

# Programming Language - C & C++

## Escape Sequence

**Table 3.2. C++ Escape Sequence Codes**

Character Name	ASCII Symbol	C++ Code	ASCII Decimal Code	ASCII Hex Code
Newline	NL (LF)	\n	10	0xA
Horizontal tab	HT	\t	9	0x9
Vertical tab	VT	\v	11	0xB
Backspace	BS	\b	8	0x8
Carriage return	CR	\r	13	0xD
Alert	BEL	\a	7	0x7
Backslash	\	\\	92	0x5C
Question mark	?	\?	63	0x3F
Single quote	'	\'	39	0x27
Double quote	"	\"	34	0x22

ASCII control characters			ASCII printable characters			Extended ASCII characters										
00	NULL	(Null character)	32	space	64	@	96	`	128	Ç	160	á	192	Ł	224	Ó
01	SOH	(Start of Header)	33	!	65	A	97	a	129	ü	161	í	193	ł	225	ô
02	STX	(Start of Text)	34	"	66	B	98	b	130	é	162	ó	194	Ł	226	õ
03	ETX	(End of Text)	35	#	67	C	99	c	131	â	163	ú	195	ł	227	ö
04	EOT	(End of Trans.)	36	\$	68	D	100	d	132	ä	164	ü	196	Ł	228	ø
05	ENQ	(Enquiry)	37	%	69	E	101	e	133	å	165	ñ	197	ł	229	ó
06	ACK	(Acknowledgement)	38	&	70	F	102	f	134	ä	166	*	198	Ł	230	µ
07	BEL	(Bell)	39	'	71	G	103	g	135	ç	167	°	199	À	231	þ
08	BS	(Backspace)	40	(	72	H	104	h	136	ê	168	¿	200	É	232	Ë
09	HT	(Horizontal Tab)	41	)	73	I	105	i	137	ë	169	®	201	Ê	233	Û
10	LF	(Line feed)	42	*	74	J	106	j	138	è	170	¬	202	Ë	234	Ü
11	VT	(Vertical Tab)	43	+	75	K	107	k	139	í	171	½	203	Ë	235	Ý
12	FF	(Form feed)	44	,	76	L	108	l	140	î	172	¼	204	Ë	236	ÿ
13	CR	(Carriage return)	45	-	77	M	109	m	141	ï	173	¸	205	¼	237	ÿ
14	SO	(Shift Out)	46	.	78	N	110	n	142	Ä	174	«	206	½	238	·
15	SI	(Shift In)	47	/	79	O	111	o	143	Å	175	»	207	¾	239	·
16	DLE	(Data link escape)	48	0	80	P	112	p	144	Æ	176	⋮	208	ø	240	≡
17	DC1	(Device control 1)	49	1	81	Q	113	q	145	æ	177	⋮	209	ð	241	±
18	DC2	(Device control 2)	50	2	82	R	114	r	146	Æ	178	⋮	210	é	242	±
19	DC3	(Device control 3)	51	3	83	S	115	s	147	ø	179	⋮	211	Ê	243	¾
20	DC4	(Device control 4)	52	4	84	T	116	t	148	ö	180	⋮	212	Ë	244	¾
21	NAK	(Negative acknowl.)	53	5	85	U	117	u	149	ö	181	À	213	ì	245	§
22	SYN	(Synchronous idle)	54	6	86	V	118	v	150	ü	182	Á	214	í	246	÷
23	ETB	(End of trans. block)	55	7	87	W	119	w	151	ü	183	Â	215	î	247	·
24	CAN	(Cancel)	56	8	88	X	120	x	152	ÿ	184	©	216	ï	248	·
25	EM	(End of medium)	57	9	89	Y	121	y	153	Û	185	⋮	217	ÿ	249	·
26	SUB	(Substitute)	58	:	90	Z	122	z	154	Ü	186	⋮	218	ÿ	250	·
27	ESC	(Escape)	59	;	91	[	123	{	155	ø	187	⋮	219	ÿ	251	·
28	FS	(File separator)	60	<	92	\	124		156	£	188	⋮	220	ÿ	252	·
29	GS	(Group separator)	61	=	93	]	125	}	157	Ø	189	¢	221	ÿ	253	·
30	RS	(Record separator)	62	>	94	^	126	~	158	×	190	¥	222	ÿ	254	·
31	US	(Unit separator)	63	?	95	_			159	f	191	¬	223	ÿ	255	nbsp

## Variables

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

int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;

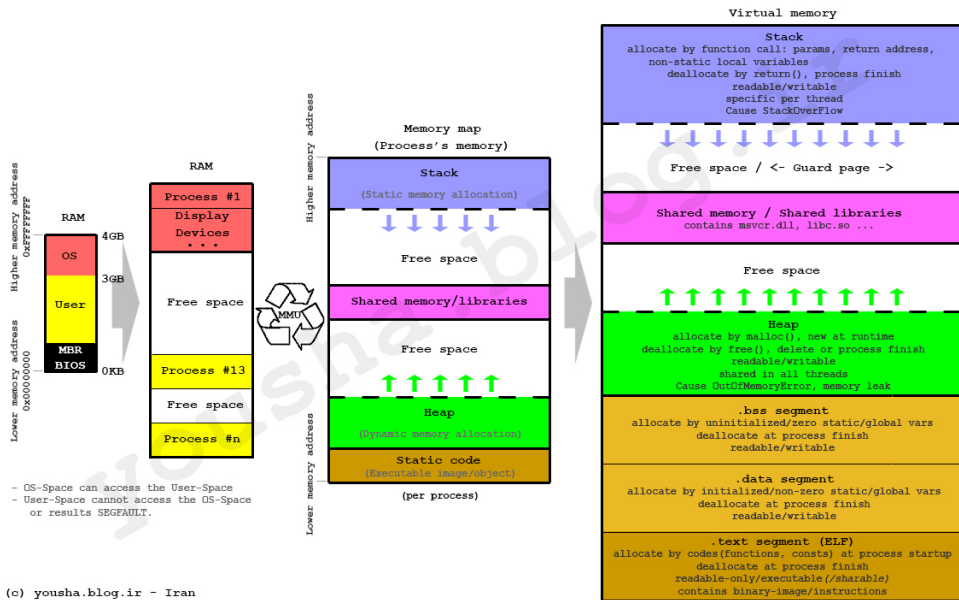
int main ()
{
    unsigned short Age;
    float ANumber, AnotherOne;

    cout << "Enter your age:" ;
    cin >> Age;
    ...
}
```

Global variables

Local variables

Instructions



(c) yousha.blog.ir - Iran

Computer programming Basic structure and cout Variable declaration and cin

the if and while statements Formating a C++ program

## 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.

Address	Identifier	Value
0x22fd4c	myInt	???

Memory

Programming Design - Introduction 21 / 62 Ling-Chieh Kung (NTU IM)

Computer programming Basic structure and cout Variable declaration and cin  
the if and while statements Formatting a C++ program

## 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)!

Address	Identifier	Value
0x20c648	yourInt	5
0x22f44c	myInt	10

Memory

Programming Design - Introduction 24 / 62 Ling-Chieh Kung (NTU IM)

Computer programming Basic structure and cout Variable declaration and cin  
the if and while statements Formatting a C++ program

## When we execute this program

```

#include <iostream>
using namespace std;

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

    return 0;
}
    
```

Address	Identifier	Value
(3) 0x20c630	(no name)	17
(1) 0x20c648	num1	13
(2) 0x22f44c	num2	4

Console

Memory

Programming Design - Introduction 30 / 62 Ling-Chieh Kung (NTU IM)

Computer programming Basic structure and cout Variable declaration and cin  
the if and while statements Formatting a C++ program

## When we execute this program

```

#include <iostream>
using namespace std;

