Function overloading
– Operator overloading
3
Ritika sharma


TechVidvan
Overloading and Overriding in C++
int add(int, int);
int add(int, int, int);
double add(double, double);
Function Overloading
Base Class
class A
int add(int, int);
Derived Class
class B
int add(int, int);
Function Overriding

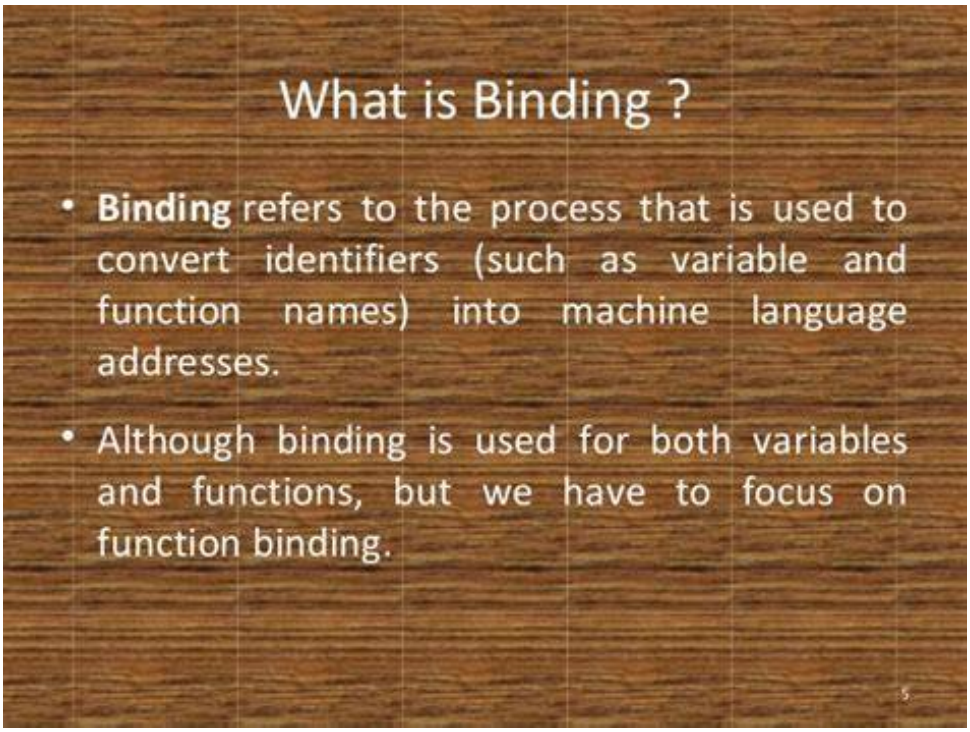


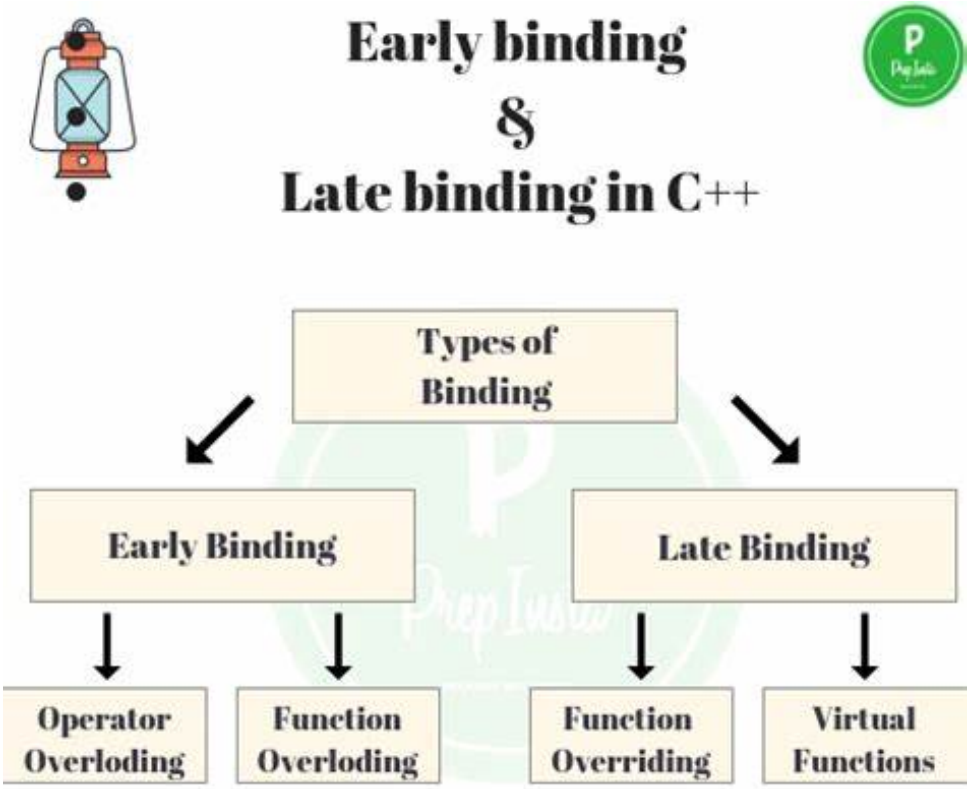
Polymorphism



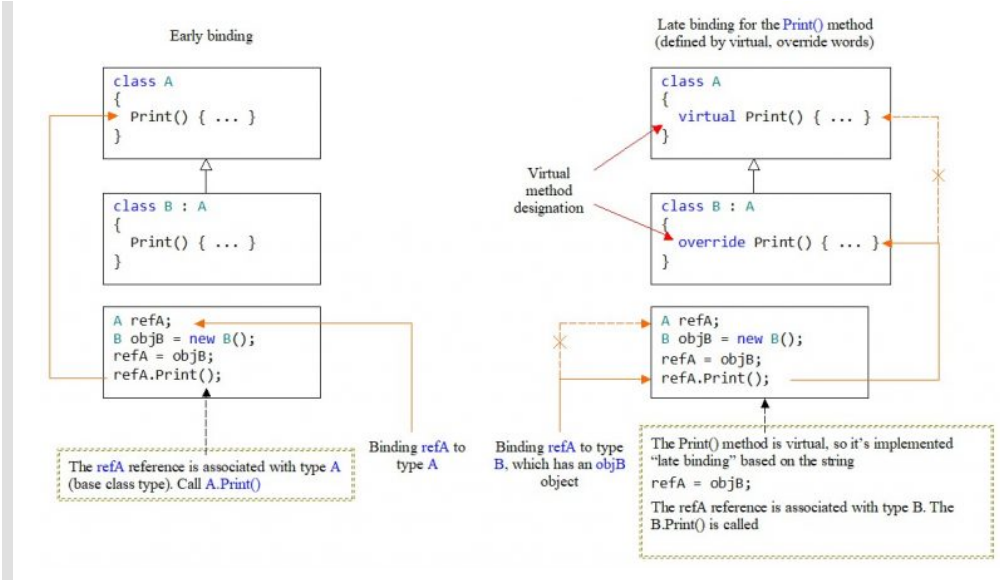


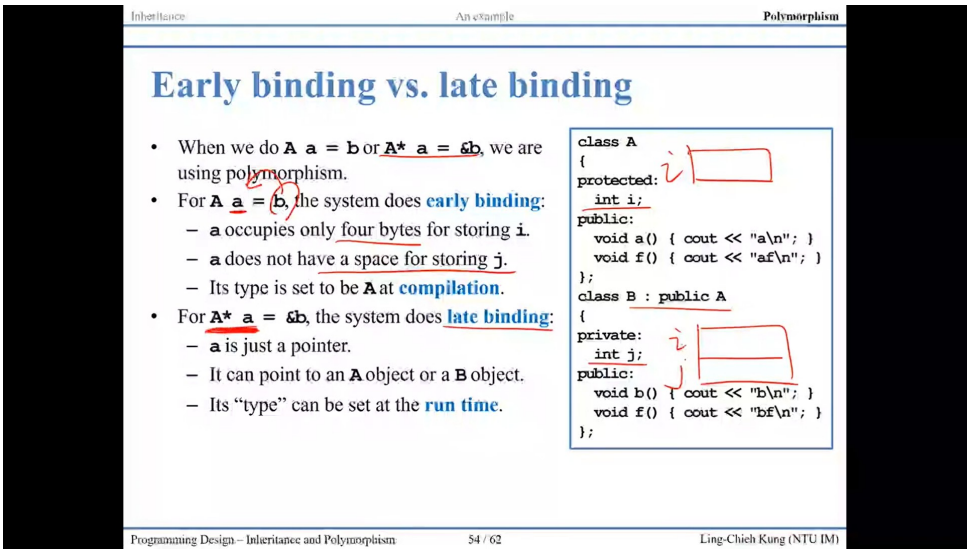
Early Binding and Late Binding



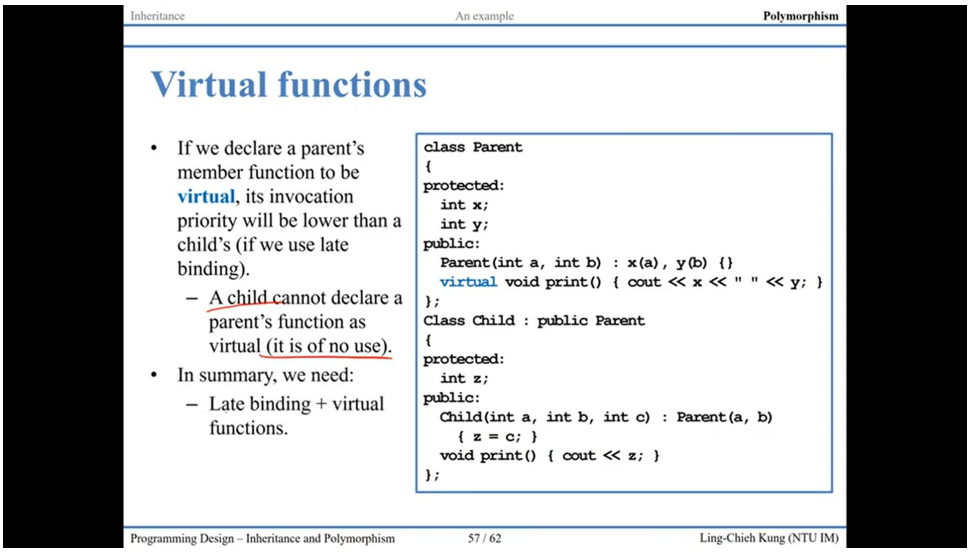


Sr. No.	Early Binding	Late Binding
1.	Early binding is the process of linking a function with an object during the compilation process.	Late binding is a run-time polymorphism with method overriding.
2.	Static binding is another name for early binding.	Dynamic binding is another name for late binding.
3.	Early binding happens at compile time.	Late binding happens at run time.
4.	Execution speed is faster in early binding.	Execution speed is lower in late binding.
5.	The class information is used by Early binding to resolve method calls.	The object is used by Late binding to resolve method calls.