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

    return 0;
}
    
```

Address	Identifier	Value
(3)		
(1) 0x20c648	num1	13
(2) 0x22f44c	num2	4

(4) 17

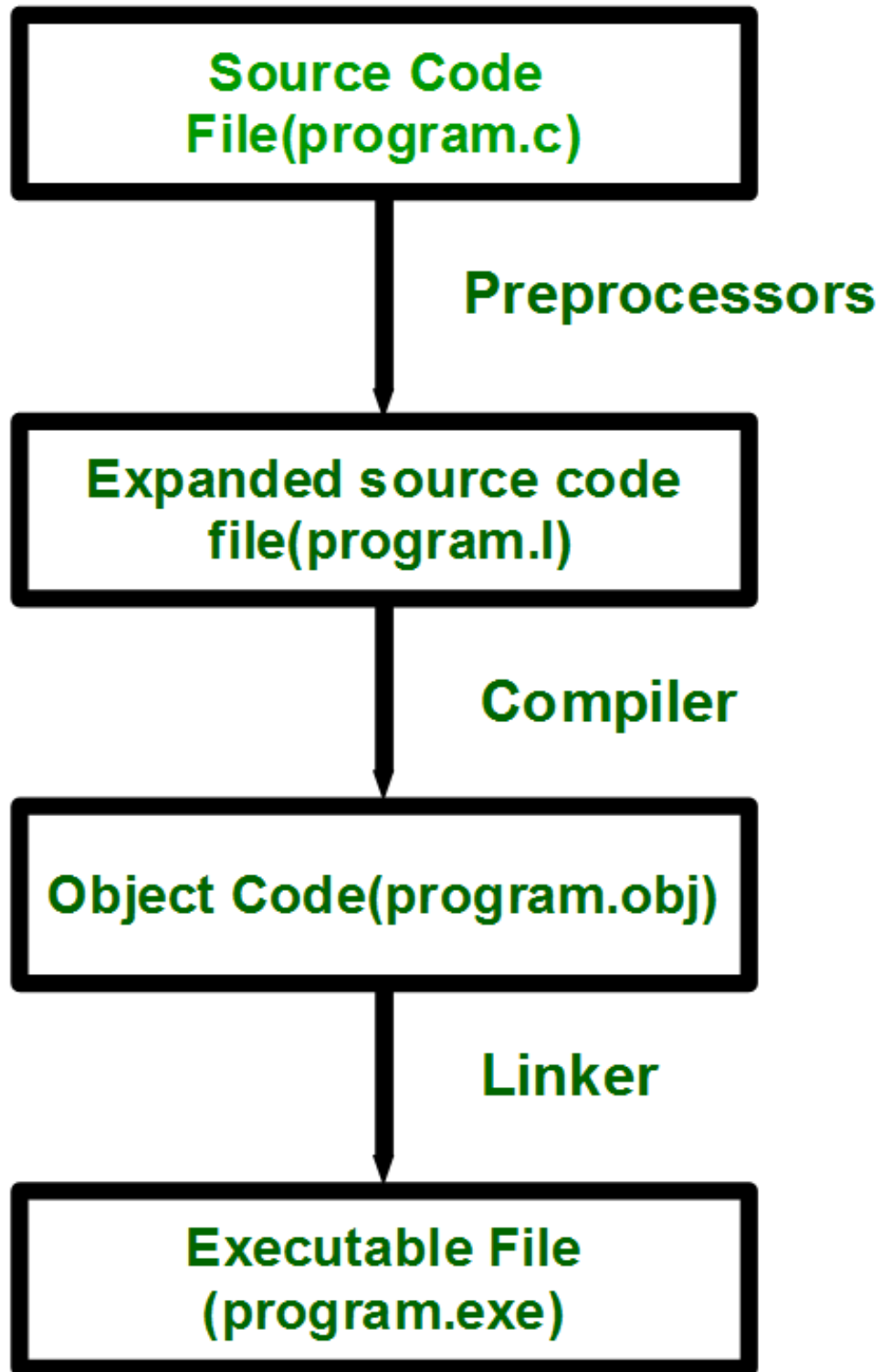
Console

Memory

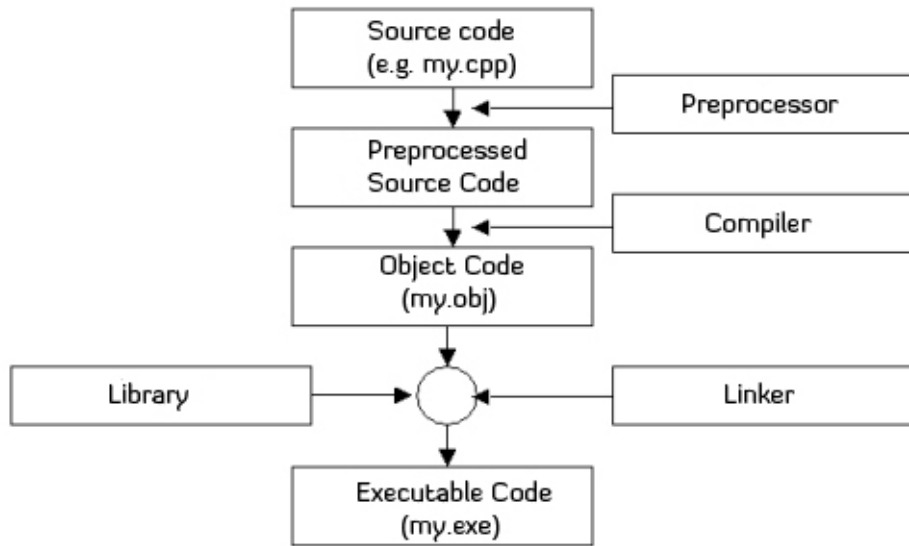
Programming Design - Introduction 31 / 62 Ling-Chieh Kung (NTU IM)

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

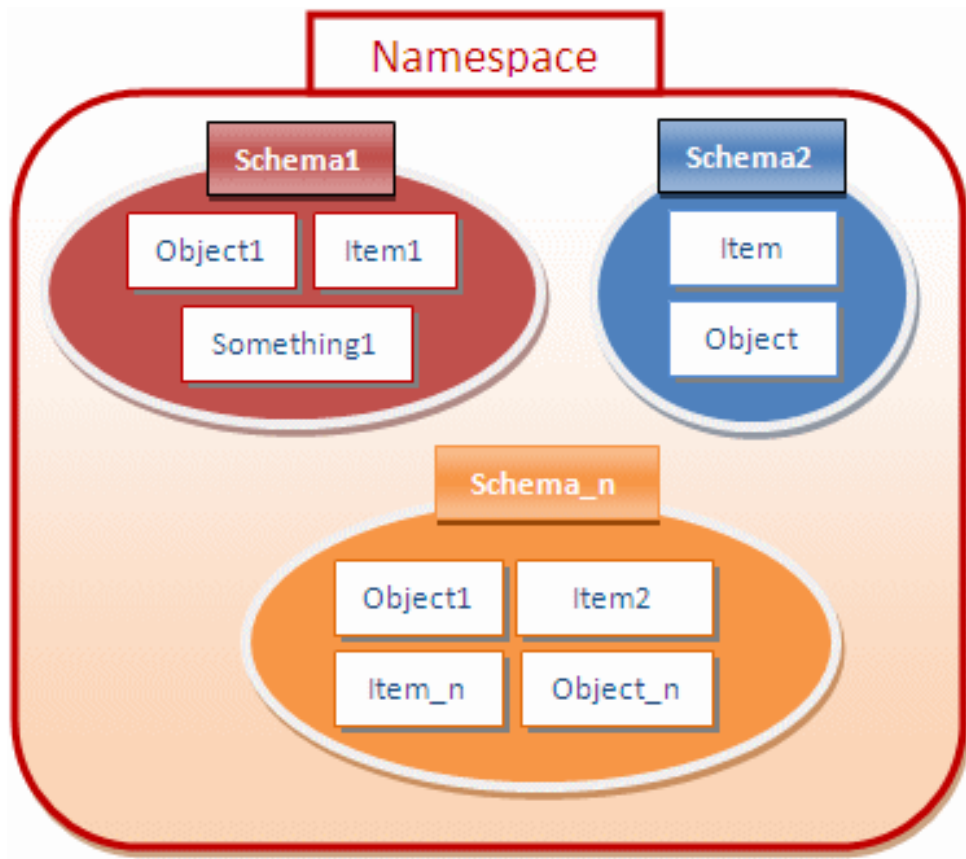
# Preprocessors

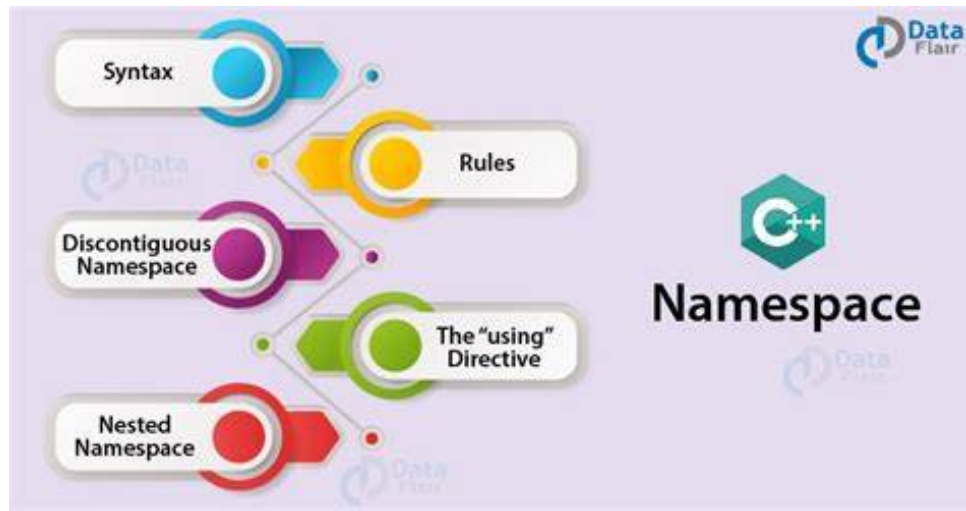






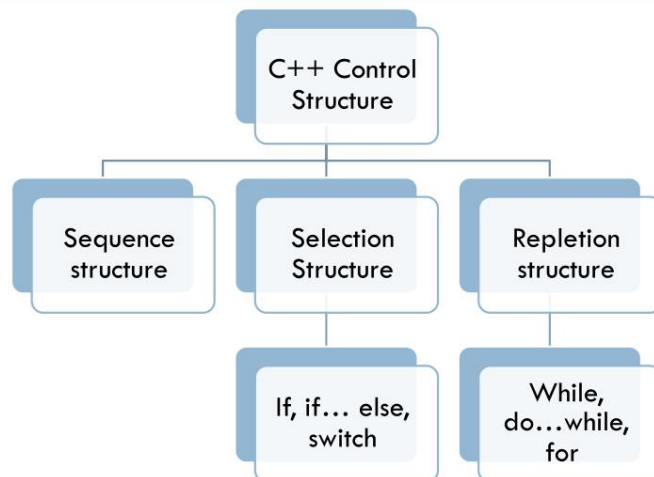
## Namespace





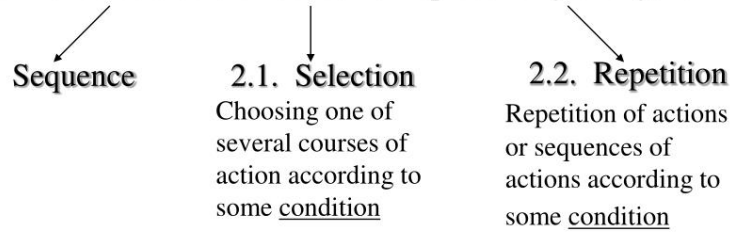
## Selection and Repetition

### Control structures in C++



### 3. Control Structures

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



**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

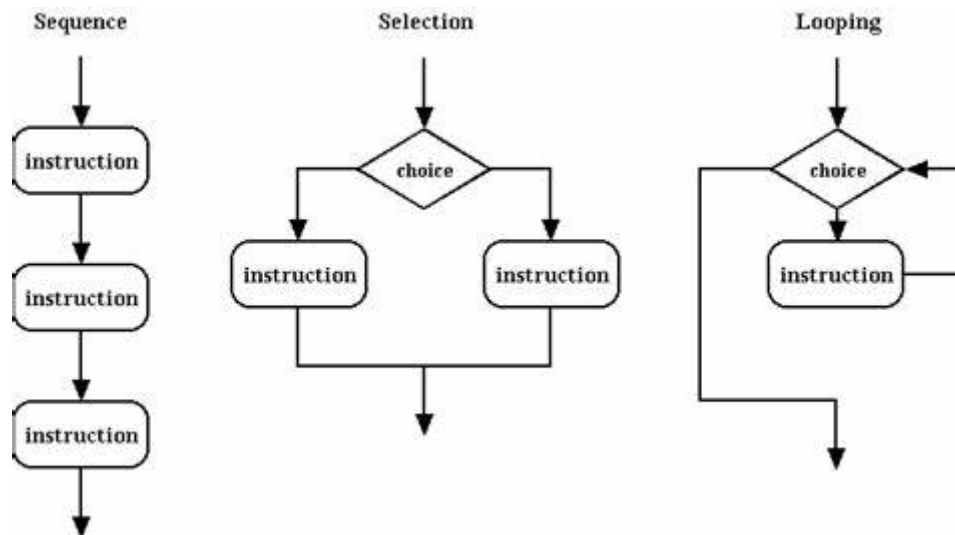
Syntax Constructs:

built in	<b>if</b> <b>if/else</b> <b>? :</b> <b>switch</b>	<b>while</b> <b>for</b> <b>do/while</b>
----------	--	---

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

09/2013

C++ Programming

L2.42

# Repetition

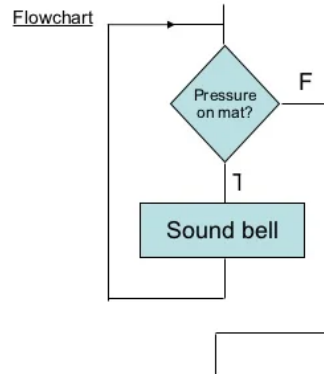
- Carries out a particular action over and over again until the condition is met.
- A loop is created to return the program to where the repetition has started for as long as it takes until the condition is met.
- There are two ways of testing to see if the end condition is met: 1) pre-test loops 2) 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

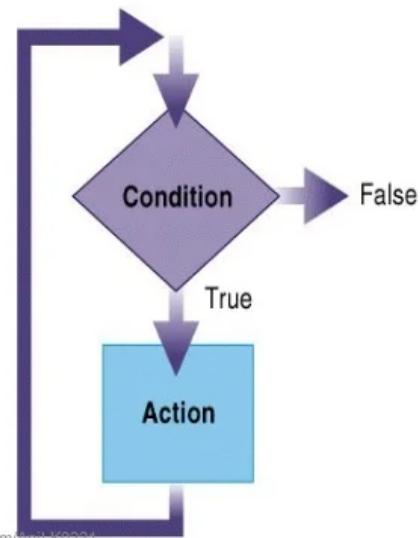
ENDWHILE



## Repetition (Loop)

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

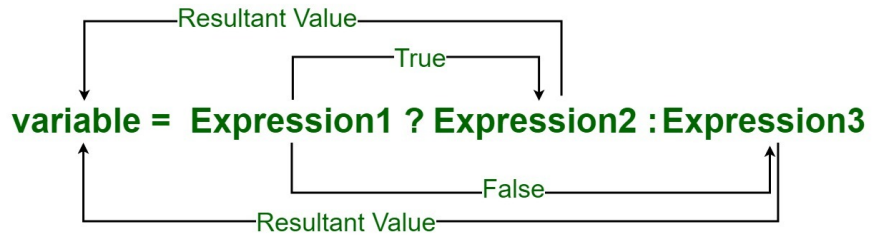
### Do-While Control Structure



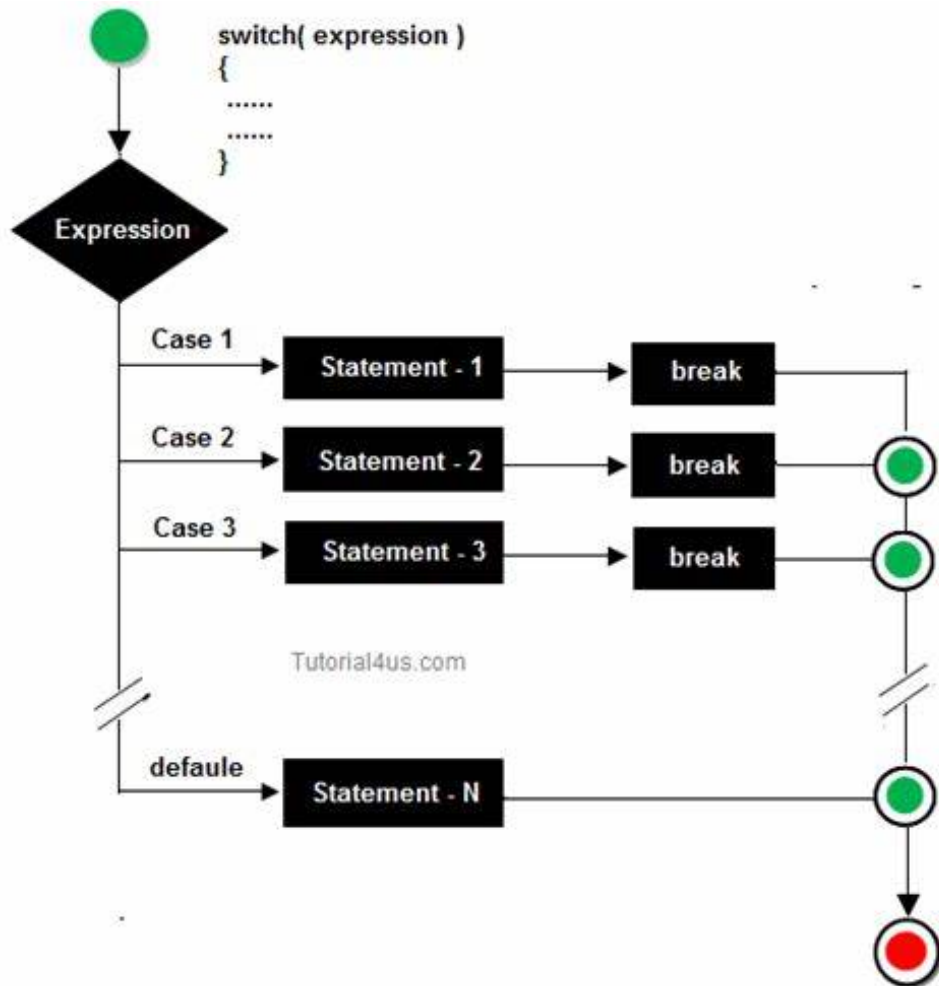
<https://www.facebook.com/AmiLR0221>

## Ternary If

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



## Switch Case



## While

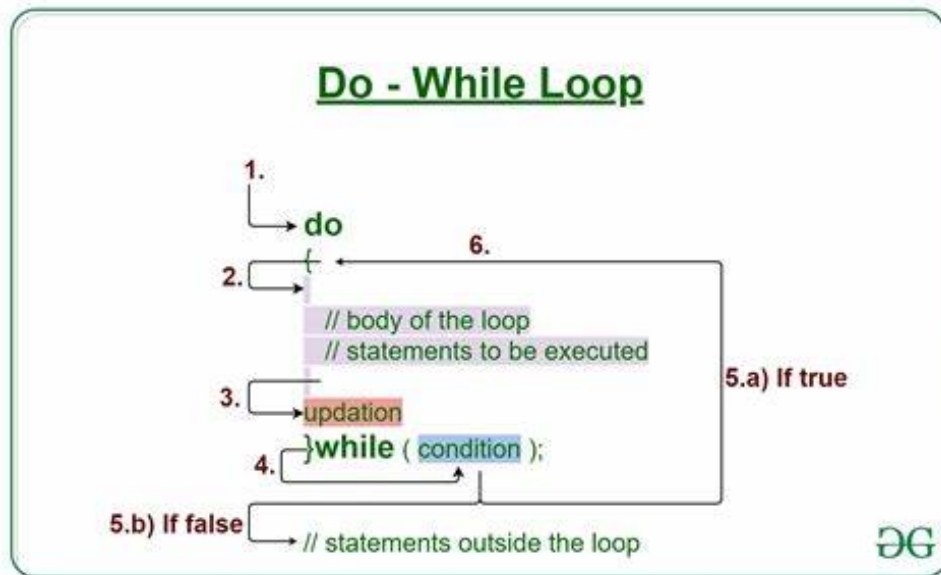
# while Statement in C++

```
#include <iostream>
int main()
{
    int count = 1;
    while (count <= 10)
    {
        std::cout << "Hello there" <<std::endl;
        count = count + 1;
    }
    return 0;
}
```

The flowchart illustrates the execution of a while loop. It starts with 'Enter while loop', leading to a decision diamond 'Test condition'. If the condition is 'TRUE', the flow goes to a box labeled 'Statements' and then loops back to 'Test condition'. If the condition is 'FALSE', the flow goes to 'Exit while loop'.



## 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

## Scope of Variables

# Scope of variables

```
#include <iostream.h>
```

```
int x;
char ch;
```

Global variables

```
main()
{
```

```
int age;
float number;
```

Local variables

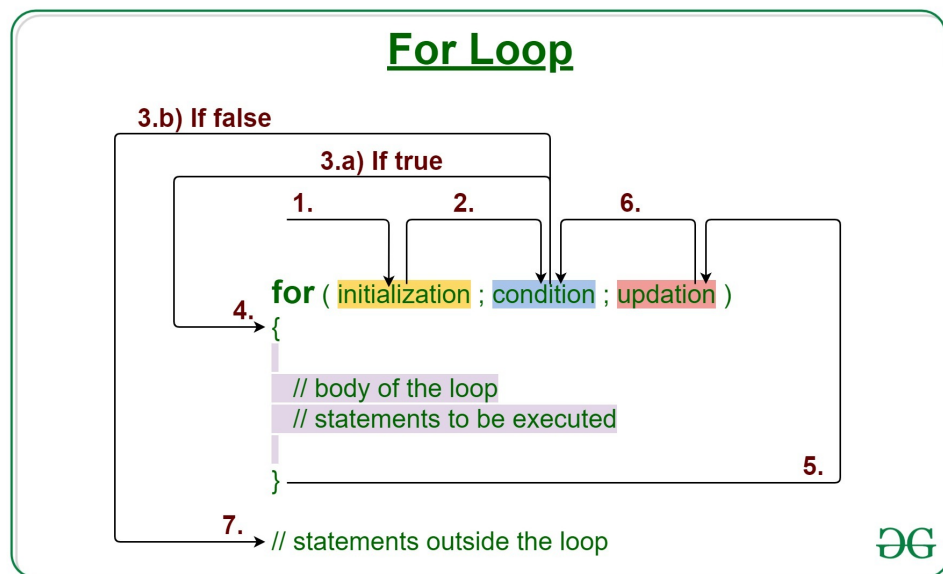
```
cout << "Enter your age: ";
cin >> age;
...
```

```
}
```

Lecturer: Hong Mom

11

## For Loop



## Nested Loop



# Nested Loops

**Outer Loop**

**Inner Loop**

```
for (i=1; i<=3; i++)
{
  for (j=1; j<=3; j++)
  {
    // ...
  }
}
```

For each run of the outer loop, the inner loop runs maximum number of times.

i	j
1	--> 1 to 3
2	--> 1 to 3
3	--> 1 to 3

Outer Loop Variable - i

Inner Loop Variable - j

Sumit Tiwari (SoftEthics Tech)

## Break and Continue

```
for(i=1; i<=5; i++)
{
  if(i==3)
  break;
  printf("%d \n", i);
}
```

1

2

```
for(i=1; i<=5; i++)
{
  if(i==3)
  continue;
  printf("%d \n", i);
}
```

1

2

4

5

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

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