Virtual Function



Abstract classes

- The two virtual functions are different in their natures:
  - `print()` is invoked in the children's implementations.
  - `beatMonster()` should not be invoked by any one.
- We may set `beatMonster()` to be a **pure virtual function**:

```
class Character
{
  // ...
  virtual void beatMonster(int exp) = 0;
};
```

  - Now we do not need to implement it.
  - Moreover, we **cannot** create `Character` objects!

Summary

- Polymorphism is a technique to make our program clearer, more flexible and more powerful.
  - It is based on **inheritance**.
  - It is tightly related to **function overriding**, **late binding**, and **virtual functions**.
- The key action is to "use a parent pointer to point to a child object".

Virtual Functions

```
class A {
public:
  void x() {cout<<"A::x";};
  virtual void y() {cout<<"A::y";};
};

class B : public A {
public:
  void x() {cout<<"B::x";};
  virtual void y() {cout<<"B::y";};
};

int main () {
   B b;
   A *ap = &b; B *bp = &b;
   b.x (); // prints "B::x"
   b.y (); // prints "B::y"
   bp->x (); // prints "B::x"
   bp->y (); // prints "B::y"
   ap->x (); // prints "A::x"
   ap->y (); // prints "B::y"
   return 0;
};
```

- Only matter with pointer or reference
  - Calls on object itself resolved statically
  - E.g., `b.y();`
- Look first at pointer/reference type
  - If non-virtual there, resolve statically
    - E.g., `ap->x();`
  - If virtual there, resolve dynamically
    - E.g., `ap->y();`
- Note that virtual keyword need not be repeated in derived classes
  - But it's good style to do so
- Caller can force static resolution of a virtual function via scope operator
  - E.g., `ap->A::y();` prints "`A::y`"

CSE 332: C++ Polymorphism

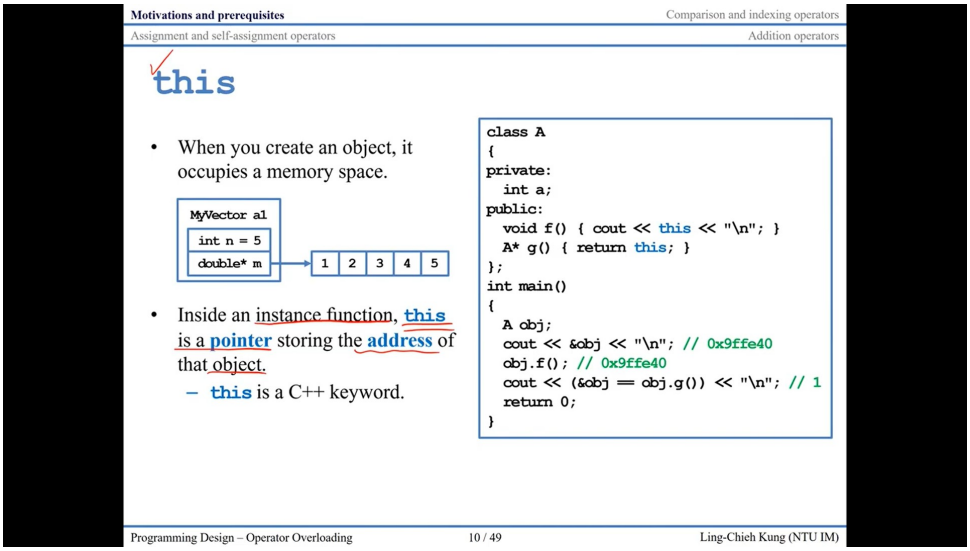
Virtual Functions

```
class A {
public:
  A () {cout<<" A";}
  virtual ~A () {cout<<" ~A";}
  virtual f(int);
};

class B : public A {
public:
  B () :A() {cout<<" B";}
  virtual ~B() {cout<<" ~B";}
  virtual f(int) override; //C++11
};

int main (int, char *[]) {
   // prints "A B"
   A *ap = new B;

   // prints "~B ~A" : would only
   // print "~A" if non-virtual
   delete ap;

   return 0;
};
```

- Used to support polymorphism with pointers and references
- Declared virtual in a base class
- Can override in derived class
  - Overriding only happens when signatures are the same
  - Otherwise it just *overloads* the function or operator name
    - More about overloading next lecture
- Ensures derived class function definition is resolved dynamically
  - E.g., that destructors farther down the hierarchy get called
- Use **`final`** (C++11) to prevent overriding of a virtual method
- Use **`override`** (C++11) in derived class to ensure that the signatures match (error if not)

CSE 332: C++ Polymorphism

this



Constant Object

Constant objects

- A constant object cannot invoke a function that modifies its instance variables.
  - In C++, functions that may be invoked by a constant object must be declared as a **constant instance function**.
- For a constant instance function:
  - It can be called by non-constant objects.
  - It cannot modify any instance variable.
- For a non-constant instance function:
  - It cannot be called by constant objects even if no instance variable is modified.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  MyVector(const MyVector& v);
  ~MyVector();
  void print() const;
};
```


Operator Overload

Overloading an operator

- An operator is overloaded by "implementing a **special instance function**".
  - It cannot be implemented as a static function.
- Let *op* be the operator to be overloaded, the "special instance function" is always named

  `operator`*op*

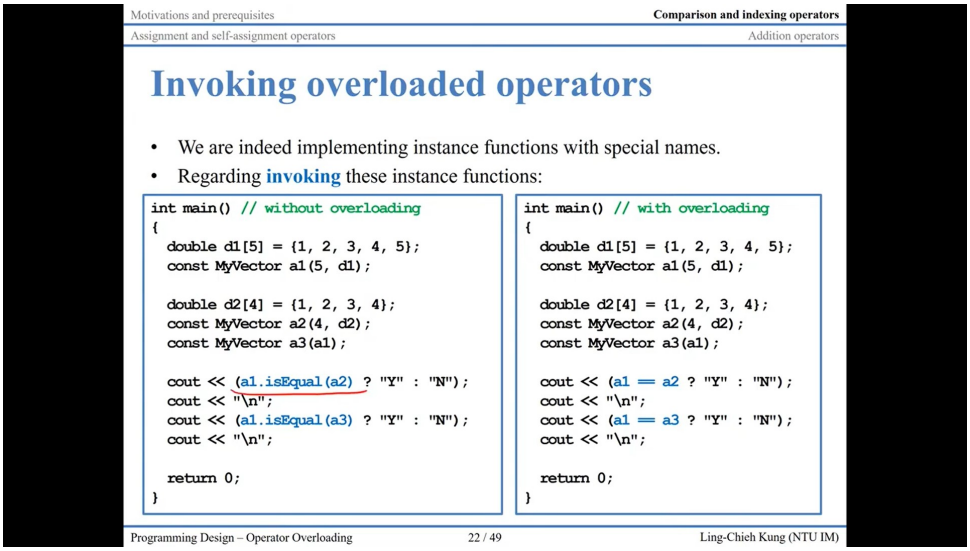
  - The keyword `operator` is used for overloading operators.
- Let's overload `==` for `MyVector`.

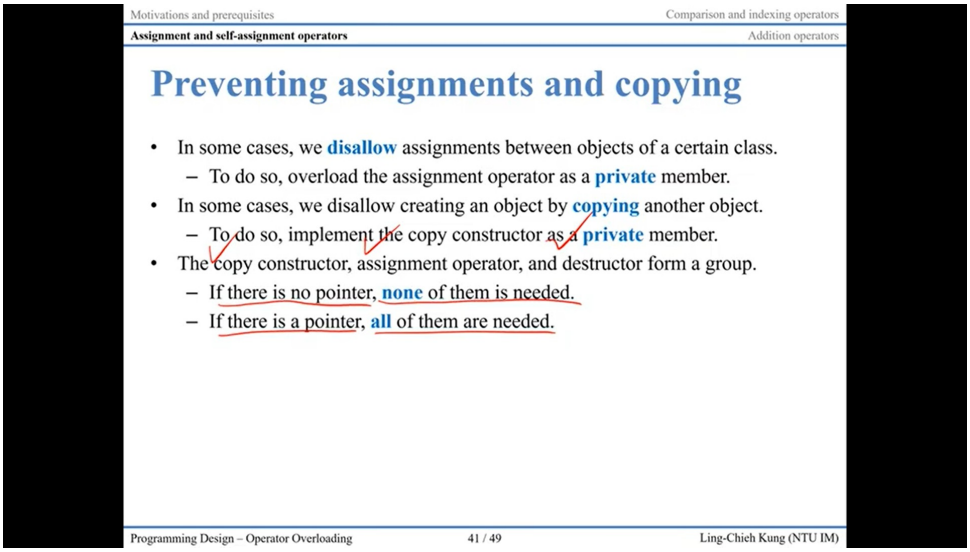
Overloading ==

- To overload `==`, simply do this:

```
class MyVector
{
private:
  int n;
  double* m;
public:
  // others
  bool operator==
    (const MyVector& v) const;
};
```

```
bool MyVector::operator==
  (const MyVector& v) const
{
  if(this->n != v.n)
    return false;
  else {
    for(int i = 0; i < n; i++) {
      if(this->m[i] != v.m[i])
        return false;
    }
  }
  return true;
}
```





Assignment Operator

SYMBOLS	MEANINGS
"="	**EQUAL:** it assigns value to the variable
"+="	**ADD EQUAL:** it means it updated the value of the variable after addition a+=5 (a=a+5)
"-="	**SUBTRACT EQUAL:** it means it updated the value of the variable after subtraction a-=5 (a=a-5)
"*="	**MULTIPLY EQUAL:** it means it updated the value of the variable after multiplication a*=5 (a=a*5)
"/="	**DIVISION EQUAL:** it means it updated the value of the variable after division a/=5 (a=a/5)
"%="	**REMAINDER EQUAL:** it means it updated the value of the variable after remainder a%=5 (a=a%5)
"&="	**AND EQUAL:** it means both conditions are necessary
"|="	**NOT EQUAL:** it means condition are not equal

String Class

String Class

- The **String** class contains several methods that can be used to perform operations on a string.

Method Name	Return Type	Description	Usage
compareTo(String t)	int	Compares two strings	string1.compareTo(string2);
length()	int	Gets length of string	string1.length();
charAt(int i)	char	Finds ith character	string1.charAt(3);
equals(String t)	Boolean	Are two strings equal?	string1.equals(string2);
substring(int i, int j)	String	Finds substring from ith character to character before jth one	String1.substring(0, 2)

Correspondence between the C library and the C++ string Class

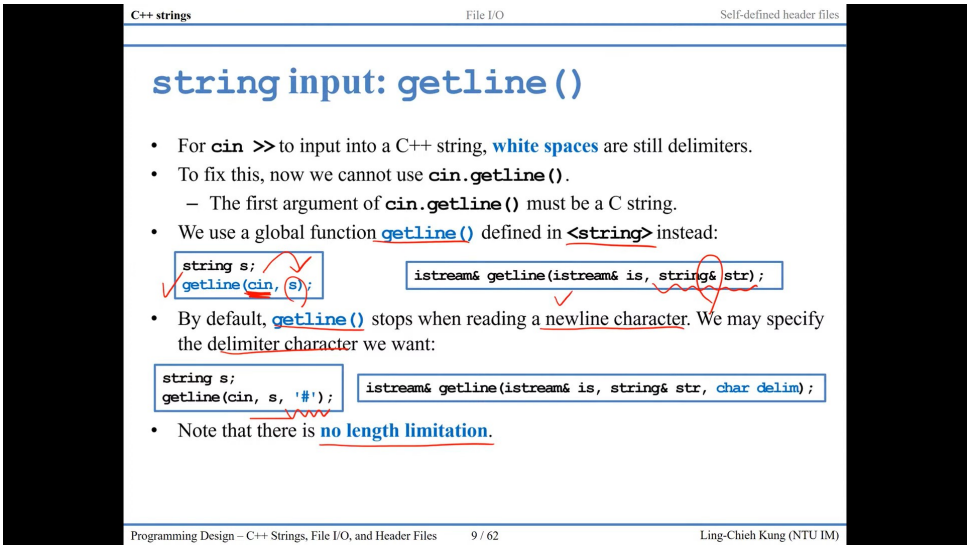
C Library Functions	C++ string operators/methods
strcpy	= (the assignment operator)
strcat	+= (assign+concat operator)
strcmp	= =, !=, <, >, <=, >=
strchr, strstr	.find( ) method
strrchr	.rfind( ) method
strlen	.size( ) or .length( ) methods

CS 103 26

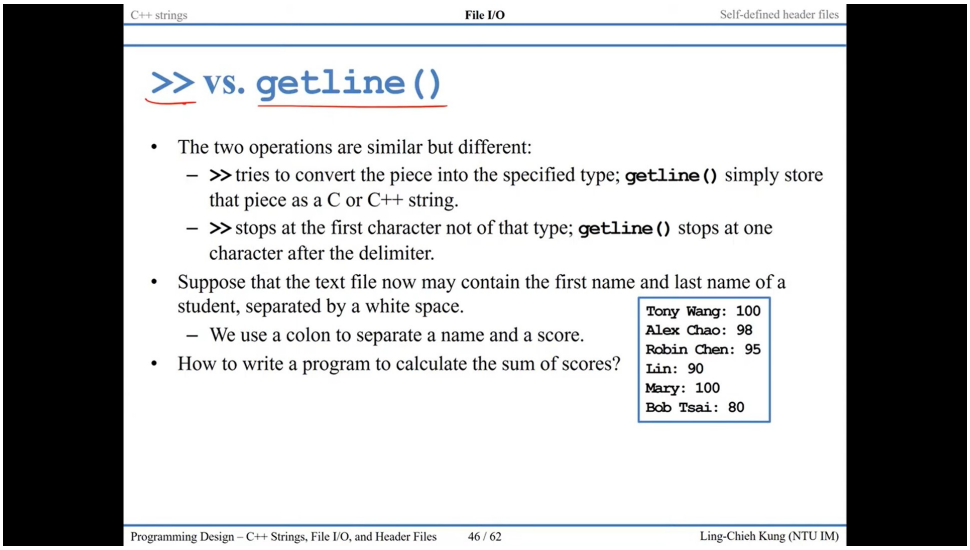
String Processing Function

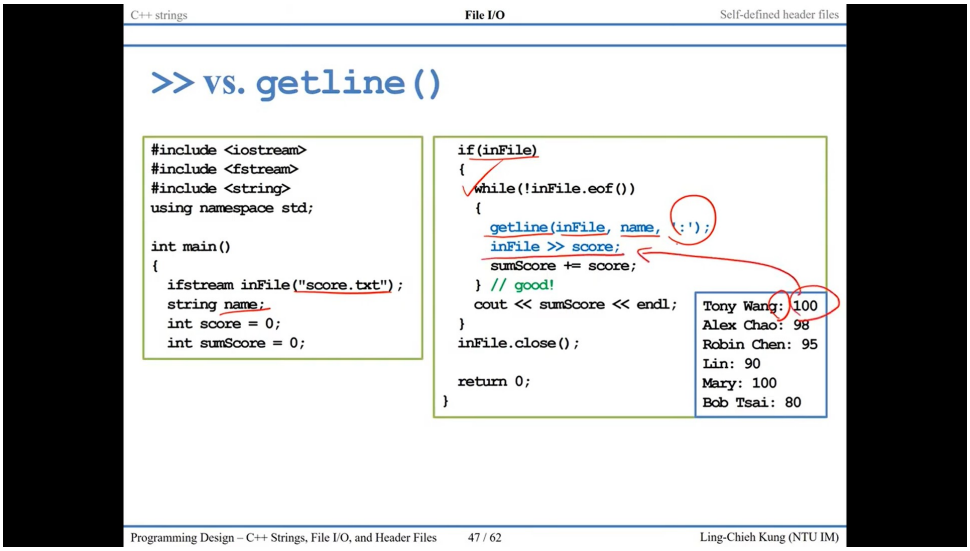
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

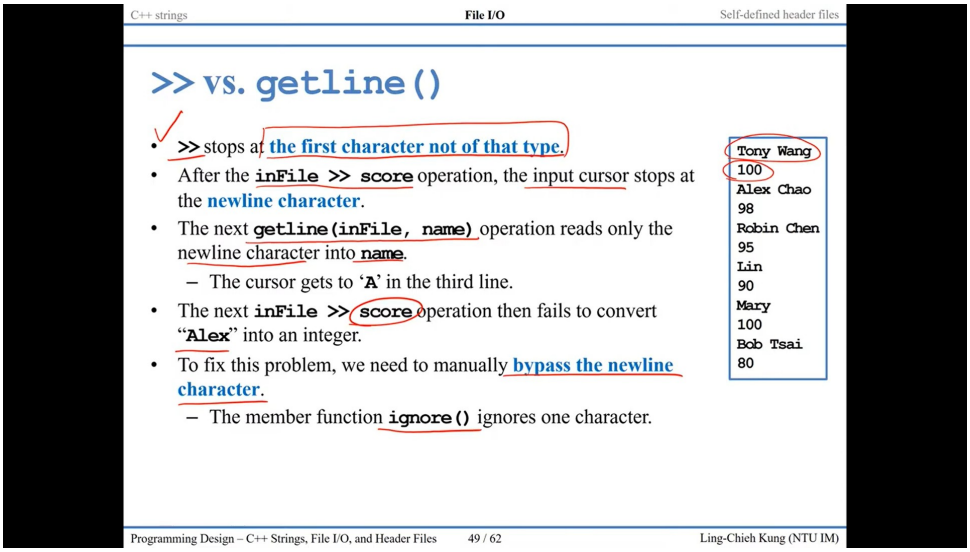
getline()

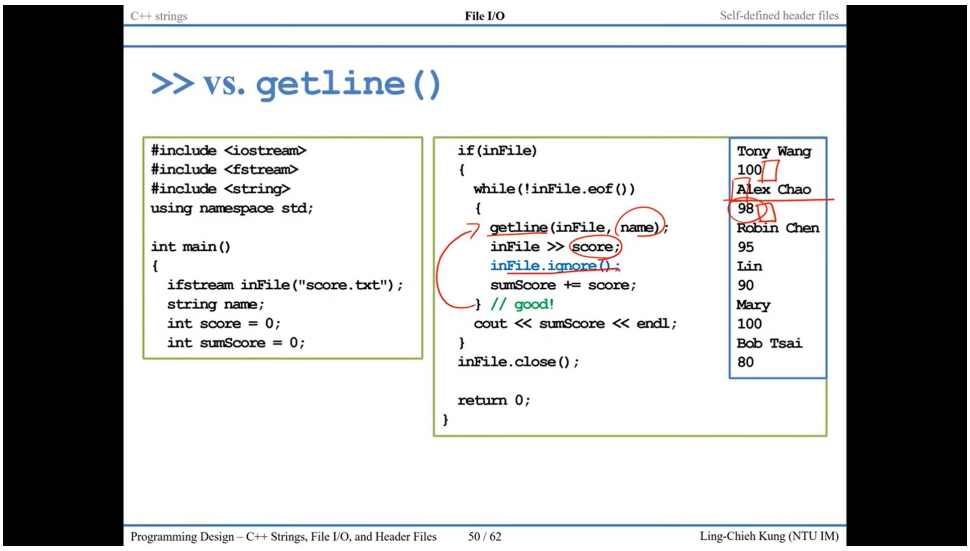


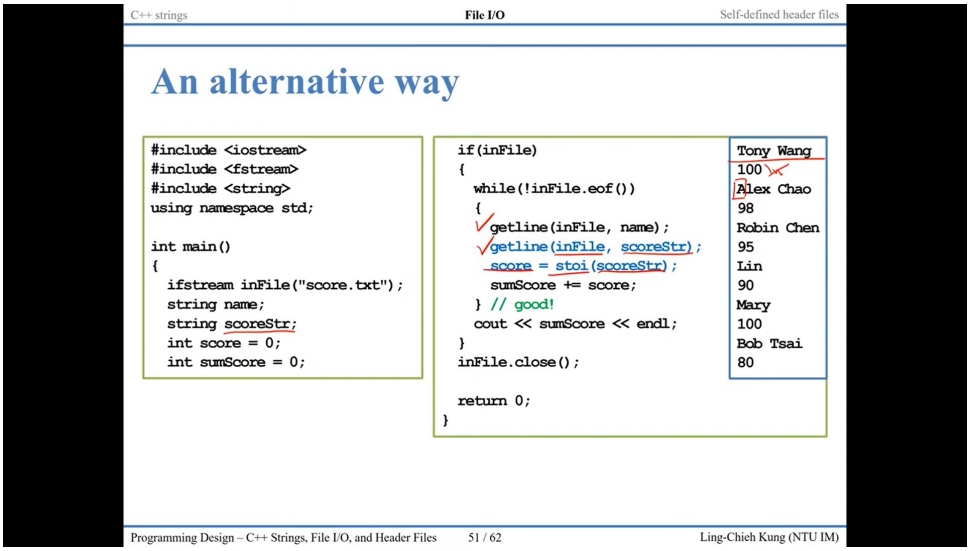
getline() vs >>



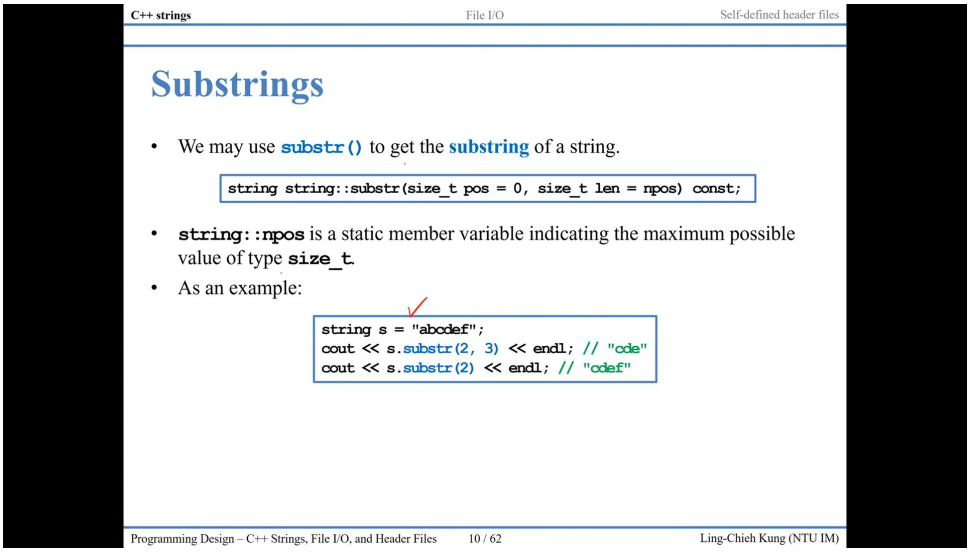

C++ strings
File I/O
Self-defined header files
>> vs. getline()
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
ifstream inFile("score.txt");