```
for (int exp; test exp; update exp)
{
  statement/s
  if (condition)
  break;
  statement/s;
}
```

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 (initial 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

### Operators in C

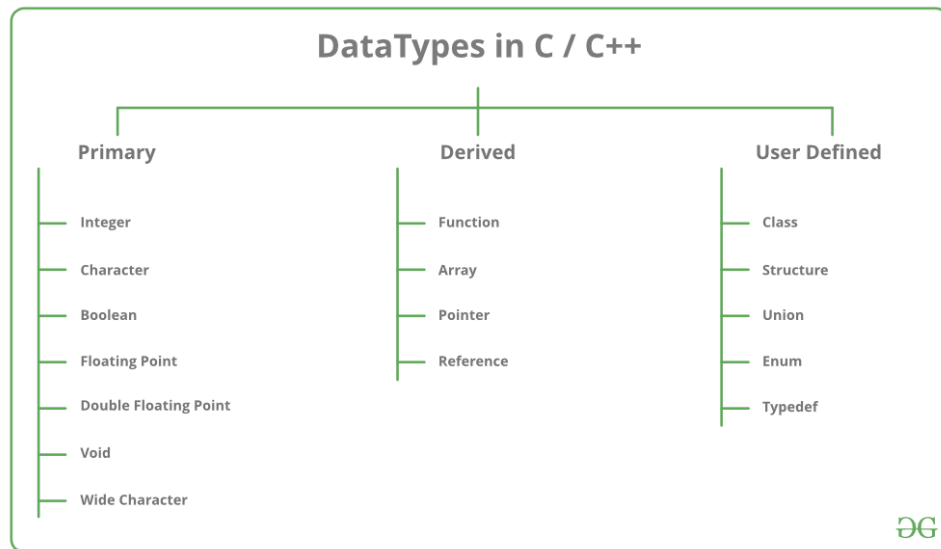
	Operator	Type
Unary operator	+, -, --	Unary operator
Binary operator	+, -, *, /, %	Arithmetic operator
	<, <=, >, >=, ==, !=	Relational operator
	&&,   , !	Logical operator
	&,  , <<, >>, ~, ^	Bitwise operator
	=, +=, -=, *=, /=, %=	Assignment operator
Ternary operator	?:	Ternary or conditional operator

## Priority of Operations

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

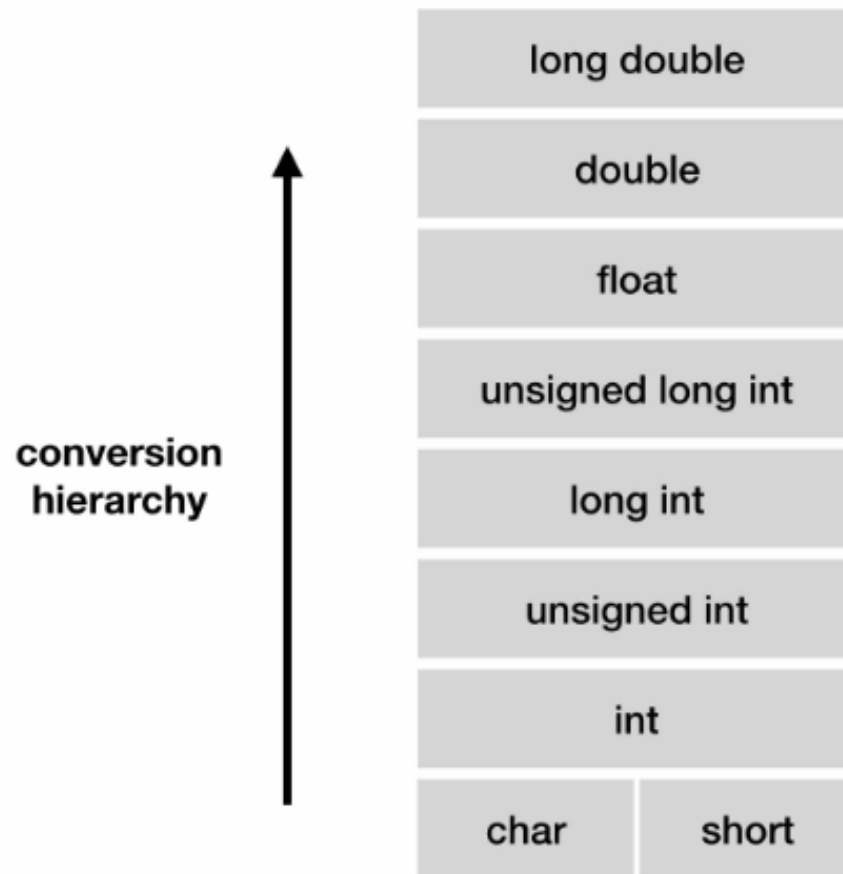
Signs of operations	Name of operation, explanation	Associativity
() [] . ->	Primary	From left to right
+ - ~ ! * & ++ -- sizeof (type) (type cast)	Unary	From right to left
* / %	Multiplicative, arithmetical, binary	From left to right
+ -	Additive, arithmetical, binary	From left to right
>> <<	Shift	From 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. )	$3.4 \cdot 10^{-38}$ to $3.4 \cdot 10^{38}$ (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. )	$1.7 \cdot 10^{-308}$ to $1.7 \cdot 10^{308}$ (both positive and negative)
long double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit. )	$1.7 \cdot 10^{-308}$ to $1.7 \cdot 10^{308}$ (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.

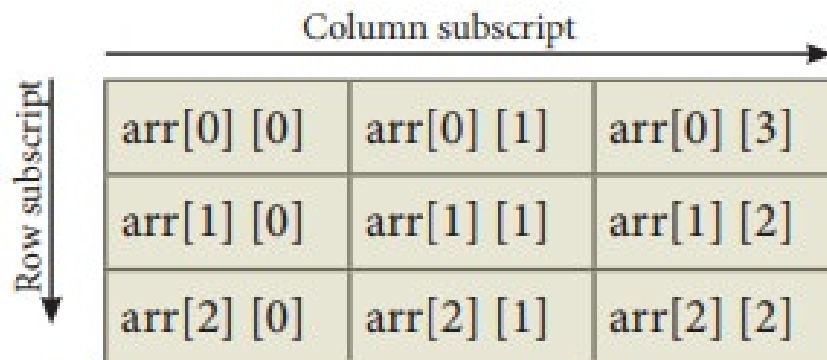
```
int main() {
    int a = 31;
    int b = 3;
    float x = a/b;
    float y = static_cast<float>(a)/b;
    cout << "Output without static_cast = " << x << endl;
    cout << "Output with static_cast = " << y << endl;
}
```

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

# 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`

## Array



The array `arr` can be conceptually viewed in matrix form with 3 rows and columns. 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

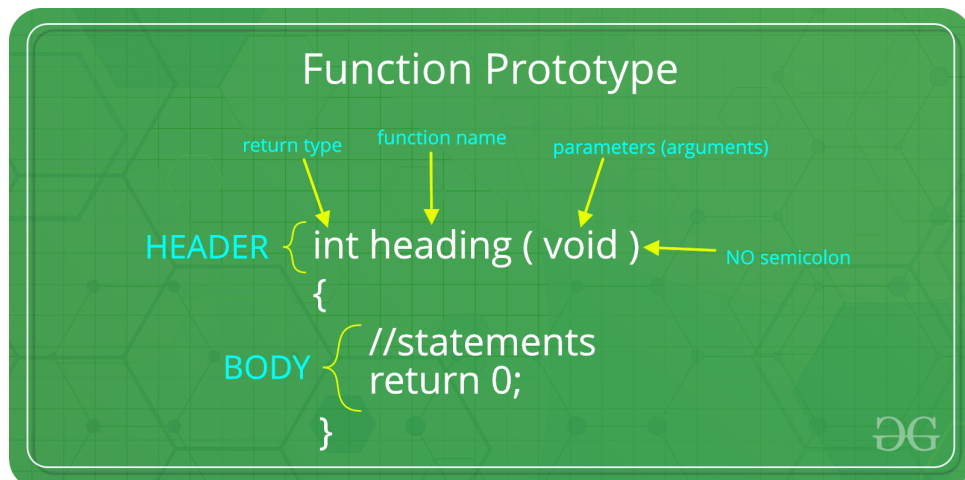
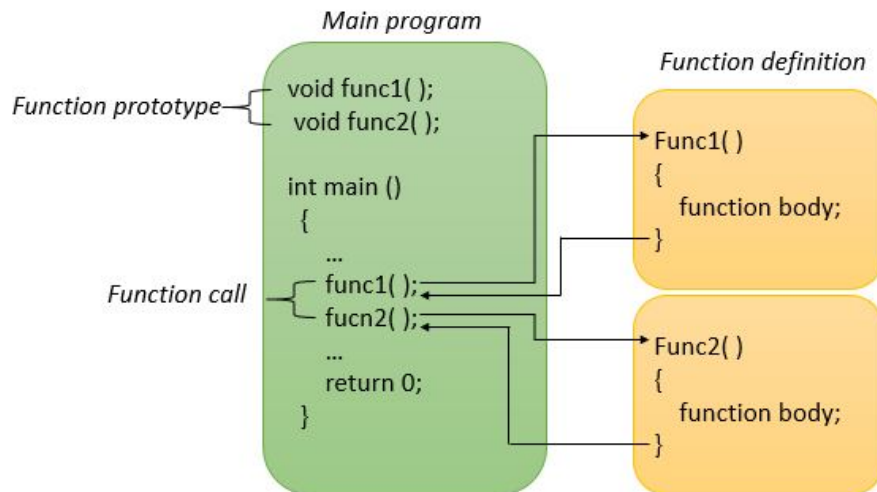
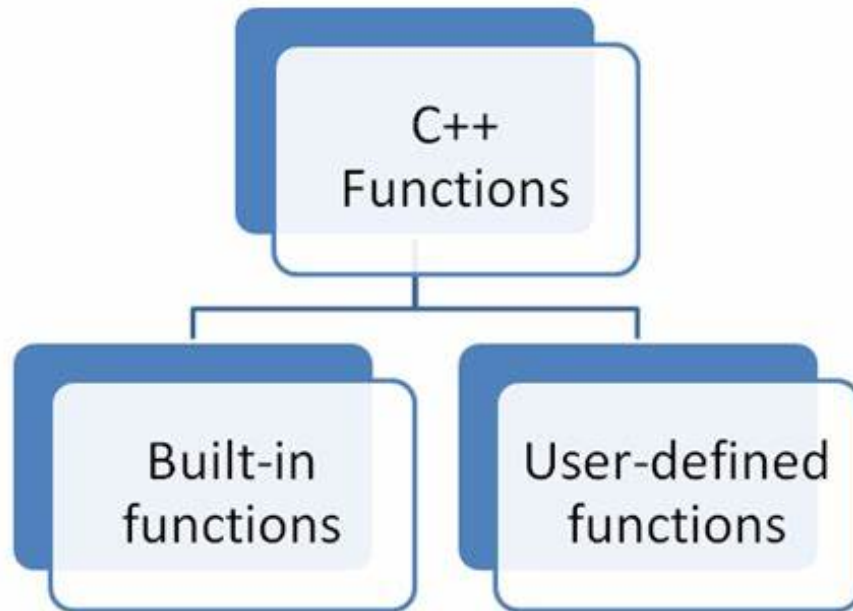
```
int sumOfArray( int values[], int numValues )
```

21

language built-in array	container library array
<pre>#include &lt;iostream&gt; using namespace std;  int main() {     int myarray[3] = {10,20,30};      for (int i=0; i&lt;3; ++i)         ++myarray[i];      for (int elem : myarray)         cout &lt;&lt; elem &lt;&lt; '\n'; }</pre>	<pre>#include &lt;iostream&gt; #include &lt;array&gt; using namespace std;  int main() {     array&lt;int,3&gt; myarray {10,20,30};      for (int i=0; i&lt;myarray.size(); ++i)         ++myarray[i];      for (int elem : myarray)         cout &lt;&lt; elem &lt;&lt; '\n'; }</pre>

## Function





## Parameters

# Function Definition

```
def add(a, b):
    return a + b
```

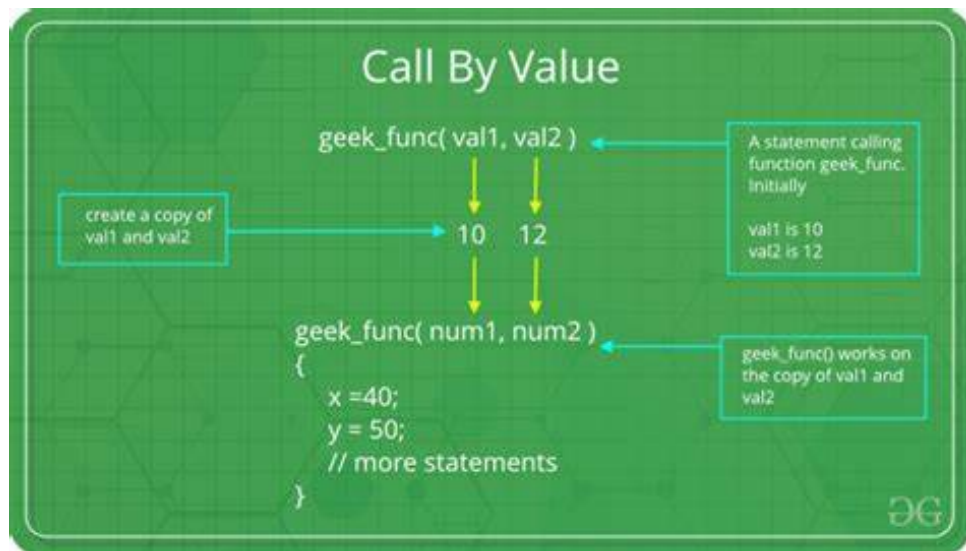
# Function Call

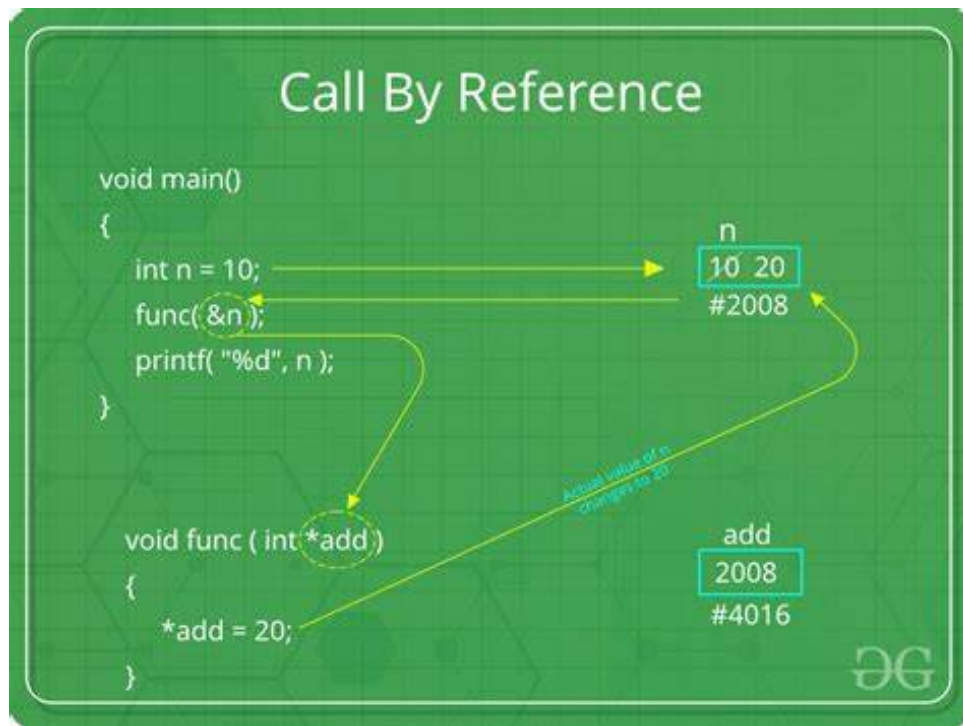
```
add(2, 3)
```

## Arguments

getKT.com

## Call by Value and Call by Reference





Programming in C - Call By

<u>CALL BY VALUE</u>	<u>CALL BY REFERENCE</u>	<u>CALL BY ADDRESS</u>
<p>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.</p> <pre> 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); }                 </pre> <p>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.</p>	<p>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.</p> <pre> 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]); }                 </pre> <p>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.</p>	<p>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.</p> <pre> 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(&amp;a,&amp;b); /*calling swap function*/     printf("After swap: a=%d,b=%d",a,b); }                 </pre> <p>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</p>

# 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

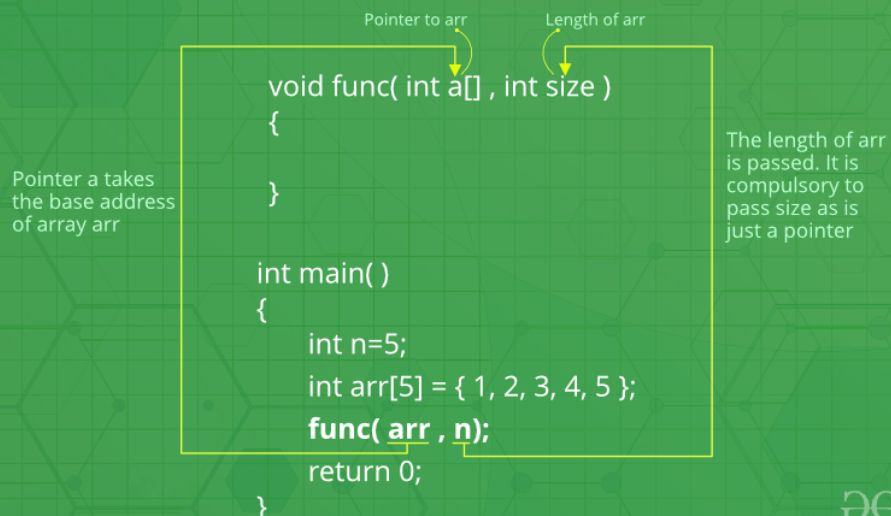
Programmer passes the references of the actual arguments to the formal parameters

Memory is allocated only for actual arguments and formal parameters share that memory

Visit [www.PEDIAA.com](http://www.PEDIAA.com)

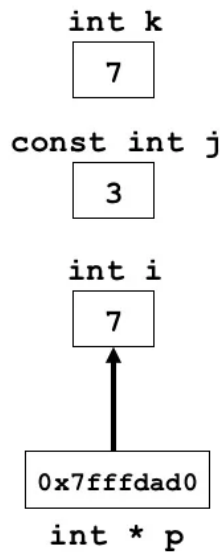
## Passing Array to Function

### Passing array to function in C



## Pointer

## What's a Pointer in C++?

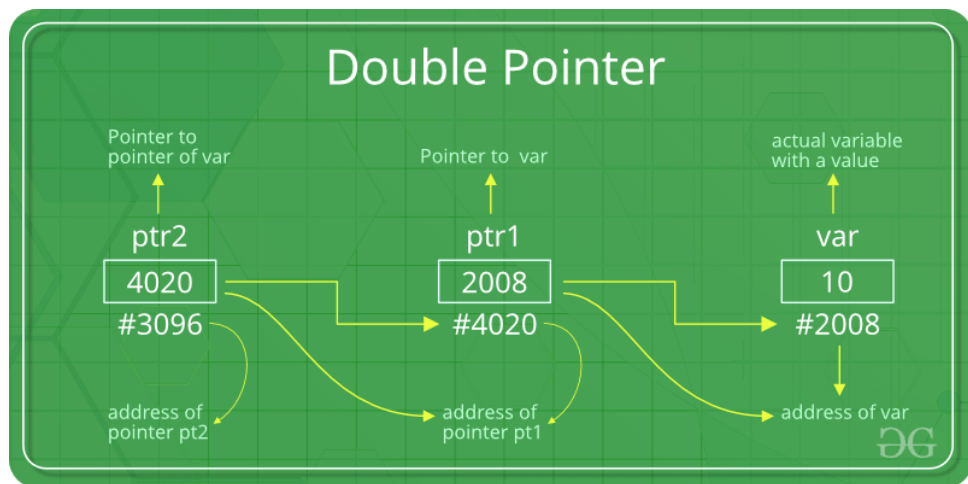
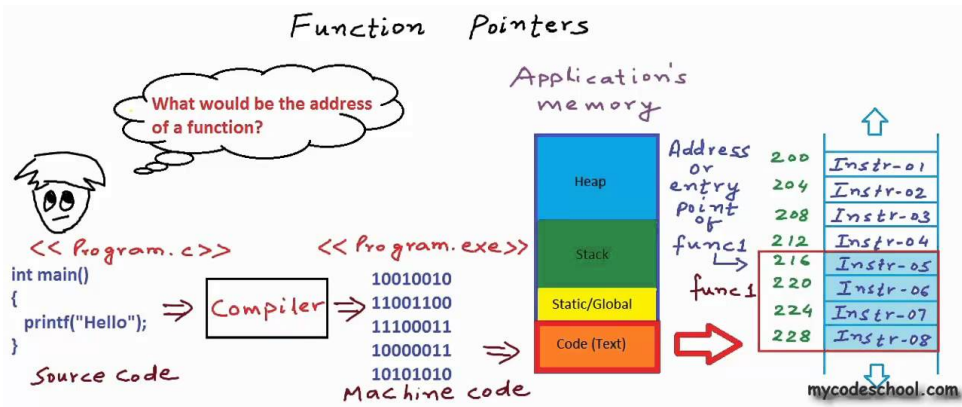


- A variable holding an address
  - Of what it “points to” in memory
- Can be untyped
 