string name;
int score = 0;
int sumScore = 0;
if(inFile)
{
while(!inFile.eof())
{
getline(inFile, name, ':');
inFile >> score;
sumScore += score;
} // good!
cout << sumScore << endl;
}
inFile.close();
return 0;
}
Tony Wang: 100
Alex Chao: 98
Robin Chen: 95
Lin: 90
Mary: 100
Bob Tsai: 80
Programming Design – C++ Strings, File I/O, and Header Files 47 / 62 Ling-Chieh Kung (NTU IM)

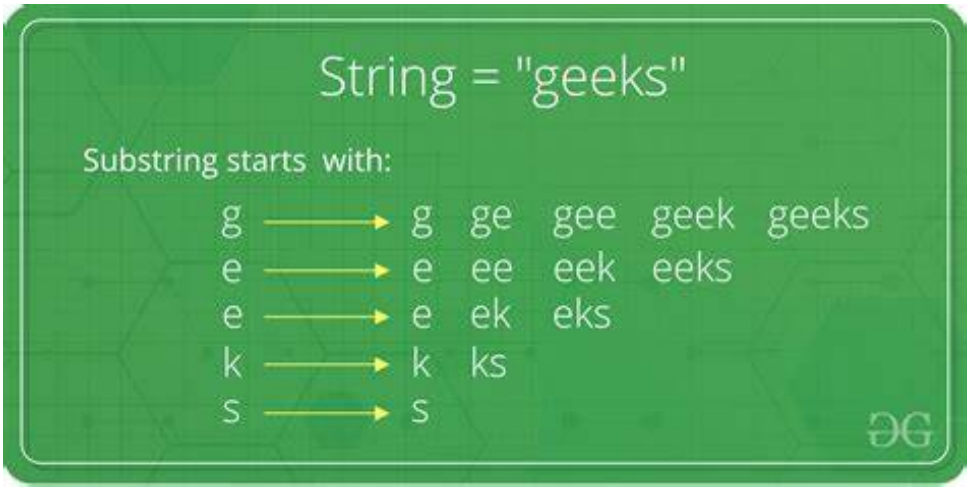

C++ strings
File I/O
Self-defined header files
>> vs. getline()
>> stops at the first character not of that type.
After the inFile >> score operation, the input cursor stops at the newline character.
The next getline(inFile, name) operation reads only the newline character into name.
– The cursor gets to 'A' in the third line.
The next inFile >> score operation then fails to convert "Alex" into an integer.
To fix this problem, we need to manually bypass the newline character.
– The member function ignore() ignores one character.
Tony Wang
100
Alex Chao
98
Robin Chen
95
Lin
90
Mary
100
Bob Tsai
80
Programming Design – C++ Strings, File I/O, and Header Files 49 / 62 Ling-Chieh Kung (NTU IM)


C++ strings
File I/O
Self-defined header files
>> vs. getline()
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
ifstream inFile("score.txt");
string name;
int score = 0;
int sumScore = 0;
if(inFile)
{
while(!inFile.eof())
{
getline(inFile, name);
inFile >> score;
inFile.ignore();
sumScore += score;
} // good!
cout << sumScore << endl;
}
inFile.close();
return 0;
}
Tony Wang
100
Alex Chao
98
Robin Chen
95
Lin
90
Mary
100
Bob Tsai
80
Programming Design – C++ Strings, File I/O, and Header Files 50 / 62 Ling-Chieh Kung (NTU IM)

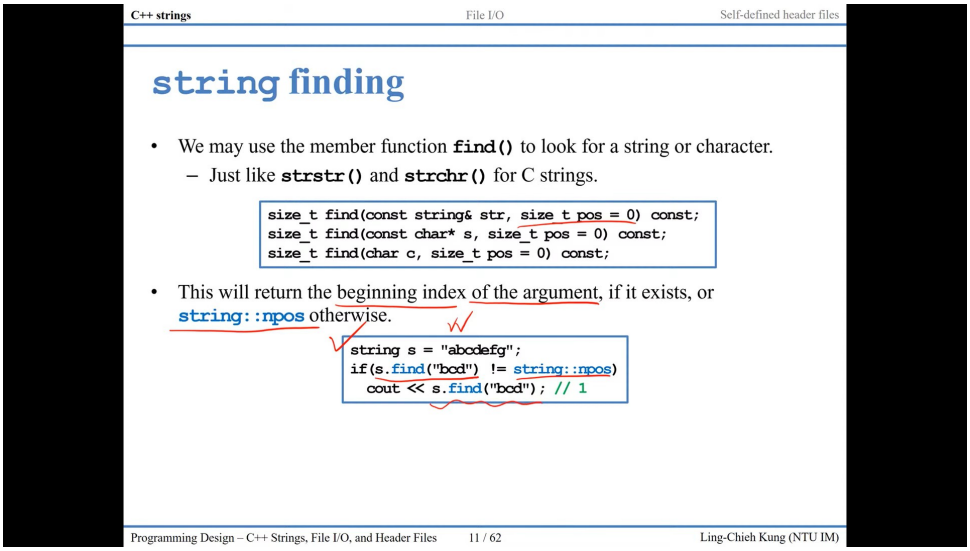

C++ strings
File I/O
Self-defined header files
An alternative way
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
ifstream inFile("score.txt");
string name;
string scoreStr;
int score = 0;
int sumScore = 0;
if(inFile)
{
while(!inFile.eof())
{
getline(inFile, name);
getline(inFile, scoreStr);
score = stoi(scoreStr);
sumScore += score;
} // good!
cout << sumScore << endl;
}
inFile.close();
return 0;
}
Tony Wang
100
Alex Chao
98
Robin Chen
95
Lin
90
Mary
100
Bob Tsai
80
Programming Design – C++ Strings, File I/O, and Header Files 51 / 62 Ling-Chieh Kung (NTU IM)

substr()

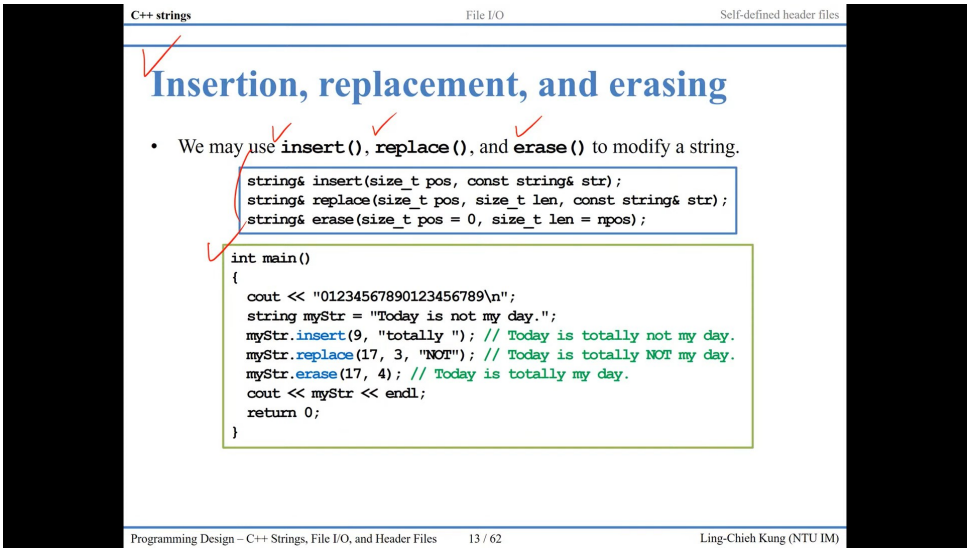

C++ strings
File I/O
Self-defined header files
Substrings
We may use substr() to get the substring of a string.
string string::substr(size_t pos = 0, size_t len = npos) const;
string::npos is a static member variable indicating the maximum possible value of type size_t.
As an example:
string s = "abcdef";
cout << s.substr(2, 3) << endl; // "cde"
cout << s.substr(2) << endl; // "cdef"
Programming Design – C++ Strings, File I/O, and Header Files 10 / 62 Ling-Chieh Kung (NTU IM)


String = "geeks"
Substring starts with:
g → g ge gee geek geeks
e → e ee eek eeks
e → e ek eks
k → k ks
s → s

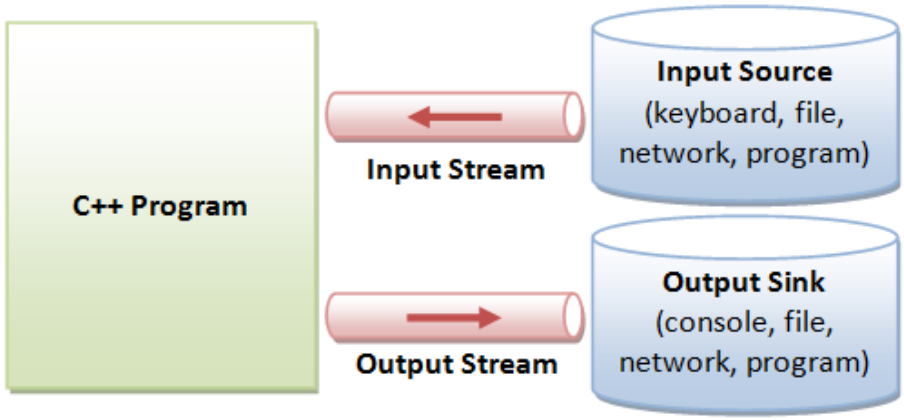
find()



Insertion, Replacement, Erasing



File Stream



Internal Data Formats:

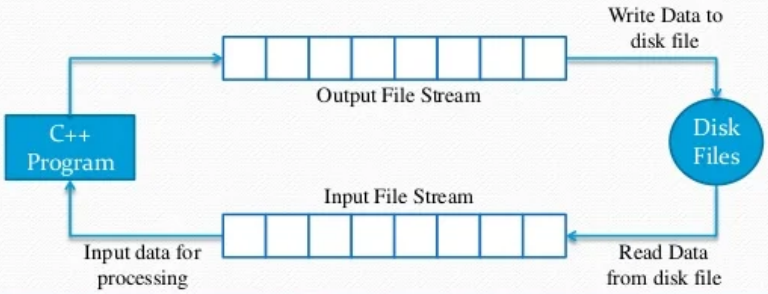
- Text: char, wchar_t
- int, float, double, etc.

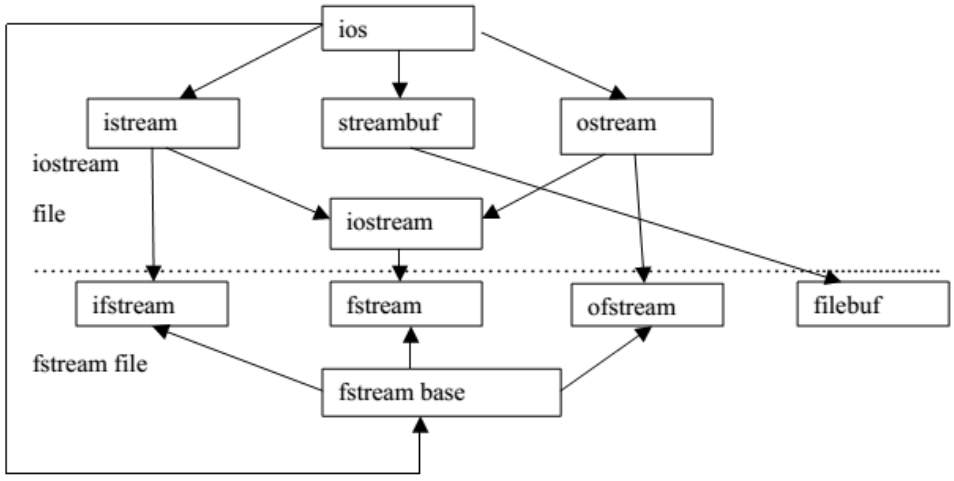
External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

File Streams

- File Streams are of 2 types
  - Input file stream
  - Output file stream





Stream classes for file operations

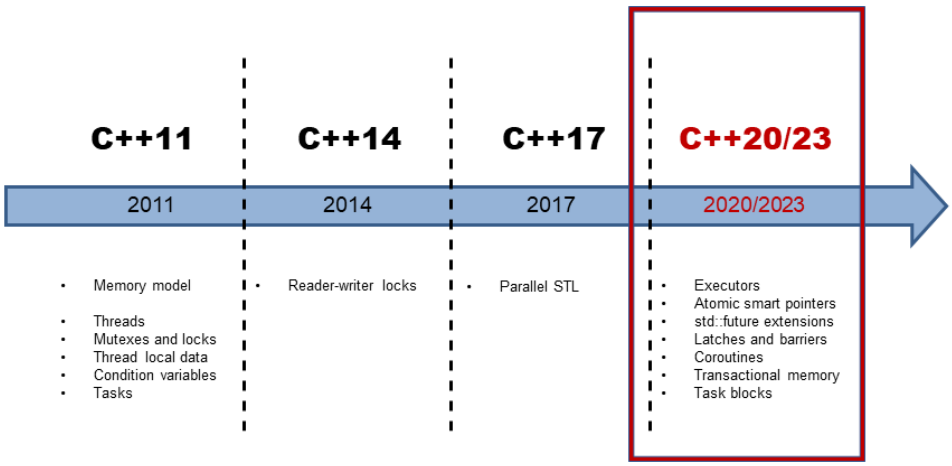
Object Oriented Programming in C++

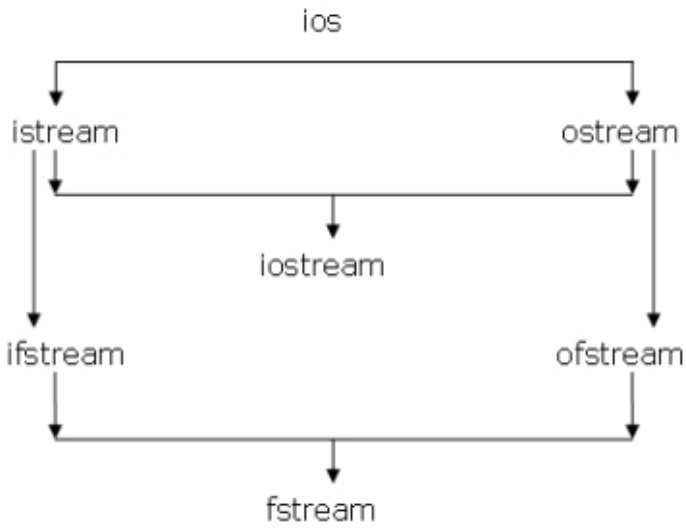
File Opening Mode

File Mode Parameter	Meaning
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::binary	file open in binary mode
ios::in	open file for reading only
ios::out	open file for writing only
ios::nocreate	open fails if the file does not exist
ios::noreplace	open fails if the file already exist
ios::trunc	delete the contents of the file if it exist

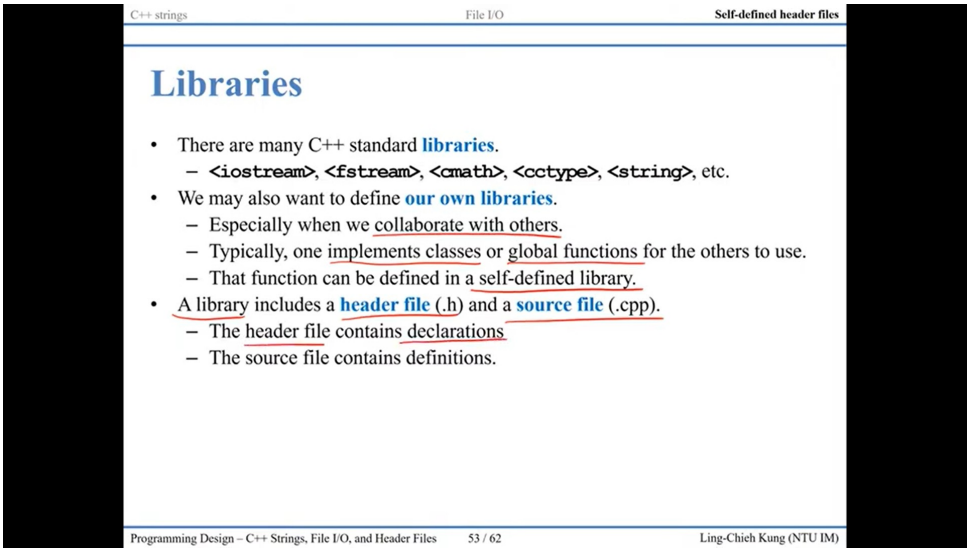
Lecture Slides By Adil Aslam

C++ SL(Strandard Library)





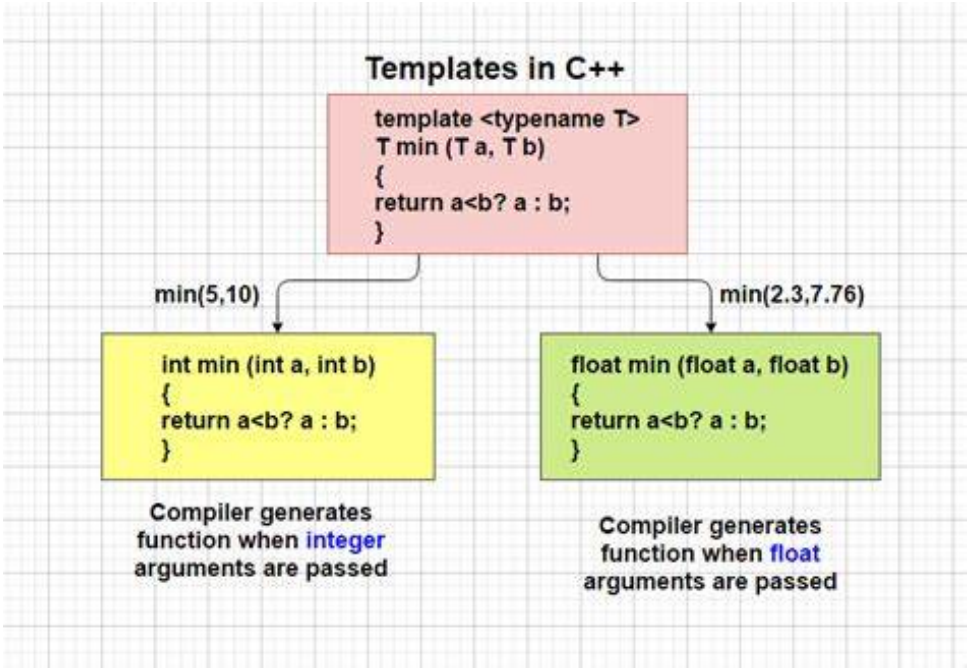
Self-defined Library



Including a header and a source file

- When your main program also wants to include a self-defined source file, the include statement needs not be changed.
  - `#include "myMax.h"`
- We add a source file myMax.cpp.
  - In the source file, we **implement** those functions declared in the header file.
  - The main file names of the header and source files can be different.
- The two source files (main.cpp and myMax.cpp) must be **compiled together**.
  - Each environment has its own way.

Templates



Template declaration

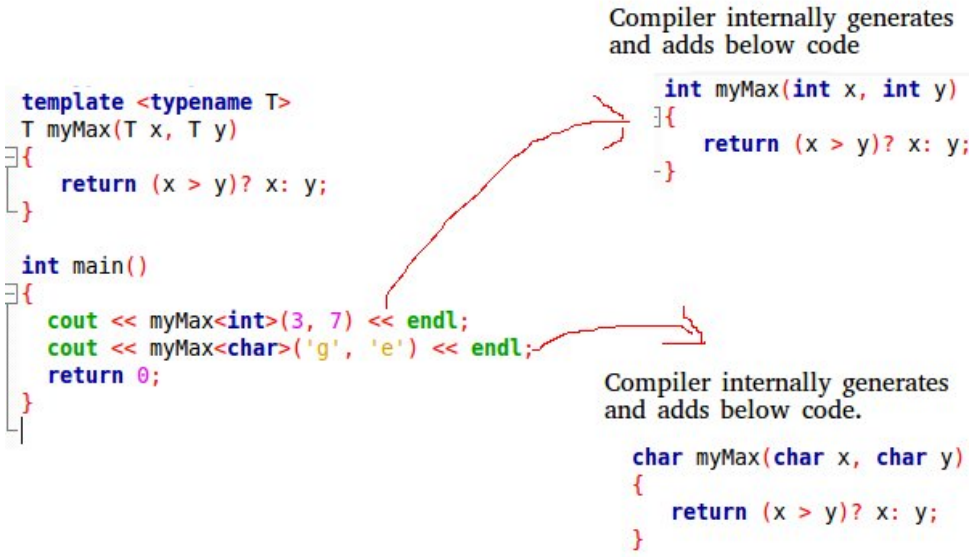
- To declare a type parameter, use the keywords `template` and `typename`.