```
void * v; // can point to any type
```
- However, usually they're typed
  - Checked by compiler
  - Can only be assigned addresses of variables of the type to which it can point

```
int * p; // can only point to int
```
- Addresses: garbage, something, nothing
  - When created: `int * p = i;` VS. `int * q;`
  - `q = 0;` // now it points to nothing
  - `p = NULL;` // not portable, use 0 instead

CSE 232: C++ pointers, arrays, and references



## 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 (&amp;) and Dereference Operator (\*)

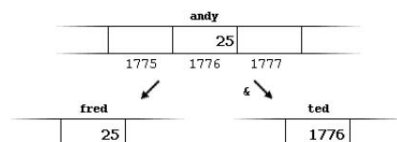
- Reference operator (&) as already discussed, gives the address of a variable.
- To get the value stored in the memory address, we use the dereference operator (\*).
- **For Example:** If a number variable is stored in the memory address 0x123, and it contains a value 5.
- The reference (&) operator 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

## Address Operator (&amp;)

- The '&' operator returns the 'address of' its operand.
- '&' can be translated 'address of'

```
andy = 25;
fred = andy;
ted = &andy;
```

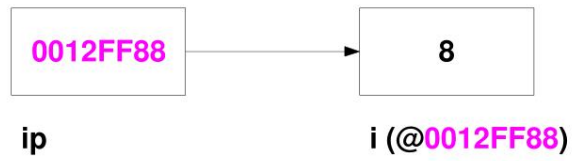




---

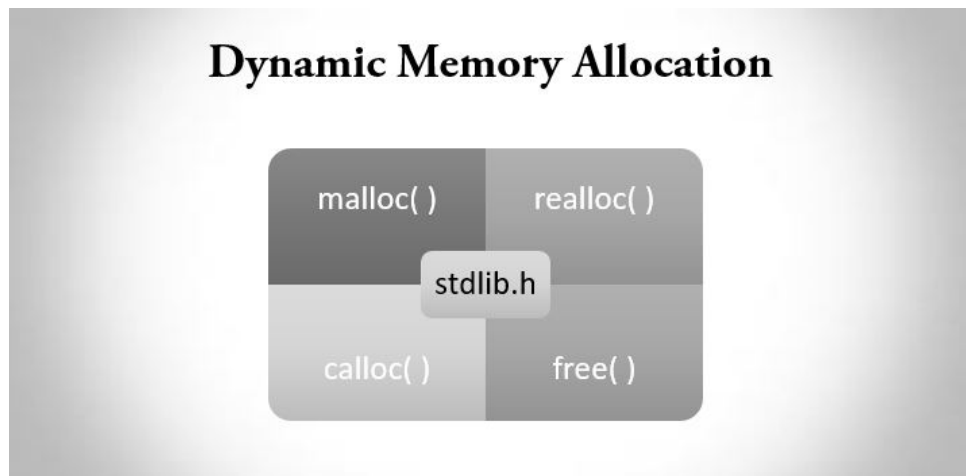
## Address of operator

---



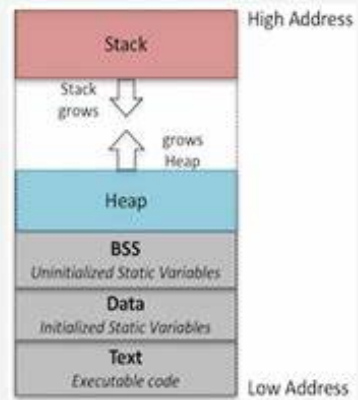
```
int i = 8;  
int *ip;  
  
ip = &i;
```

## DMA(Dynamic Memory Allocation)





**calloc ()**  
**malloc()**  
**realloc()**  
**free()**  
**new & delete**

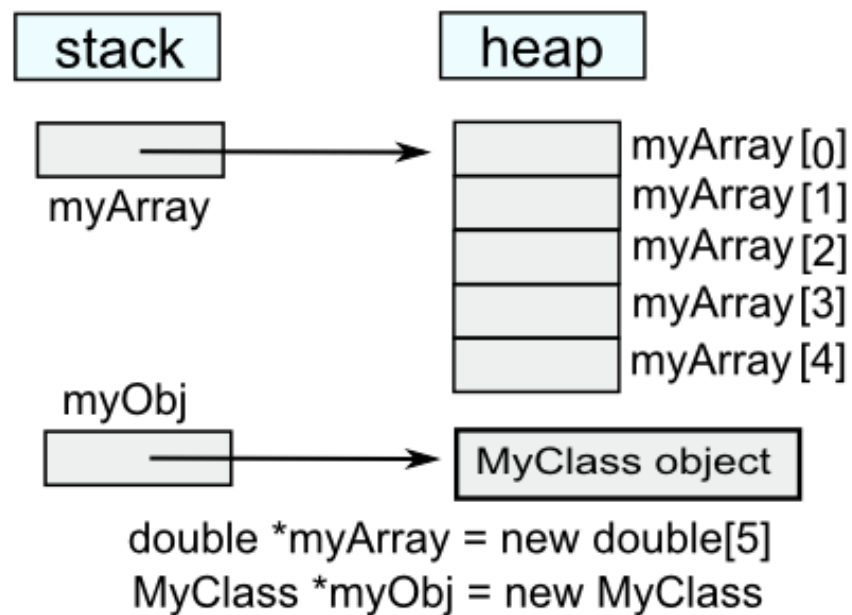


**C/C++**

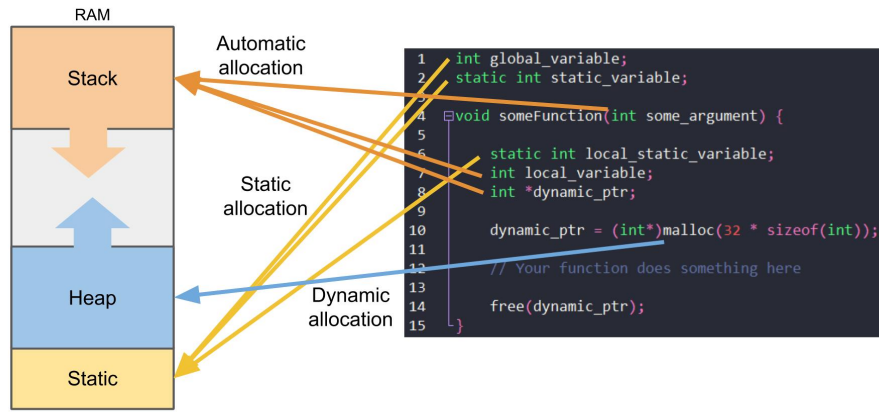


## DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

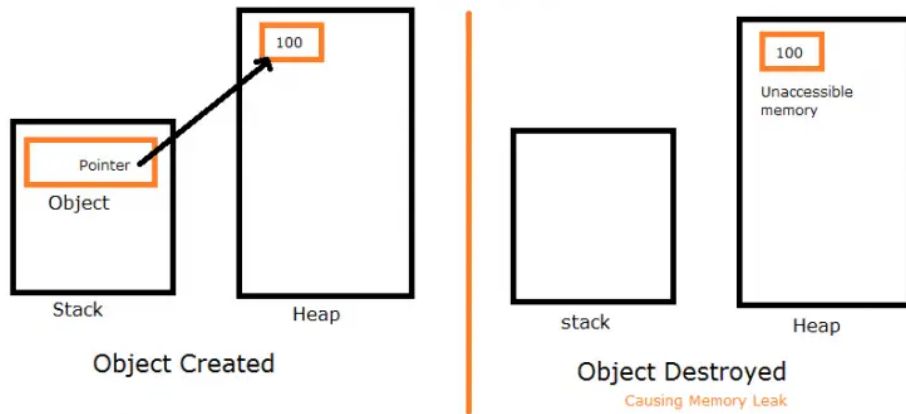
malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
<code>int *ptr;ptr = malloc( 20 * sizeof(int) );</code> For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	<code>int *ptr;Ptr = calloc( 20, 20 * sizeof(int) );</code> 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 <code>int *ptr;ptr = (int*)malloc(sizeof(int)*20 );</code>	Same as malloc () function <code>int *ptr;ptr = (int*)calloc( 20, 20 * sizeof(int) );</code>



## Memory Allocation



## Memory Leak



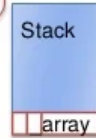
## Heap vs Stack

`int _array[10];` → stored in stack

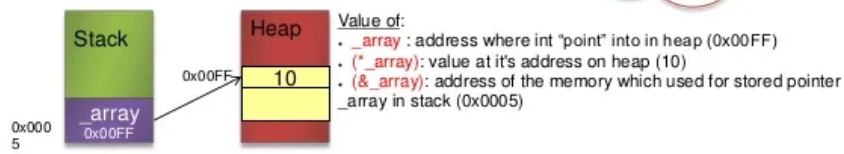
`int *_array = new int[n]`

- Pointer `_array` is stored in Stack
- Data of array is stored in Heap

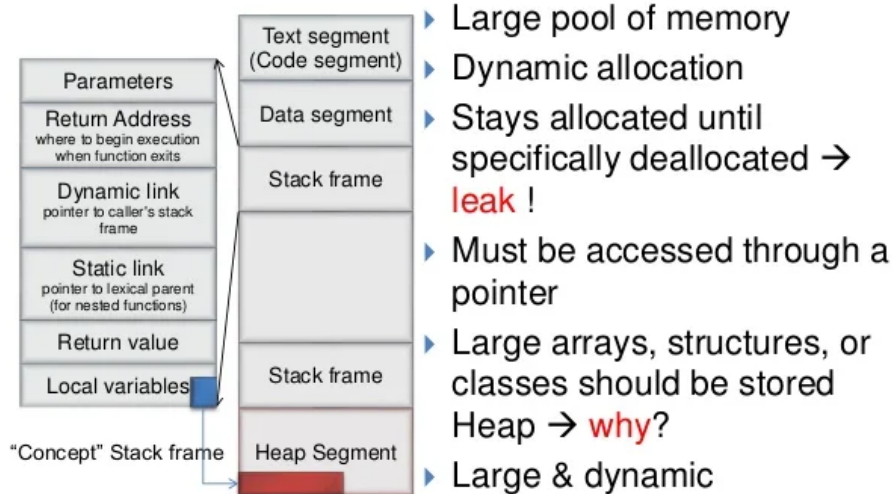
Always stored in stack?



Any chance to store `_array` in heap?

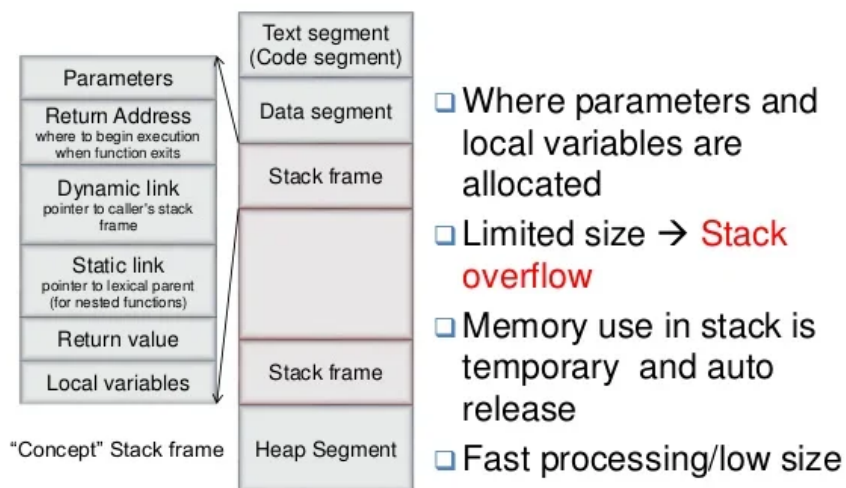


# Heap



▶ 7

# Stack



▶ 6

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  
Dynamic memory allocation (DMA)

Using pointers in functions  
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)

## Multi-Dimension Array

Pointers and multi-dimensional arrays

```
int C[3][2][2]
int (*P)[2][2] = C; ✓
Print C // 800
Print *C or C[0] or &C[0][0] // 800
    ↓
    int (*)[2]
```

mycodeschool.com

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

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

Arrays In Java - 2D Arrays (Multidimensional Arrays) by Deepak (Part 1) || Arrays for ...

## Multi-Dimensional (2-D)

**What is Multi-Dimensional Array :**

- An array having multiple rows or columns is known as multi-dimensional array.
- These are also known as array of arrays because array is present in another array.
- There are two types of multi-dimensional array :
  1. 2-D Array
  2. 3-D Array
- We can represent 2-D multi-dimensional array as follows :

	Columns			
	0,0	0,1	0,2	- Index Positions
Rows	10	20	30	
	1,0	1,1	1,2	
	40	50	60	- Elements in Array

OR

Array pointing another array

0	10	20	30	- Index Positions
1	40	50	60	- Elements in Array

21:44 / 22:08      Scroll for details

Basics of pointers      Using pointers in functions

Dynamic memory allocation (DMA)      Arrays and pointer arithmetic

## Example: lower triangular arrays

- Let's visualize the memory events.
- In general, the space of the three 1-dim dynamic arrays may be **separated**.
- However, the space of the array elements in each array are **contiguous**.