```
template<typename T>
class TheClassName
{
  // T can be treated as a type inside the class definition block
};
```

  - Some old codes write **`class`** instead of **`typename`**. Both are fine.
- We then do this to all member functions:

```
template<typename T>
T TheClassName<T>::f(T t)
{
  // t is a variable whose type is T
};
```

```
template<typename T>
void TheClassName<T>::f(int i)
{
  // follow the rule even if T is not used
};
```



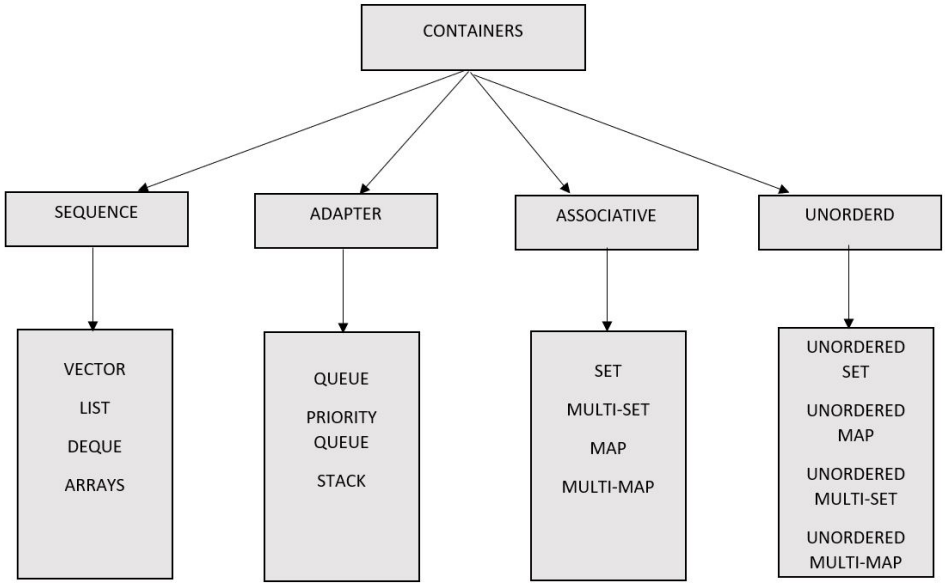
In [ ]:
```
#include <iostream>
#include <cmath>
using namespace std;


template<typename KeyType>
class Character {
protected:
    static const int EXP_LV =100;
    KeyType name;
    int level;
    int exp;
    int power;
    int knowledge;
    int luck;
    void levelUp(int pInc, int kInc, int lInc);
public:
    Character(KeyType n, int lv, int po, int kn, int lu);
    virtual void print();
    virtual void beatMonster(int exp) = 0; //pure virtual function
    KeyType getName();
};


//construct
template<typename KeyType>
Character<KeyType>::Character(KeyType n, int lv, int po, int kn, int lu)
    : name(n), level(lv), exp(pow(lv - 1, 2) * EXP_LV),
      power(po), knowledge(kn), luck(lu) {
}

template<typename KeyType>
void Character<KeyType>::print(){
    cout << this->name
         << ": Level " << this->level
         << " (" << this->exp << "/" << pow(this->level, 2) * EXP_LV
         << ")," << this->power << "-" << this->knowledge
         << "-" << this->luck << endl;
}
```

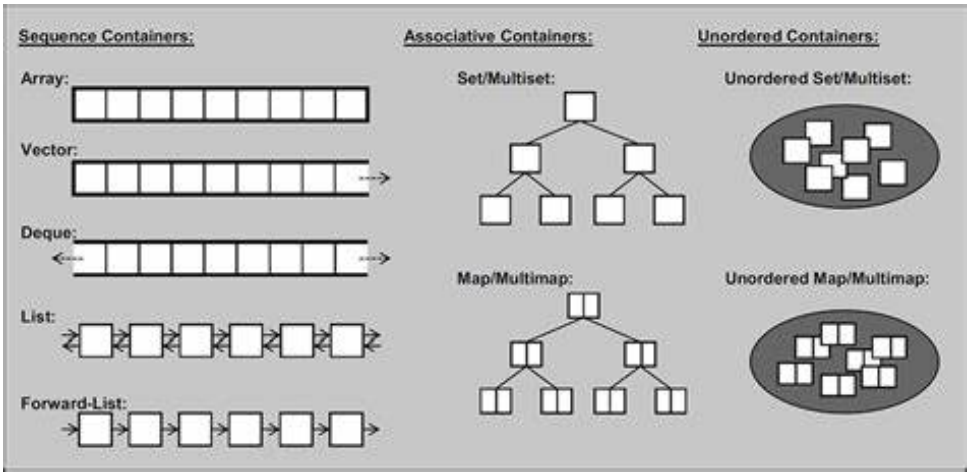
```
template<typename KeyType>
KeyType Character<KeyType>::getName(){
    return this->name;
}

template<typename KeyType>
void Character<KeyType>::beatMonster(int exp){
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * EXP_LV){
        this->levelUp(0, 0, 0);
    }
}

template<typename KeyType>
void Character<KeyType>::levelUp(int pInc, int kInc, int lInc){
    this->level++;
    this->power += pInc;
    this->knowledge += kInc;
    this->luck += lInc;
}


template<typename KeyType>
class Warrior : public Character<KeyType> {
private:
    static const int PO_LV = 10;
    static const int KN_LV = 5;
    static const int LU_LV = 5;
public:
    Warrior(KeyType n, int lv = 0);
    void print();
    void beatMonster(int exp);
};

template<typename KeyType>
Warrior<KeyType>::Warrior(KeyType n, int lv)
        : Character<KeyType>(n, lv, lv * PO_LV, lv * KN_LV, lv * LU_LV) {}

template<typename KeyType>
void Warrior<KeyType>::print() {
    cout << "Warrior: ";
    Character<KeyType>::print();
}

template<typename KeyType>
void Warrior<KeyType>::beatMonster(int exp){ // function overloading
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * Character<KeyType>::EXP_LV){
        this->levelUp(PO_LV, KN_LV, LU_LV);
    }
}



template<typename KeyType>
class Wizard : public Character<KeyType> {
private:
    static const int PO_LV = 4;
    static const int KN_LV = 9;
    static const int LU_LV = 7;
public:
```

```
    Wizard(KeyType n, int lv = 0);
    void print();
    void beatMonster(int exp);
};

template<typename KeyType>
Wizard<KeyType>::Wizard(KeyType n, int lv)
    : Character<KeyType>(n, lv, lv * PO_LV, lv * KN_LV, lv * LU_LV) {}

template<typename KeyType>
void Wizard<KeyType>::print() {
    cout << "Wizard: ";
    Character<KeyType>::print();
}

template<typename KeyType>
void Wizard<KeyType>::beatMonster(int exp){ // function overloading
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * Character<KeyType>::EXP_LV){
        this->levelUp(PO_LV, KN_LV, LU_LV);
    }
}


template<typename KeyType>
class Team {
private:
    int memberCount;
    Character<KeyType>* member[10];
public:
    Team();
    ~Team();
    void addWarrior(KeyType name, int lv);
    void addWizard(KeyType name, int lv);
    void memberBeatMonster(KeyType name, int exp);
    void printMember(KeyType name);
};

template<typename KeyType>
Team<KeyType>::Team(){
    memberCount = 0;
    for (int i = 0; i < 10; i++){
        member[i] = nullptr;
    }
}

template<typename KeyType>
Team<KeyType>::~Team(){
    for (int i = 0; i < memberCount; i++){
        delete member[i];
    }
}

template<typename KeyType>
void Team<KeyType>::addWarrior(KeyType name, int lv){
    if (memberCount < 10){
        member[memberCount] = new Warrior<KeyType>(name, lv);
        memberCount++;
    }
}
```

```cpp
template<typename KeyType>
void Team<KeyType>::addWizard(KeyType name, int lv){
    if (memberCount < 10){
        member[memberCount] = new Wizard<KeyType>(name, lv);
        memberCount++;
    }
}

template<typename KeyType>
void Team<KeyType>::memberBeatMonster(KeyType name, int exp){
    for(int i = 0; i < memberCount; i++){
        if (member[i]->getName() == name){
            member[i]->beatMonster(exp);
            break;
        }
    }
}

template<typename KeyType>
void Team<KeyType>::printMember(KeyType name){
    for(int i = 0; i < memberCount; i++){
        if(member[i]->getName() == name){
            member[i]->print();
            break;
        }
    }
}



int main(){
    cout << "Welcome to Learning C++ Data Structure and Algorithm" << endl;
    cout << "Topic: OOP - Class" << endl << endl;

    Team<string> t1;
    t1.addWarrior("Alice", 1);
    t1.memberBeatMonster("Alice", 10000);
    t1.addWizard("Bob", 2);
    t1.printMember("Alice");

    Team<int> t2;
    t2.addWarrior(1, 1);
    t2.memberBeatMonster(1, 10000);
    t2.addWizard(2, 2);
    t2.printMember(1);


    return 0;
}
```

C++ STL(Strandard Template Library)





Intro to the C++ Standard Template Library (STL)

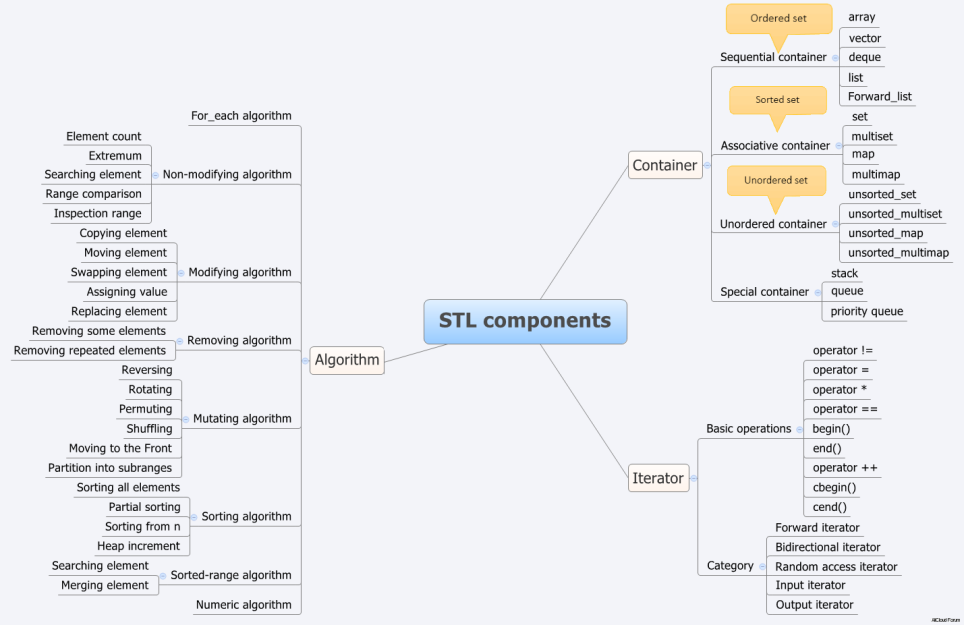
- The STL is a collection of related software elements
  - Containers
    - Data structures: store values according to a specific organization
  - Iterators
    - Variables used to give flexible access to the values in a container
  - Algorithms
    - Functions that use iterators to access values in containers
    - Perform computations that modify values, or creates new ones
  - Function objects
    - Encapsulate a function as an object, use to modify an algorithm
- The STL makes use of most of what we've covered
  - Extensive use of function and class templates, concepts
- The STL makes use of several new ideas too
  - typedefs, traits, and associated types

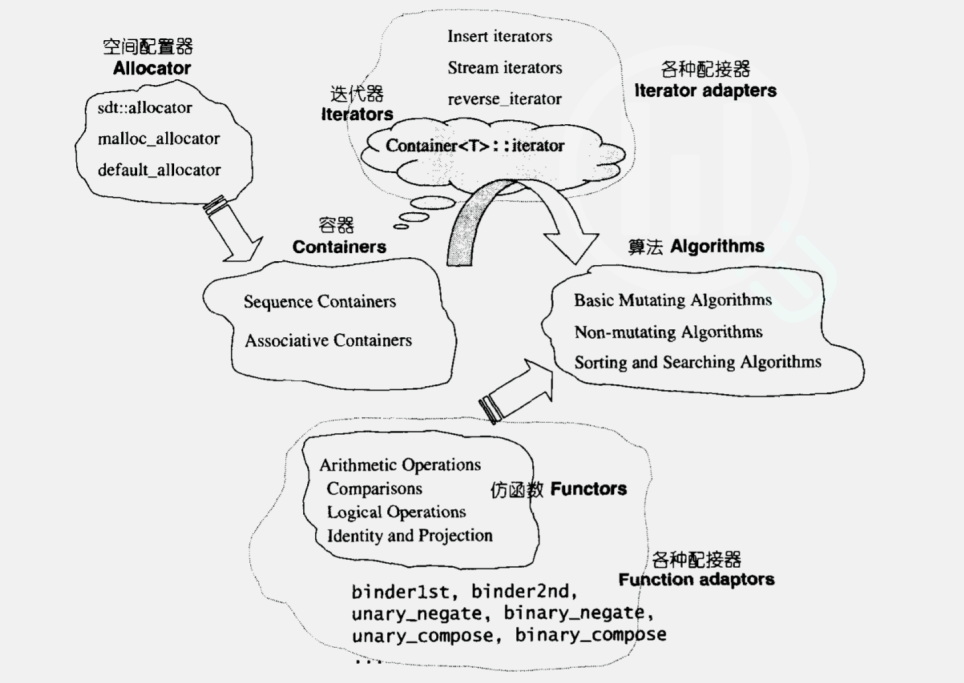
CSE 332: C++ STL containers

Four STL Components

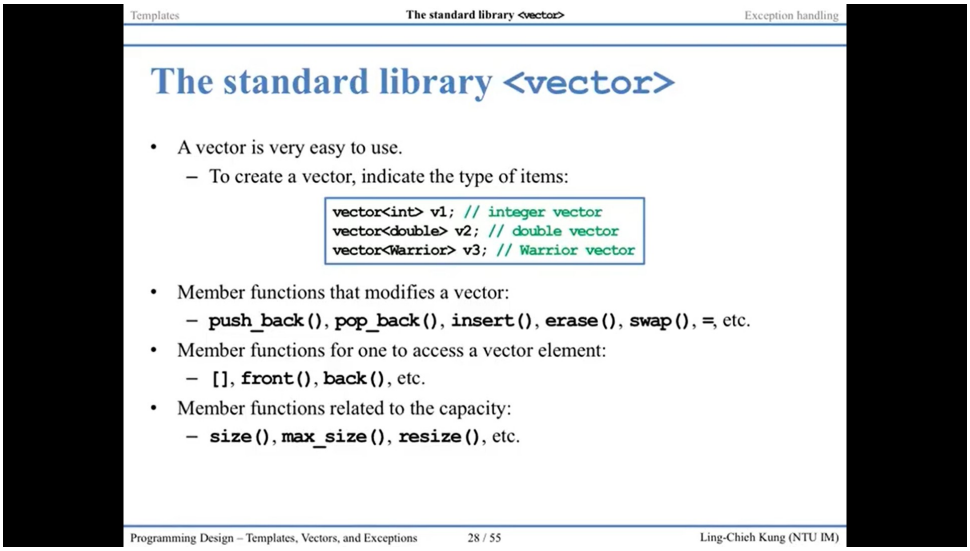
- **Containers**: used to manage collections of objects of a certain kind.
- **Algorithms**: act on containers.
  - E.g. initialization, sorting, searching….
- **Iterators**: used to step through the elements of collections of objects.
- **Functors** are objects that can be treated as though they are a function or function pointer.

2018/9/8

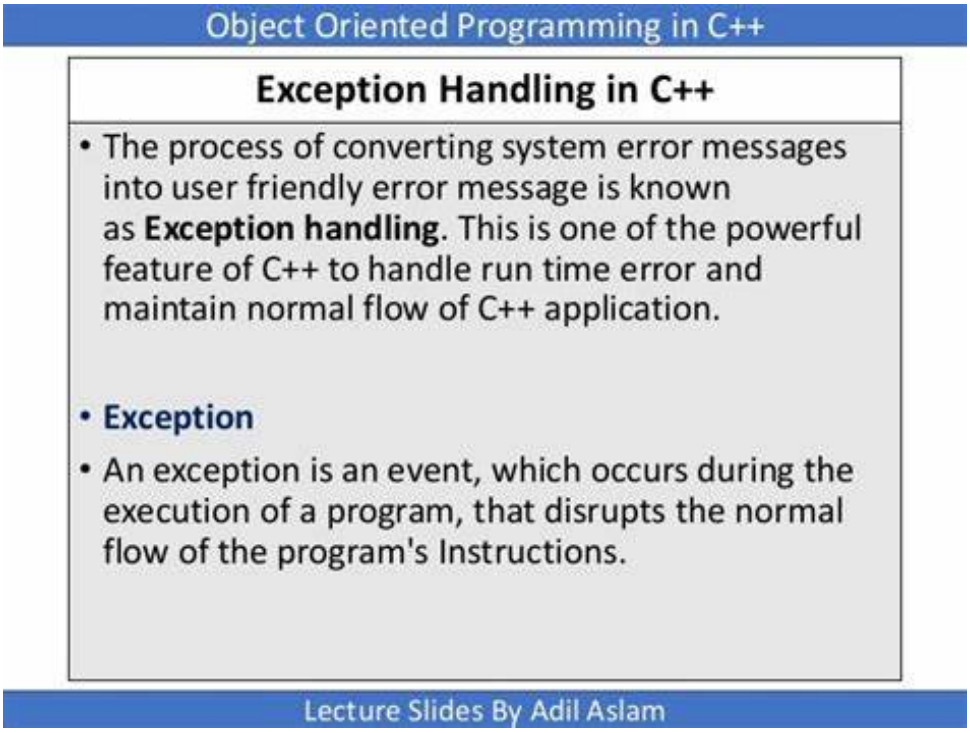




Vector



Exception Handling



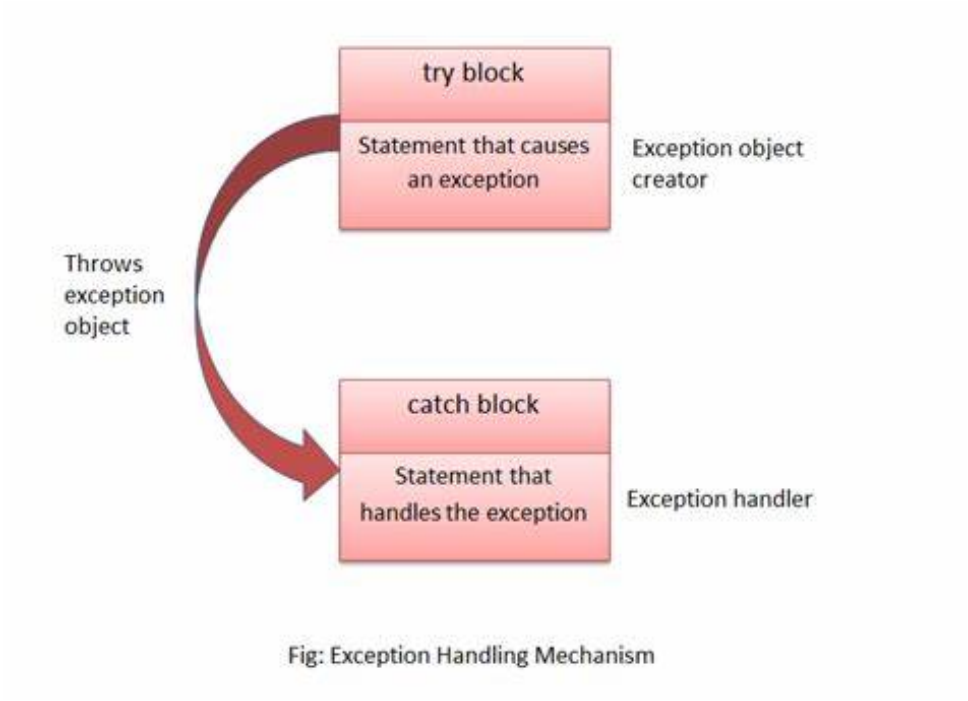
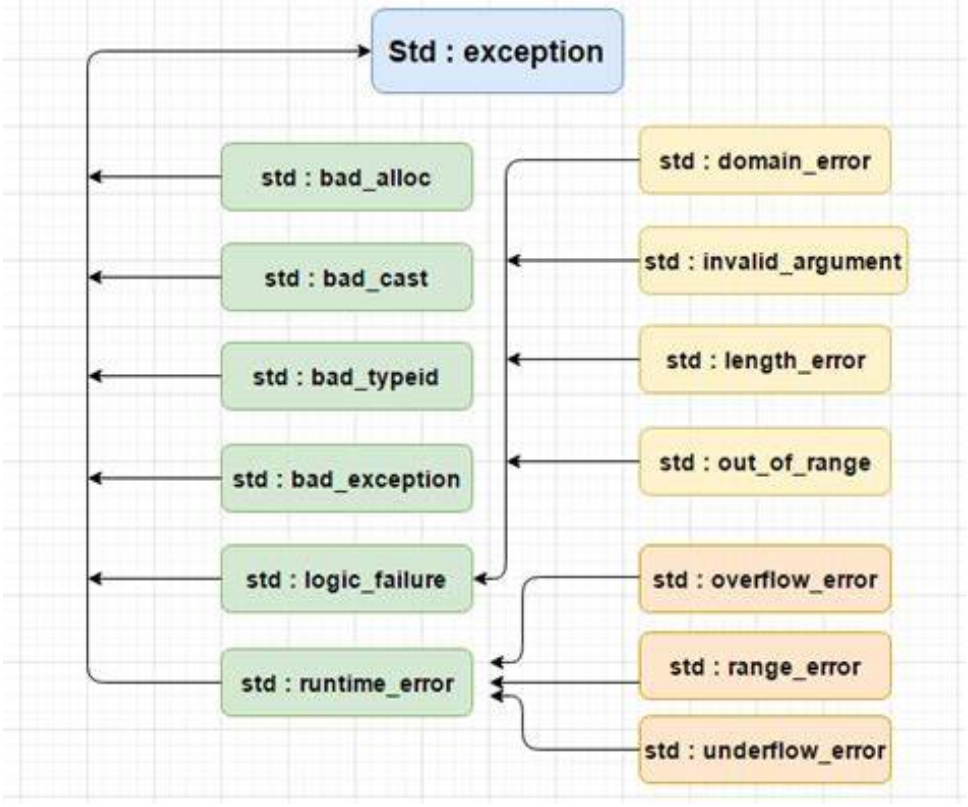


Fig: Exception Handling Mechanism



Standard exception classes

- In the C++ standard library, we have the following standard exception classes:
  - Inheritance and polymorphism!