```

int main()
{
    int r = 3;
    int** array = new int*[r];
    for(int i = 0; i < r; i++)
    {
        array[i] = new int[i + 1];
        for(int j = 0; j <= i; j++)
            array[i][j] = j + 1;
    }
    print(array, r); // later
    // some delete statements
    return 0;
}
    
```

Address	Identifier	Value
0x20c544	r	3
0x20c548	array	0x20c554
0x20c550		
0x20c554	N/A	0x20c66c
0x20c55c	N/A	0x20c670
0x20c664	N/A	0x20c678
0x20c66c	N/A	1
0x20c670	N/A	1
0x20c674	N/A	2
0x20c678	N/A	1
0x20c67c	N/A	2
0x20c680	N/A	3

Memory

Programming Design - Pointers      70 / 87      Ling-Chieh Kung (NTU IM)

Basics of pointers      Using pointers in functions

Dynamic memory allocation (DMA)      Arrays and pointer arithmetic

## Example: lower triangular arrays

- To pass a two-dimensional dynamic array, just pass that pointer.

```

int main()
{
    int r = 3;
    int** array = new int*[r];
    for(int i = 0; i < r; i++)
    {
        array[i] = new int[i + 1];
        for(int j = 0; j <= i; j++)
            array[i][j] = j + 1;
    }
    print(array, r);
    // some delete statements
    return 0;
}
    
```

```

void print(int** arr, int r)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j <= i; j++)
            cout << arr[i][j] << " ";
        cout << "\n";
    }
}
    
```

Programming Design - Pointers      71 / 87      Ling-Chieh Kung (NTU IM)



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

## Example: lower triangular arrays

- An alternative:

```
int main()
{
    int r = 3;
    int** array = new int*[r];
    for(int i = 0; i < r; i++)
    {
        array[i] = new int[i + 1];
        for(int j = 0; j <= i; j++)
            array[i][j] = j + 1;
    }
    print(array, r);
    // some delete statements
    return 0;
}
```

```
void printID(int* arr, int n)
{
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << "\n";
}

void print(int** arr, int r)
{
    for(int i = 0; i < r; i++)
        printID(arr[i], i + 1);
}
```

Programming Design – Pointers 72 / 87 Ling-Chieh Kung (NTU IM)

## Pointer Arithmetic

### Introduction to C++

#### Pointer Arithmetic

- The C++ language allows you to perform integer addition or subtraction operations on pointers.
- If `ptr` points to an integer, `ptr + 1` is the address of the next integer in memory after `ptr`.
- `ptr - 1` is the address of the previous integer before `ptr`.
- Note** that `ptr + 1` does not return the *memory address* after `ptr`, but the memory address of the *next object of the type* that `ptr` points to. If `ptr` points to an integer (assuming 4 bytes), `ptr + 3` means 3 integers after `ptr`, which is 12 memory addresses after `ptr`. If `ptr` points to a char, which is always 1 byte, `ptr + 3` means 3 chars after `ptr`, which is 3 memory addresses after `ptr`.
- When calculating the result of a pointer arithmetic expression, the compiler always multiplies the integer operand by the size of the object being pointed to. This is called **scaling**.

### Lecture Slides By Adil Aslam

## 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

CS-2303, C-Term 2010

Arrays in C &amp; C++

35



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

### Indexing and pointer arithmetic

- The array indexing operator `[]` is just an **interface** for doing pointer arithmetic.
  - Interface: a (typically safer and easier) way of completing a task.

```
int x[3] = {1, 2, 3};
for(int i = 0; i < 3; i++)
  cout << x[i] << " "; // x[i] == *(x + i)
for(int i = 0; i < 3; i++)
  cout << *(x + i) << " "; // 1 2 3
```

- `x[i]` and `*(x + i)` are identical, but using the former is safer and easier.

- The address stored in an array variable (e.g., `x`) **cannot be modified**.

```
int x[3] = {1, 2, 3};
for(int i = 0; i < 3; i++)
  cout << *(x++) << " "; // error!
```

Programming Design – Pointers 79 / 87 Ling-Chieh Kung (NTU IM)

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

### Example 3: returning a pointer

- Recall that we want to find the first negative number in an array.
  - We want its **value** and **index**.
  - We return its address.
- Three issues remain.
  - Why not return its index?
  - What if all elements in `a` are nonnegative?
  - Why not `const int a[]`?

```
#include <iostream>
using namespace std;
int* firstNeg(int a[], const int n) {
  for(int i = 0; i < n; i++) {
    if(a[i] < 0)
      return &a[i];
  } // what if a[i] >= 0 for all i?
}
int main()
{
  int a[5] = {0};
  for(int i = 0; i < 5; i++)
    cin >> a[i];
  int* p = firstNeg(a, 5);
  cout << *p << " " << p - a << "\n";
  return 0; // what is p - a?
}
```

Programming Design – Pointers 83 / 87 Ling-Chieh Kung (NTU IM)

Basics of pointers  
Dynamic memory allocation (DMA)

Using pointers in functions  
Arrays and pointer arithmetic

## Example 3: returning a pointer

- To take the possibility of having no negative number into consideration:

```
#include <iostream>
using namespace std;
int* firstNeg(int a[], const int n) {
    for(int i = 0; i < n; i++) {
        if(a[i] < 0)
            return &a[i];
    }
    return nullptr;
}

int main()
{
    int a[5] = {0};
    for(int i = 0; i < 5; i++)
        cin >> a[i];
    int* p = firstNeg(a, 5);
    if(p != nullptr)
        cout << *p << " " << p - a << "\n";
    return 0;
}
```

Programming Design – Pointers 84 / 87 Ling-Chieh Kung (NTU IM)

Basics of pointers  
Dynamic memory allocation (DMA)

Using pointers in functions  
Arrays and pointer arithmetic

## Example 3: returning a pointer

- Why not `const int a[]`?
  - We return the address of `a[i]`, which allows the caller to alter the element.
  - `const int*` and `int*` are different!

```
#include <iostream>
using namespace std;
int* firstNeg(int a[], const int n) {
    for(int i = 0; i < n; i++) {
        if(a[i] < 0)
            return &a[i];
    }
    return nullptr;
}

int main()
{
    int a[5] = {0};
    for(int i = 0; i < 5; i++)
        cin >> a[i];
    int* p = firstNeg(a, 5);
    if(p != nullptr)
        *p = -1 * *p; // *p at the LHS of =
    return 0;
}
```

- One cannot modify the variable pointed by a **constant pointer**.

Programming Design – Pointers 85 / 87 Ling-Chieh Kung (NTU IM)

Basics of pointers  
Dynamic memory allocation (DMA)

Using pointers in functions  
Arrays and pointer arithmetic

## Example 3: returning a pointer

- To use `const int a[]`, we need to change the return type.
  - We should also **return const int\***.
  - This is an `int*` that cannot be put at the LHS of an assignment operator.

```
#include <iostream>
using namespace std;
const int* firstNeg
(const int a[], const int n) {
    for(int i = 0; i < n; i++) {
        if(a[i] < 0)
            return &a[i];
    }
    return nullptr;
}

int main()
{
    int a[5] = {0};
    for(int i = 0; i < 5; i++)
        cin >> a[i];
    const int* p = firstNeg(a, 5);
    if(p != nullptr)
        cout << *p << "\n"; // OK
    return 0;
}
```

Programming Design – Pointers 86 / 87 Ling-Chieh Kung (NTU IM)

## Main Function Arguments

## Command line arguments

argument count

array of string arguments

```
#include <stdio.h>

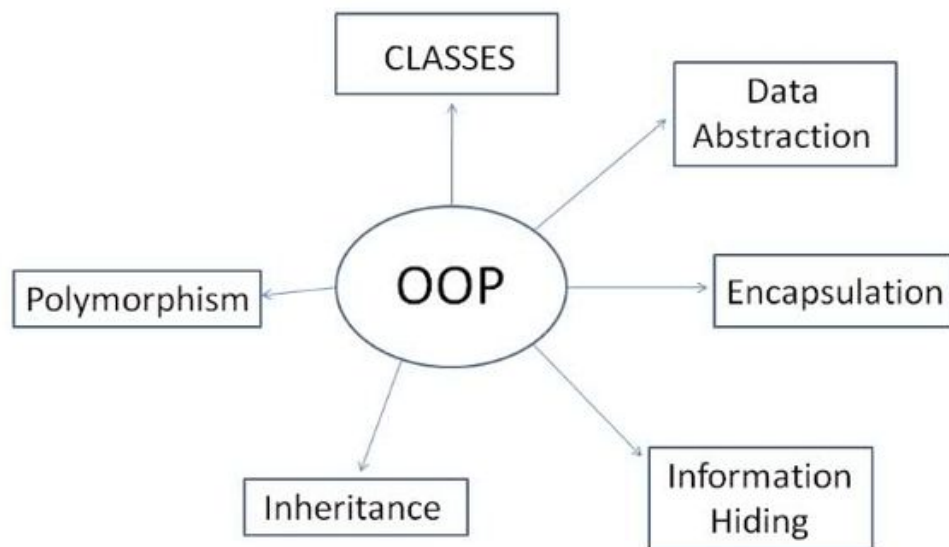
int main(int argc, char *argv[]) {
}
```

argv[0] is the program name with full path

argv[1] is the first argument

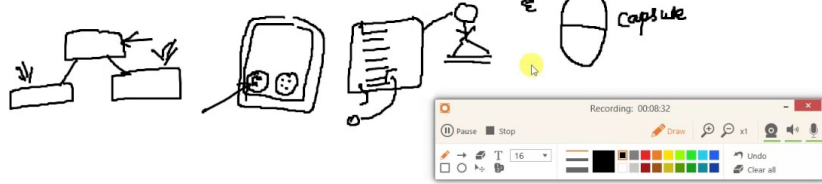
argv[2] is the second argument, etc.

## 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.

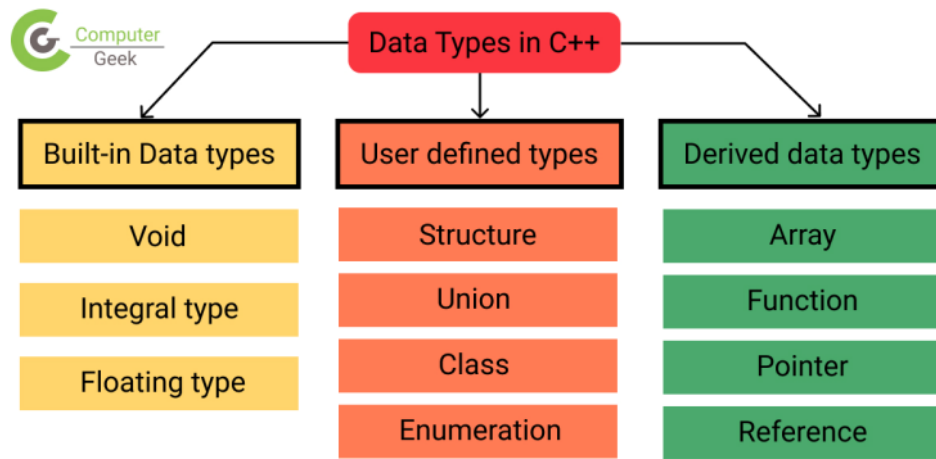


## Data Abstraction

### What is Data Abstraction?

- A data abstraction is a simplified view of an object that
  - includes only features one is interested in.
  - while hides away the unnecessary details.
- In programming languages, a data abstraction becomes an abstract data type (ADT) or a user-defined type.
- In OOP, it is implemented as a class

## Struct in C and Class in C<sup>++</sup>



## Structure Types (Using typedef)

```

typedef struct car { ← Old Type
    char engine[50];
    char fuel_type[10];
    int fuel_tank_cap;
    int seating_cap;
    float city_mileage;
}car; ← New Type

int main() {
    struct car c1;
}
  
```

car becomes a new data type.

153

## C Programming

## Structure

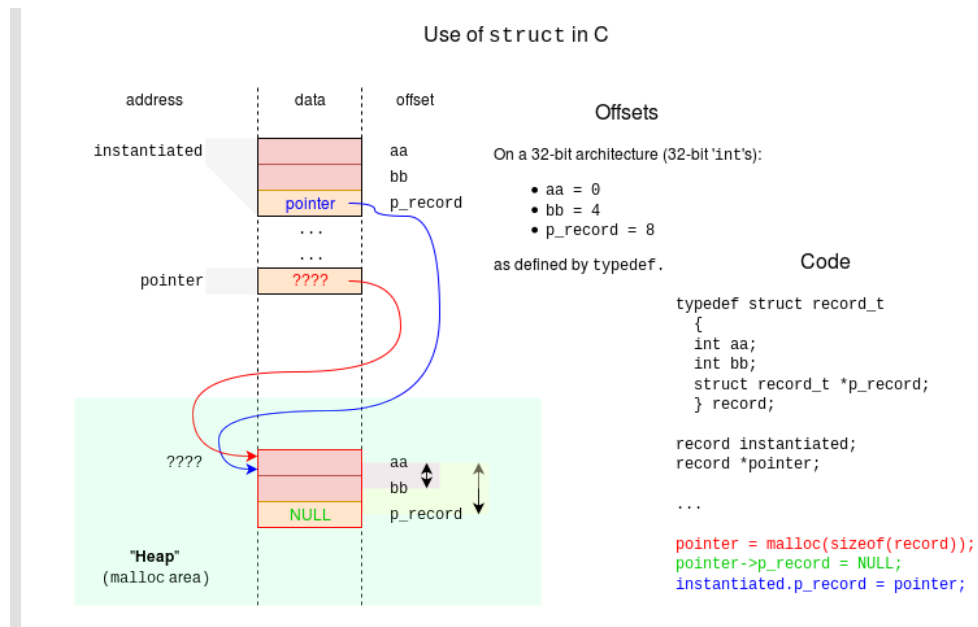
### Structures within Structures

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 [ ]: #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++

```

class myClass {
public:
    int myNum;
private:
    string myStr;
protected:
    float myFloat;

public:
    myClass() {
        ...
    }
    myClass(int x, string y){
        ...
    }

private:
    void myMethodOne() {
        ...
    }

protected:
    void myMethodTwo() {
        ...
    }
};
    
```

**Data and Access Modifiers**  
Access Modifiers are used to assign the accessibility to the class members.

**Constructors**  
A constructor is a member function which initialises objects of a class.

**Methods**  
Methods are functions that perform operations on the class data.

[enjoyalgorithms.com](http://enjoyalgorithms.com)

## Types of variables in C++

```

class GFG {
public:
    static int a;
    int b;
public:
    func()
    {
        int c;
    };
};
    
```

**Static Variable** (points to `static int a;`)

**Instance Variable** (points to `int b;`)

**Local Variable** (points to `int c;`)



```

#include<iostream>
using namespace std;
class student
{
private :
    int id;
    char name[20];
public :
    int a;
    Void getdata(void);
    Void display (void)
    {
        cout << id << '\t' << name << endl;
    }
};
    
```

**Data Members** (points to `private` section)

**Member Functions** (points to `public` section)

```

keyword
class classname ← classname
{
    Access specifiers: ← //private/public/protected
    Data members/variables; //variables
    Member functions () {} //methods
}; ← //end of class with a semicolon
    