```
try {
  g(s, i);
}
// this also works
catch(logic_error e) {
  cout << "...\n";
}
```

  - Include **<stdexcept>** to use them.

```
exception
    logic_error
        domain_error
        invalid_argument
        length_error
        out_of_range
    runtime_error
        range_error
        overflow_error
        underflow_error
```


Throwing an exception

- Let the client **catch** the exception:

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

void f(int a[], int n) throw(logic_error)
{
  int i = 0;
  cin >> i;
  if(i < 0 || i > n)
    throw logic_error("...");
  a[i] = 1;
}
```

```
int main()
{
  int a[5] = {0};
  try {
    f(a, 5);
  }
  catch(logic_error e) {
    cout << e.what();
  }
  for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
  return 0;
}
```

- **what()** returns the message generated when throwing an exception.

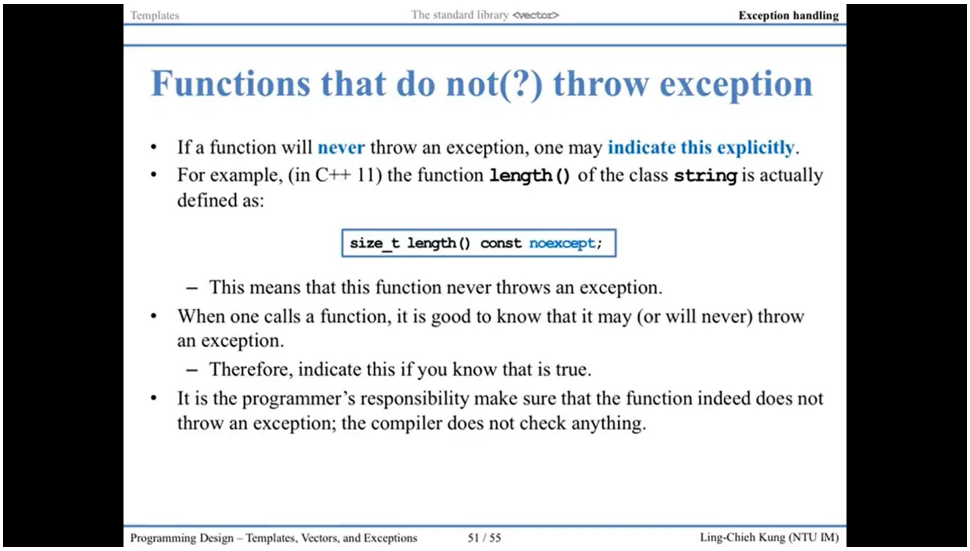
Modifying the function header

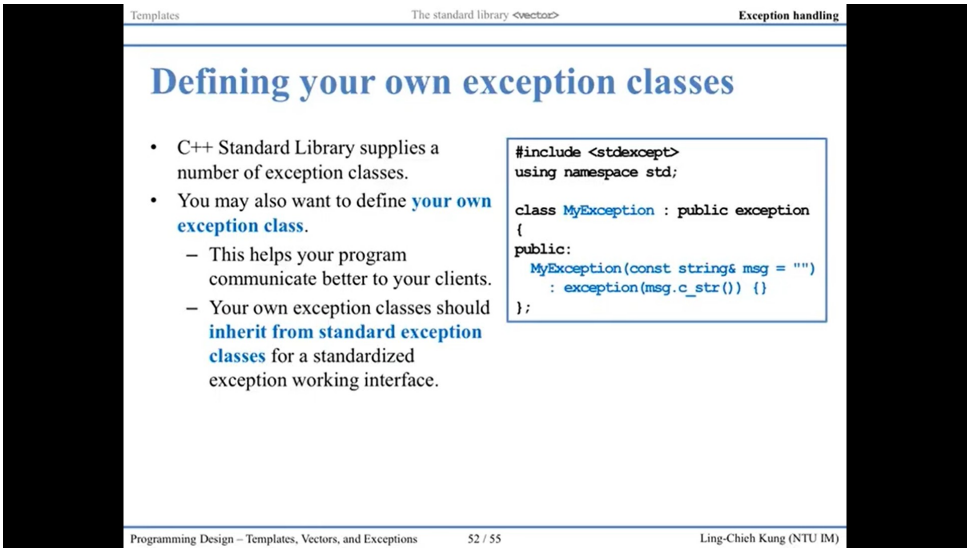
- Functions that throw an exception may have a **throw clause** at the end of their headers.
  - This restricts the exceptions that a function can throw.
  - Omitting a **throw** clause allows a function to throw any exception.
- To allow multiple types of exceptions:

```
void f(int a[], int n) throw(type1, type2)
```

- The documentation of a function (or method) should indicate any exception it might throw.

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

void f(int a[], int n)
  throw(logic_error)
{
  int i = 0;
  cin >> i;
  if(i < 0 || i > n)
    throw logic_error("...");
  a[i] = 1;
}
```


Templates
The standard library <vector>
Exception handling
Functions that do not(?) throw exception
If a function will never throw an exception, one may indicate this explicitly.
For example, (in C++ 11) the function length() of the class string is actually defined as:
size_t length() const noexcept;
This means that this function never throws an exception.
When one calls a function, it is good to know that it may (or will never) throw an exception.
Therefore, indicate this if you know that is true.
It is the programmer's responsibility make sure that the function indeed does not throw an exception; the compiler does not check anything.
Programming Design – Templates, Vectors, and Exceptions 51 / 55 Ling-Chieh Kung (NTU IM)


Templates
The standard library <vector>
Exception handling
Defining your own exception classes
C++ Standard Library supplies a number of exception classes.
You may also want to define your own exception class.
This helps your program communicate better to your clients.
Your own exception classes should inherit from standard exception classes for a standardized exception working interface.
#include <stdexcept>
using namespace std;
class MyException : public exception
{
public:
MyException(const string& msg = "")
: exception(msg.c_str()) {}
};
Programming Design – Templates, Vectors, and Exceptions 52 / 55 Ling-Chieh Kung (NTU IM)

-- Memo End --