```

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.

## 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.

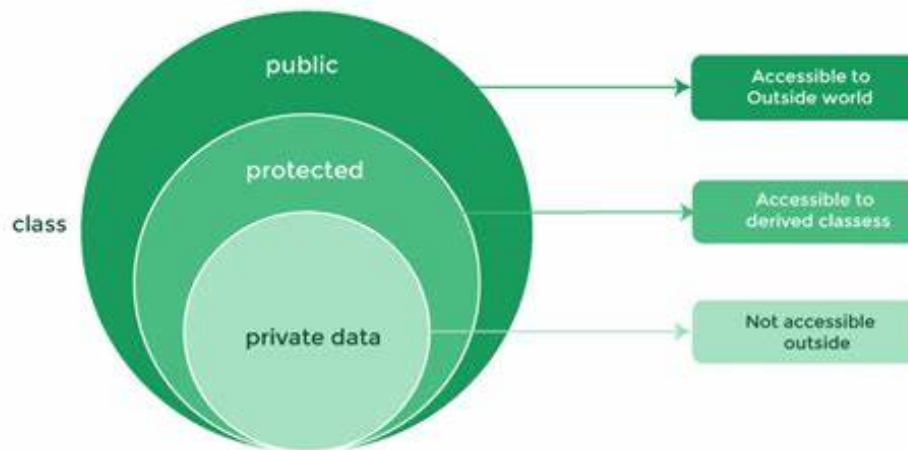
## Encapsulation

### 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

Basic concepts	Constructors and the destructor
Friends and static members	Object pointers and the copy constructor

### Visibility

- We can/must set visibility of members in a class:
  - **Public** members can be accessed **anywhere**.
  - **Private** members can be accessed only **in the class**.
  - **Protected** members will be discussed later in **this** semester.
- These three keywords are the **visibility modifiers**.
- By **default**, all members' visibility level is **private**.
  - That is why **init(5)** generates a compilation error; **init()** is private and cannot be invoked outside the class (e.g., in the main function).
- By **setting** visibility, we can **hide/open** our instance members.
  - Usually all instance variables are **private**.
  - Let's see how to do this.

Programming Design – Classes 9 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts	Constructors and the destructor
Friends and static members	Object pointers and the copy constructor

## Encapsulation

- The concepts of packaging (grouping member variables and member functions) and data hiding together form the concept of “encapsulation”.
  - Roughly speaking, we pack data (member variables) into a black box and provide only controlled interfaces (member functions) for others to access these data.
  - Others should not even know how those interfaces are implemented.
- For OOP, there are three main characteristics/functionalities:
  - Encapsulation.
  - Inheritance.
  - Polymorphism.
- The last two will be discussed later in this semester.

*later*

Programming Design – Classes 13 / 52 Ling-Chieh Kung (NTU IM)

## Class Constructor and Destructor



### 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 tilde (~)



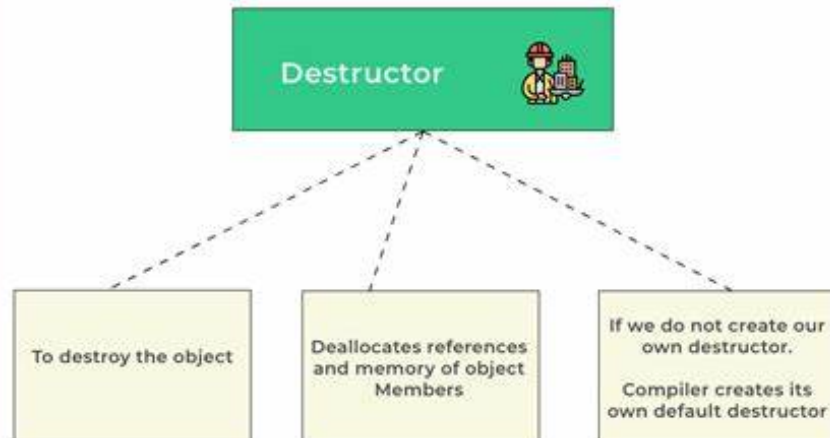
# Class Constructors

- 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 automatically invoked** when a class object is declared--if there are parameters, their values are listed in parentheses in the declaration

Object-oriented programming - IITU

3

## Destructor in C++



Constructors are a unique class functions that do the job of destroying the instances of class.

Basic concepts  
Friends and static members

Constructors and the destructor  
Object pointers and the copy constructor

## Constructors

- A constructor's name is the same as the class.
- It does not return anything, not even **void**.
- You can (and usually will) overload them.
- The constructor with no parameter is the default constructor.
- If, and only if, a programmer does not define any constructor, the compiler makes a default one which does nothing.
- A constructor may be private.
  - Be invoked only by other constructors.

```
class MyVector
{
private:
    int n;
    int* m;
public:
    MyVector() // constructors
    MyVector(int dim);
    MyVector(int dim, int value);
    void print();
};
```

Programming Design – Classes 19 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts  
Friends and static members

Constructors and the destructor  
Object pointers and the copy constructor

## Destructors

- A destructor is invoked right before an object is destroyed.
  - It must be public and have no parameter.
  - The compiler provides a default destructor that does nothing.
- To define your own destructor, use ~.
  - Typically we release dynamically allocated space in a destructor.

```
class MyVector
{
private:
    int n;
    int* m;
public:
    // ~MyVector();
};

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

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

int main()
{
    if (true)
        MyVector v1(1);
    // no memory leak
    return 0;
}
```

Programming Design – Classes 22 / 52 Ling-Chieh Kung (NTU IM)

```
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



# Getters and Setters in C++



// Getters	// Setters
<pre>int getHeight() {     return height; }</pre>	<pre>int setHeight(int h) {     height = h; }</pre>

## Static Member

Basic concepts	Constructors and the destructor
Friends and static members	Object pointers and the copy constructor

### Static members

- A class contains some instance variables and functions.
  - Each object has its own copy of instance variables and functions.
- A member variable/function may be an attribute/operation of a class.
  - When the attribute/operation is class-specific rather than object-specific.
  - A class-specific attribute/operation should be identical for all objects.
- These variables/functions are called static members.

Programming Design – Classes 29 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## Static members: an example

- In MS Windows, each window is an object.
  - Windows is written in C++.
  - Mac OS is written in Objective-C.
- Each window has some object-specific attributes.
- They also share one class-specific attribute: the color of their title bars.

```
class Window
{
private:
    int width;
    int height;
    int locationX;
    int locationY;
    int status; // 0: min, 1: usual, 2: max
    static int barColor; // 0: gray, ...
    // ...
public:
    static int getBarColor();
    static void setBarColor(int color);
    // ...
};
```

Programming Design – Classes 30 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## Static members: an example

- We have to initialize a static variable globally.
- To access static members, use class name::member name.

```
int Window::barColor = 0; // default
int Window::getBarColor()
{
    return barColor;
}
void Window::setBarColor(int color)
{
    barColor = color;
}
```

```
int main()
{
    Window w; // not used
    cout << Window::getBarColor();
    cout << "\n";
    Window::setBarColor(1);
    return 0;
}
```

Programming Design – Classes 31 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## Good programming style

- If one attribute should be identical for all objects, it should be declared as a static variable.
  - Do not make it an instance variable and try to maintain consistency.
- Do not use an object to invoke a static member.
  - This will confuse the reader.
- Use class name::member name even inside member function definition to show that it is a static member.

```
int Window::getBarColor()
{
    return Window::barColor;
}
```

Programming Design – Classes 33 / 52 Ling-Chieh Kung (NTU IM)

## Object Pointer

Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## Object pointers

- What we have done is to use an object to invoke instance functions.
  - E.g., `a.print()` where `a` is an object and `print()` is an instance function.
- If we have a pointer `ptrA` pointing to the object `a`, we may write `(*ptrA).print()` to invoke the instance function `print()`.
  - `*ptrA` returns the object `a`.
- To simplify this, C++ offers the member access operator `->`.
  - This is specifically for an object pointer to access its members.
  - `(*ptrA).print()` is equivalent to `ptrA->print()`.

```
int main()
{
    MyVector v(5);
    MyVector* ptrV = &v;
    v.print();
    ptrV->print();
    return 0;
}
```

Programming Design – Classes 38 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## Why object pointers? *MyVector v[5]*

- Object pointers can be more useful than pointers for basic data types. Why?
- When one creates an array of objects, only the default constructor may be invoked.
  - Creating an array of object pointers delays the invocation of constructors.
  - These pointers then point to dynamically allocated objects.
- Passing a pointer into a function can be more efficient than passing the object.
  - A pointer can be much smaller than an object.
  - Copying a pointer is easier than copying an object.
- Other reasons will be discussed in other lectures.

Programming Design – Classes 39 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## Passing object pointers into a function

- We may pass pointers rather than objects into this function:

```
MyVector sum(MyVector* v1, MyVector* v2, MyVector* v3)
{
    // assume that their dimensions are identical
    int n = v1->getN();
    int* sov = new int[n];
    for(int i = 0; i < n; i++)
        sov[i] = v1->getM(i) + v2->getM(i) + v3->getM(i);
    MyVector sumOfVec(n, sov);
    return sumOfVec;
}
```

- We need to create only one MyVector object in this function.
- Nevertheless, using pointers to access members requires more time.

Programming Design – Classes 44 / 52 Ling-Chieh Kung (NTU IM)

## Copy Constructor

Basic concepts Constructors and the destructor  
 Friends and static members Object pointers and the 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. `f(a1, a2, a3);`
  - When we assign an object to another object. `A a4 = a1;`
  - When we create an object with another object as the argument of the constructor. `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.

Programming Design - Classes 49 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
 Friends and static members Object pointers and the copy constructor

## 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;
}
      
```

Programming Design - Classes 50 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
 Friends and static members Object pointers and the copy constructor

## Copy constructors for MyVector

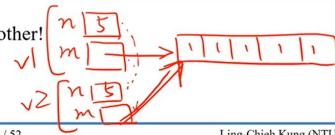
- For **MyVector**, one way to implement a copy constructor is
  - This has nothing different from the default copy constructor.
  - If **no member** is an **array/pointer**, the default copy constructor is fine.
- If there is any **array** or **pointer** member variable, the default copy constructor 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?
}
      
```



Programming Design - Classes 51 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## Deep copy

- To correctly copy a vector (by creating new values), we need to write our own copy constructor.
- We say that we implement “**deep copy**” by ourselves.
  - In the self-defined copy constructor, we **manually create another dynamic array**, set its elements’ values according to the original array, and use **m** to record its address.

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

Programming Design – Classes 52 / 52 Ling-Chieh Kung (NTU IM)

## Member Initializers

Motivations and prerequisites Comparison and indexing operators  
Assignment and self-assignment operators Addition operators

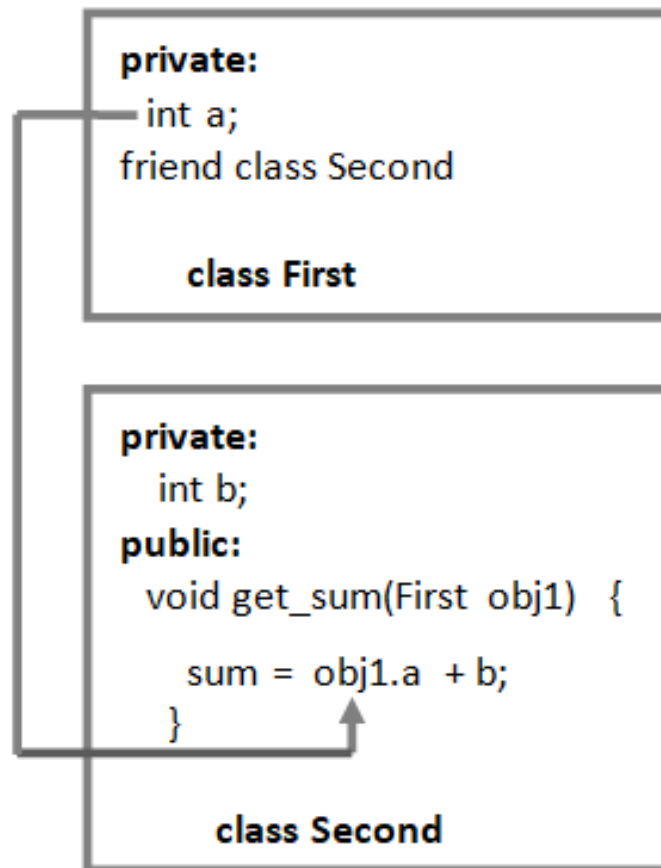
## Member initializers

- We need a **member initializer**.
  - A specific operation for initializing an instance variable.
  - Can also be used for initializing **non-constant instance variables**.
- Member initializers are used a lot in general.

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

Programming Design – Operator Overloading 17 / 49 Ling-Chieh Kung (NTU IM)

## Friend



## Friend function characteristics

- 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.



Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## friend for functions and classes

- Declare friends only if data hiding is preserved.
  - Do not set everything public!
  - Use structures rather than classes when nothing should be private (this is recommended but not required).
  - Be careful in offering public member functions (e.g., getters and setters).
- **friend** in fact **help you hide data**.
  - If a private member should be accessed only by another class/function, we should declare a friend instead of writing a getter/setter.

Programming Design – Classes 28 / 52 Ling-Chieh Kung (NTU IM)

Basic concepts Constructors and the destructor  
Friends and static members Object pointers and the copy constructor

## friend for functions and classes

- To “open” private members, another way is to declare “friends.”
- One class can allow its friends to access its private members.
- Its friends can be global functions or other classes.
  - Then inside `test()` and member functions of `Test`, those private members of `MyVector` can be accessed.
  - `MyVector` cannot access `Test`'s members.

```
class MyVector
{
// ...
friend void test();
friend class Test;
};
```

Programming Design – Classes 26 / 52 Ling-Chieh Kung (NTU IM)

```
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;

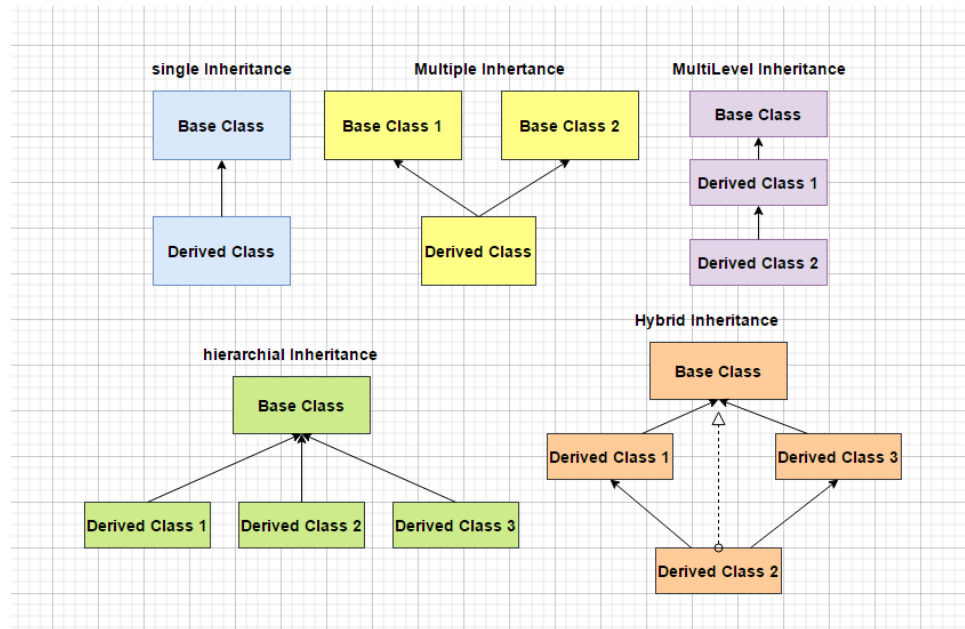
a1 += a3;
cout << "a1 + a3: ";
a1.print();
cout << endl;

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.

Inheritance An example Polymorphism

## Inheritance visibility

- In general, the visibility of a member in a child class depends on:
  - The member visibility by the parent.
  - The inheritance modifier.

Member visibility by the parent	Inheritance modifier		
	public	protected	private
✓ public	public	protected	private
✓ protected	protected	protected	private
private	private	private	private

- If you have no idea, just use public inheritance.

Programming Design – Inheritance and Polymorphism
26 / 62
Ling-Chieh Kung (NTU IM)

## Invoking parent class' constructors

- The parent class' constructor will not be inherited.
- One of them will be invoked **before** the child class' constructor is invoked.
- **✓** Create the parent before creating the child!
- **✓** If not specified, the parent's **default** constructor will be invoked.

```
MyVector: MyVector(): n(0), m(nullptr)
{
}
MyVector2D: MyVector2D()
{
  this->n = 2;
  // this->m = nullptr is redundant
}
```

```
int main()
{
  MyVector2D v;
  return 0;
}
```

## Invoking copy constructors

- How about the copy constructor?
- If we do **not** define one for the child, the system provides a **default** one.
- **✓** Before the child's default copy constructor is invoked, the parent's copy constructor will be **automatically** invoked.

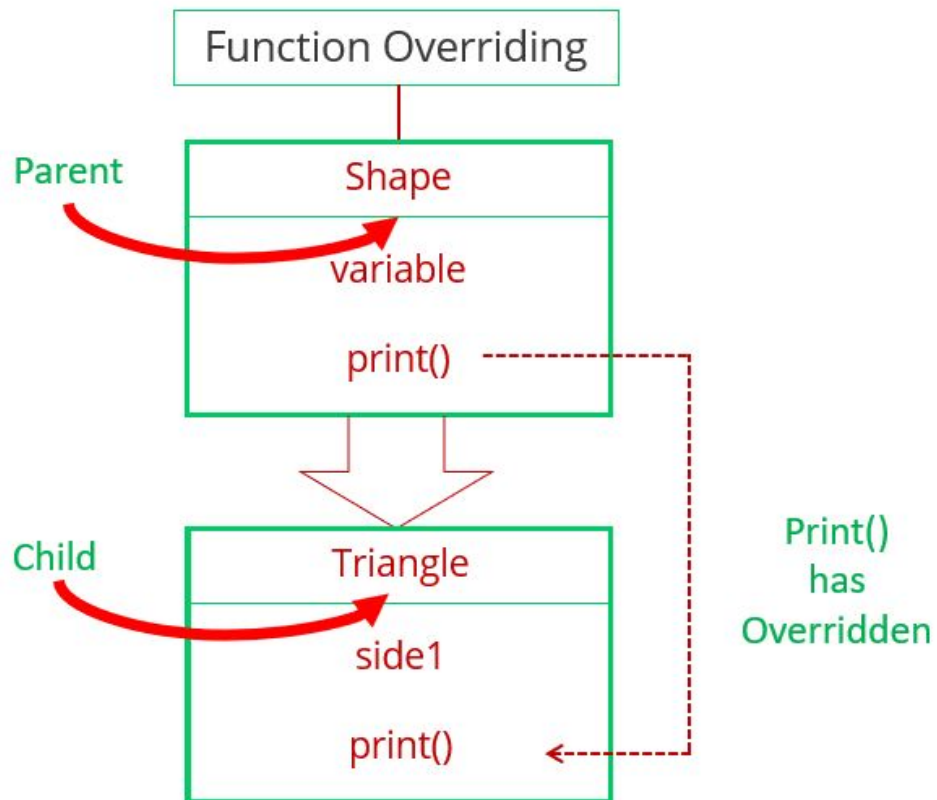
```
MyVector: MyVector(const MyVector& v)
{
  this->n = v.n;
  this->m = new double[n];
  for(int i = 0; i < n; i++)
    this->m[i] = v.m[i];
}
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
  // no copy constructor
};
```

## Invoking parent class' destructor

- When an object of the child class is to be destroyed:
  - First the child's destructor is invoked.
  - **Then** the parent's destructor is invoked **automatically**, even if we do not define a destructor for the child.

```
MyVector::~MyVector()
{
  delete [] m;
}
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
  // no destructor
};
```

## Overriding vs Overloading

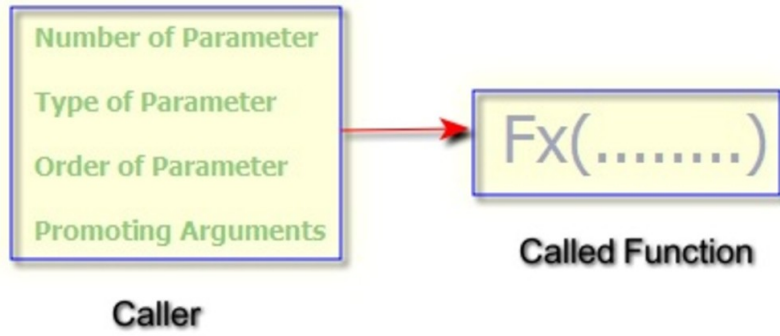


## Function signature and Parameters



- **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**.





## 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

## 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);
```

↓

**class B**

```
int add(int, int);
```

**Function Overriding**

## Overloading vs. Overriding

---

- Overriding a base class member function is similar to overloading a function or operator
  - But for overriding, definitions are distinguished by their scopes rather than by their signatures
- C++ can distinguish method definitions according to either static or dynamic type
  - Depends on whether a method is virtual or not
  - Depends on whether called via a reference or pointer vs. directly on an object
  - Depends on whether the call states the scope explicitly (e.g., `Foo::baz()` ;)

---

CSE 332: C++ Overloading

## Polymorphism

### Overview of C++ Polymorphism

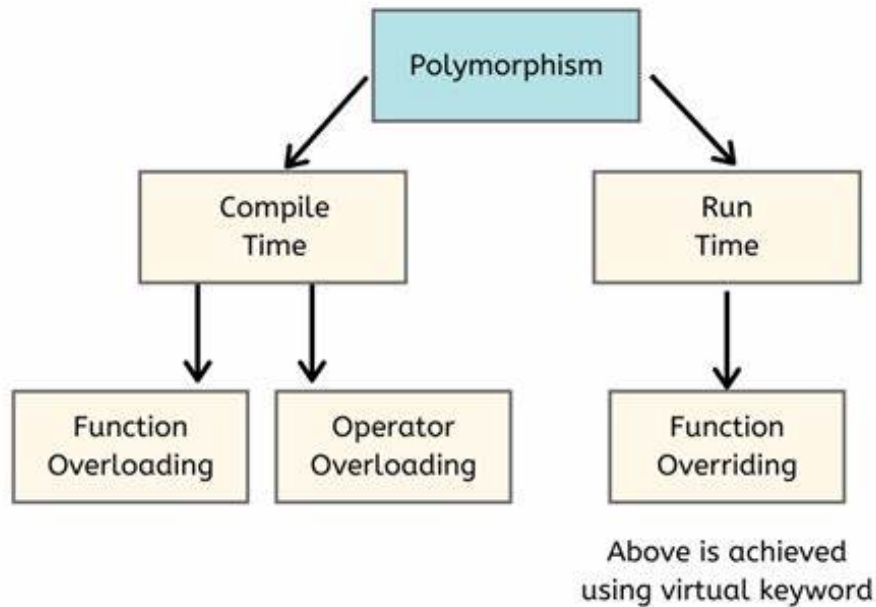
---

- Two main kinds of types in C++: native and user-defined
  - “User” defined types: declared classes, structs, unions
    - including types provided by the C++ standard libraries
  - Native types are “built in” to the C++ language itself: int, long, float, ...
  - A typedef creates a new type *name* for another type (type aliasing)
- Public inheritance creates sub-types
  - Inheritance only applies to user-defined classes (and structs)
  - A publicly derived class is-a subtype of its base class
  - Known as “inheritance polymorphism”
- Template parameters also induce a subtype relation
  - Known as “interface polymorphism”
  - We’ll cover how this works in depth, in later sessions
- Liskov Substitution Principle (for both kinds of polymorphism)
  - if S is a subtype of T, then wherever you need a T you can use an S

---

CSE 332: C++ Polymorphism

# Polymorphism in C++



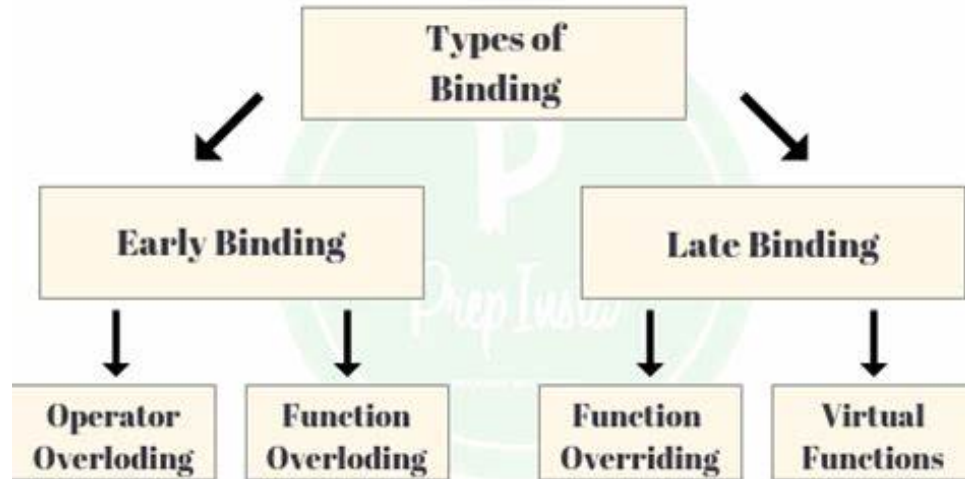
## Early Binding and Late Binding

### What is Binding ?

- **Binding** refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses.
- Although binding is used for both variables and functions, but we have to focus on function binding.

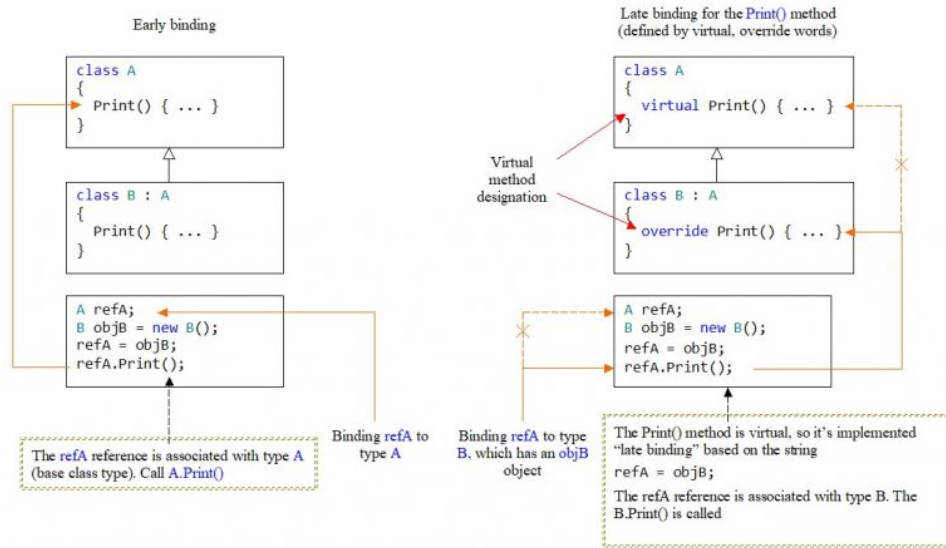


# Early binding & Late binding in C++



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.





Inheritance      An example      Polymorphism

## Early binding vs. late binding

- When we do `A a = b` or `A* a = &b`, we are using polymorphism.
- For `A a = b`, the system does **early binding**:
  - `a` occupies only four bytes for storing `i`.
  - `a` does not have a space for storing `j`.
  - Its type is set to be `A` at **compilation**.
- For `A* a = &b`, the system does **late binding**:
  - `a` is just a pointer.
  - It can point to an `A` object or a `B` object.
  - Its "type" can be set at the **run time**.

```

class A
{
protected:
    int i;
public:
    void a() { cout << "a\n"; }
    void f() { cout << "af\n"; }
};

class B : public A
{
private:
    int j;
public:
    void b() { cout << "b\n"; }
    void f() { cout << "bf\n"; }
};
    
```

Programming Design – Inheritance and Polymorphism      54 / 62      Ling-Chieh Kung (NTU IM)

## Virtual Function

Inheritance      An example      Polymorphism

## Virtual functions

- If we declare a parent's member function to be **virtual**, its invocation priority will be lower than a child's (if we use late binding).
  - A child cannot declare a parent's function as virtual (it is of no use).
- In summary, we need:
  - Late binding + virtual functions.

```

class Parent
{
protected:
    int x;
    int y;
public:
    Parent(int a, int b) : x(a), y(b) {}
    virtual void print() { cout << x << " " << y; }
};

class Child : public Parent
{
protected:
    int z;
public:
    Child(int a, int b, int c) : Parent(a, b)
    { z = c; }
    void print() { cout << z; }
};
    
```

Programming Design – Inheritance and Polymorphism      57 / 62      Ling-Chieh Kung (NTU IM)

Inheritance An example Polymorphism

## 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!

Programming Design – Inheritance and Polymorphism 59 / 62 Ling-Chieh Kung (NTU IM)

Inheritance An example Polymorphism

## 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”.

Programming Design – Inheritance and Polymorphism 62 / 62 Ling-Chieh Kung (NTU IM)

## 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;
};
```

CSE 332: C++ Polymorphism

- 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)

## 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

## this

Motivations and prerequisites
Comparison and indexing operators

Assignment and self-assignment operators
Addition operators

# this

- When you create an object, it occupies a memory space.

MyVector a1  
 int n = 5  
 double\* m

→ 1 2 3 4 5

- Inside an instance function, this is a pointer storing the address of that object.
  - `this` is a C++ keyword.

```

class A
{
private:
    int a;
public:
    void f() { cout << this << "\n"; }
    A* g() { return this; }
};

int main()
{
    A obj;
    cout << &obj << "\n"; // 0x9ffe40
    obj.f(); // 0x9ffe40
    cout << (&obj = obj.g()) << "\n"; // 1
    return 0;
}

```

Programming Design – Operator Overloading
10 / 49
Ling-Chieh Kung (NTU IM)

## 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;
};
```

## Member Initializers

### Member initializers

- We need a **member initializer**.
  - A specific operation for initializing an instance variable.
  - Can also be used for initializing non-constant instance variables.
- Member initializers are used a lot in general.

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

## 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 `operatorop`.
  - 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;
}
```

## Invoking overloaded operators

- We are indeed implementing instance functions with special names.
- Regarding **invoking** these instance functions:

```
int main() // without overloading
{
    double d1[5] = {1, 2, 3, 4, 5};
    const MyVector a1(5, d1);

    double d2[4] = {1, 2, 3, 4};
    const MyVector a2(4, d2);
    const MyVector a3(a1);

    cout << (a1.isEqual(a2) ? "Y" : "N");
    cout << "\n";
    cout << (a1.isEqual(a3) ? "Y" : "N");
    cout << "\n";

    return 0;
}
```

```
int main() // with overloading
{
    double d1[5] = {1, 2, 3, 4, 5};
    const MyVector a1(5, d1);

    double d2[4] = {1, 2, 3, 4};
    const MyVector a2(4, d2);
    const MyVector a3(a1);

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

    return 0;
}
```

## Preventing assignments and copying

- In some cases, we **disallow** assignments between objects of a certain class.
  - To do so, overload the assignment operator as a **private** member.
- In some cases, we disallow creating an object by **copying** another object.
  - To do so, implement the copy constructor as a **private** member.
- The copy constructor, assignment operator, and destructor form a group.
  - If there is no pointer, none of them is needed.
  - If there is a pointer, all of them are needed.

# Assignment Operator

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

<u>Method Name</u>	<u>Return Type</u>	<u>Description</u>	<u>Usage</u>
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()

C++ strings File I/O Self-defined header files

## string input: getline ()

- For `cin >>` to input into a C++ string, **white spaces** are still delimiters.
- To fix this, now we cannot use `cin.getline()`.
  - The first argument of `cin.getline()` must be a C string.
- We use a global function `getline()` defined in `<string>` instead:
 

```
string s;
getline(cin, s);
```

```
istream& getline(istream& is, string& str);
```
- By default, `getline()` stops when reading a newline character. We may specify the delimiter character we want:
 

```
string s;
getline(cin, s, '#');
```

```
istream& getline(istream& is, string& str, char delim);
```
- Note that there is no length limitation.

Programming Design – C++ Strings, File I/O, and Header Files 9 / 62 Ling-Chieh Kung (NTU IM)

## getline() vs >>

C++ strings File I/O Self-defined header files

## >> vs. getline ()

- The two operations are similar but different:
  - `>>` tries to convert the piece into the specified type; `getline()` simply store that piece as a C or C++ string.
  - `>>` stops at the first character not of that type; `getline()` stops at one character after the delimiter.
- Suppose that the text file now may contain the first name and last name of a student, separated by a white space.
  - We use a colon to separate a name and a score.
- How to write a program to calculate the sum of scores?
 

```
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 46 / 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;
            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				

GG

## find()

C++ strings File I/O Self-defined header files

## string finding

- We may use the member function `find()` to look for a string or character.
  - Just like `strstr()` and `strchr()` for C strings.

```
size_t find(const string& str, size_t pos = 0) const;
size_t find(const char* s, size_t pos = 0) const;
size_t find(char c, size_t pos = 0) const;
```

- This will return the beginning index of the argument, if it exists, or `string::npos` otherwise.

```
string s = "abdefg";
if (s.find("bod") != string::npos)
  cout << s.find("bod"); // 1
```

Programming Design – C++ Strings, File I/O, and Header Files 11 / 62 Ling-Chieh Kung (NTU IM)

## Insertion, Replacement, Erasing



C++ strings File I/O Self-defined header files

## Insertion, replacement, and erasing

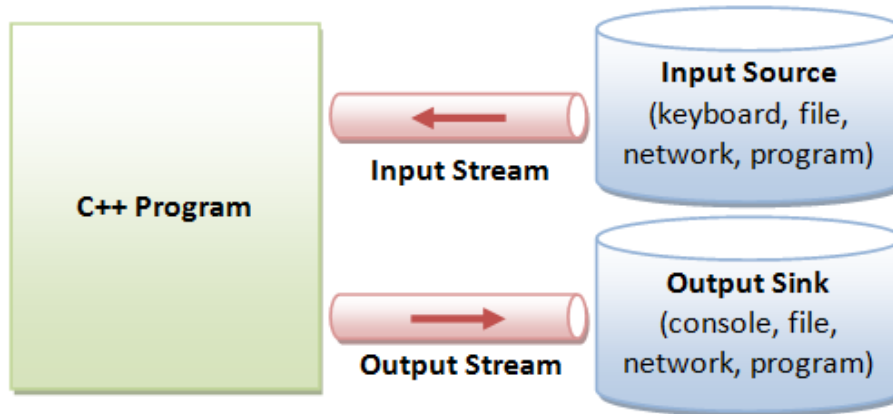
- We may use `insert()`, `replace()`, and `erase()` to modify a string.

```
string& insert(size_t pos, const string& str);
string& replace(size_t pos, size_t len, const string& str);
string& erase(size_t pos = 0, size_t len = npos);
```

```
int main()
{
    cout << "01234567890123456789\n";
    string myStr = "Today is not my day.";
    myStr.insert(9, "totally "); // Today is totally not my day.
    myStr.replace(17, 3, "NOT"); // Today is totally NOT my day.
    myStr.erase(17, 4); // Today is totally my day.
    cout << myStr << endl;
    return 0;
}
```

Programming Design – C++ Strings, File I/O, and Header Files 13 / 62 Ling-Chieh Kung (NTU IM)

## File Stream



### Internal Data Formats:

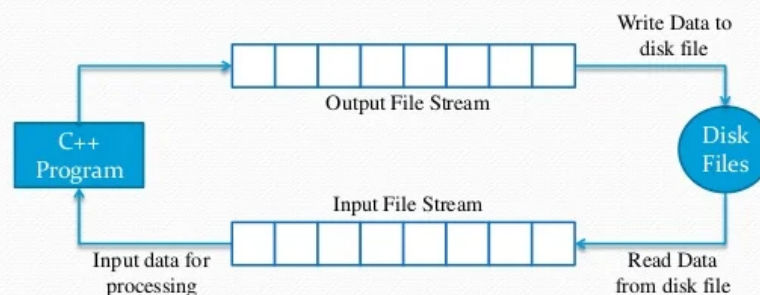
- Text: `char`, `wchar_t`
- `int`, `float`, `double`, etc.

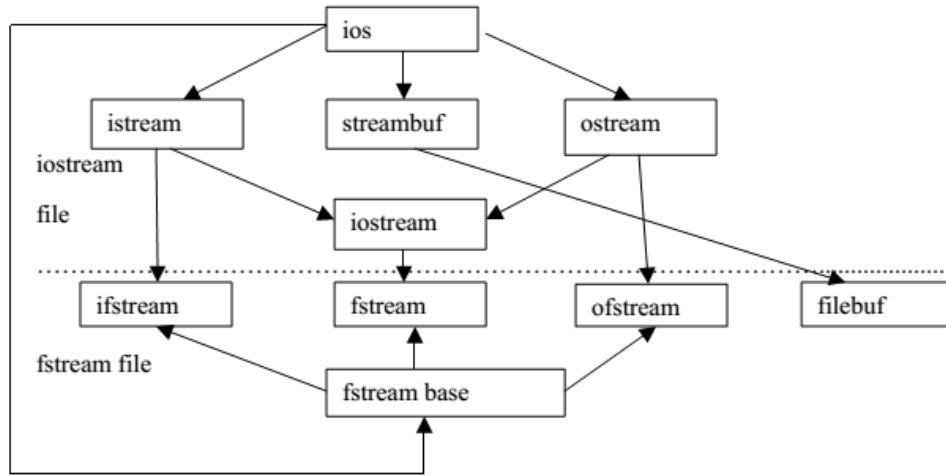
### External Data Formats:

- 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

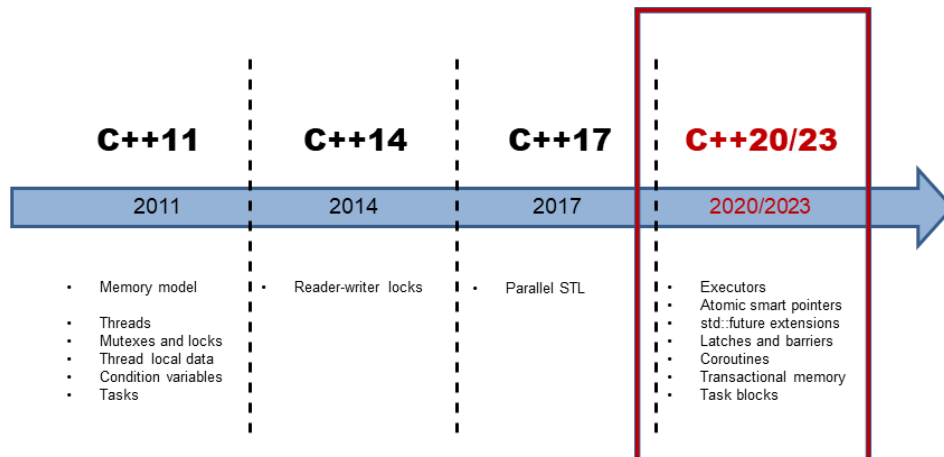
Object Oriented Programming in C++

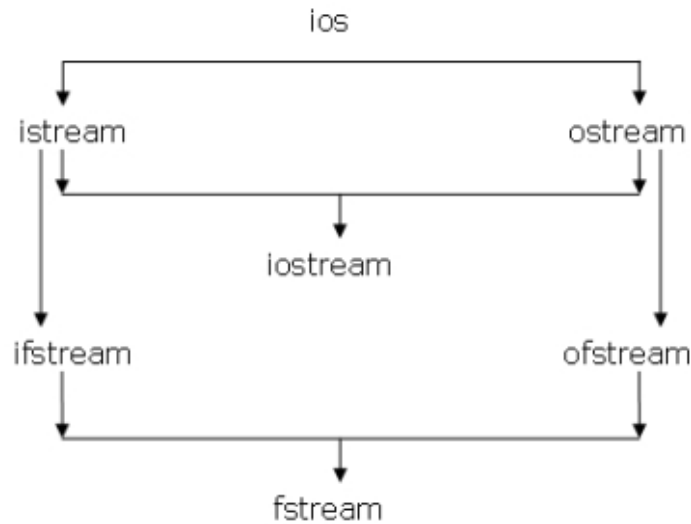
File Opening Mode

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(Standard Library)





## Self-defined Library

C++ strings File I/O Self-defined header files

### Libraries

- There are many C++ standard **libraries**.
  - `<iostream>`, `<fstream>`, `<cmath>`, `<cctype>`, `<string>`, etc.
- We may also want to define **our own libraries**.
  - Especially when we collaborate with others.
  - Typically, one implements classes or global functions for the others to use.
  - That function can be defined in a self-defined library.
- A library includes a header file (.h) and a source file (.cpp).
  - The header file contains declarations
  - The source file contains definitions.

Programming Design – C++ Strings, File I/O, and Header Files 53 / 62 Ling-Chieh Kung (NTU IM)

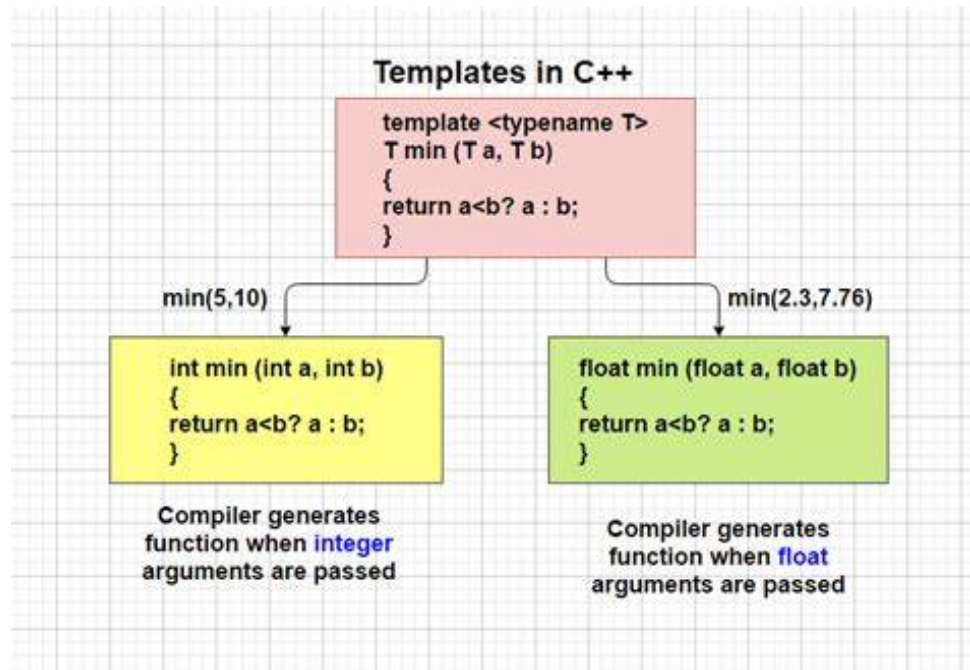
C++ strings File I/O Self-defined header files

### 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.

Programming Design – C++ Strings, File I/O, and Header Files 58 / 62 Ling-Chieh Kung (NTU IM)

## 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
};
                
```

Programming Design – Templates, Vectors, and Exceptions
Ling-Chieh Kung (NTU IM)

```

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

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

Compiler internally generates and adds below code

```

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

Compiler internally generates and adds below code.

```

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

```

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++;
    }
}

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;

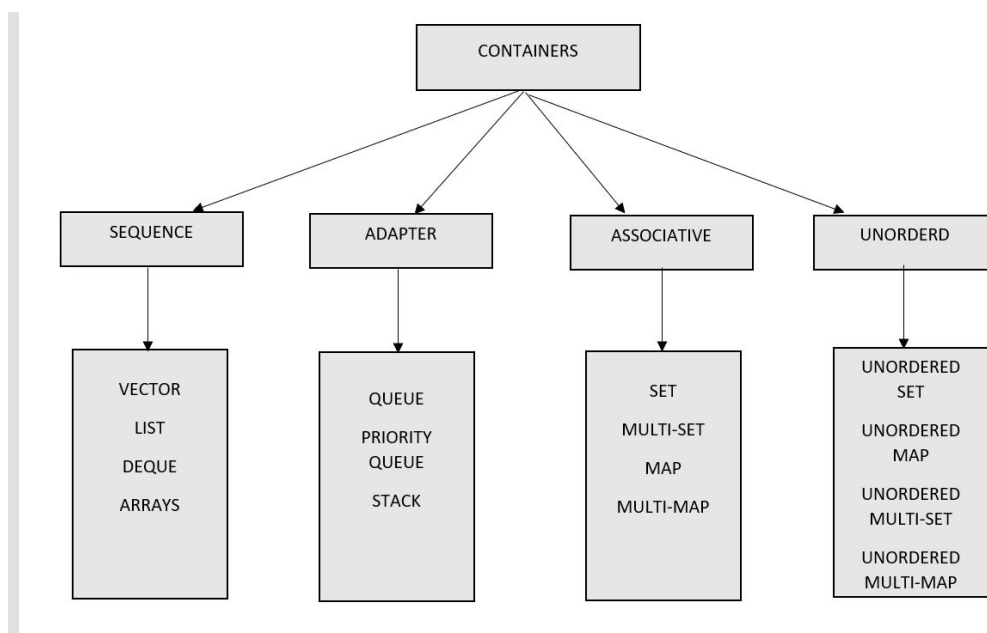
    Team<string> t1;
    t1.addWarrior("Alice", 1);
    t1.memberBeatMonster("Alice", 10000);
    t1.addWizard("Bob", 2);
    t1.printMember("Alice");

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

    return 0;
}

```

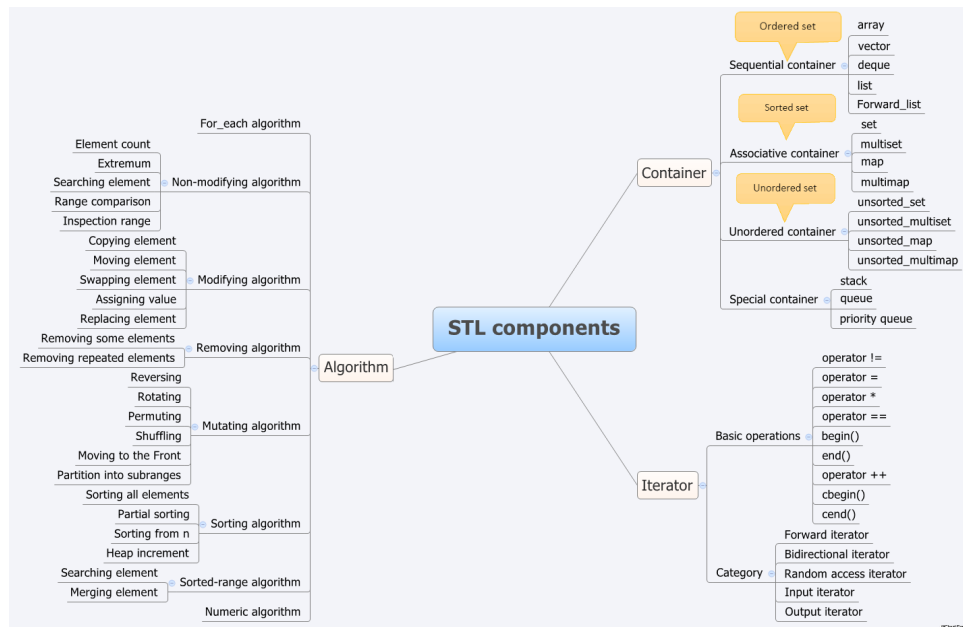
## C++ STL(Standard Template Library)

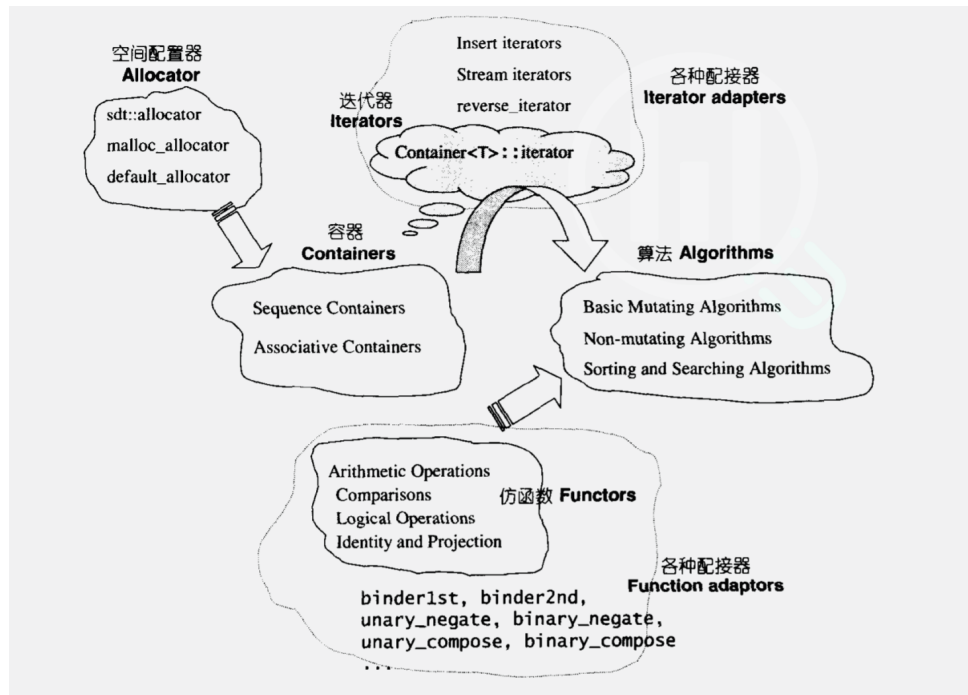


# Intro to the C++ Standard Template Library (STL)

- 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

© Ren-Song Tsay, NTHU, Taiwan

## Vector

The standard library <vector>

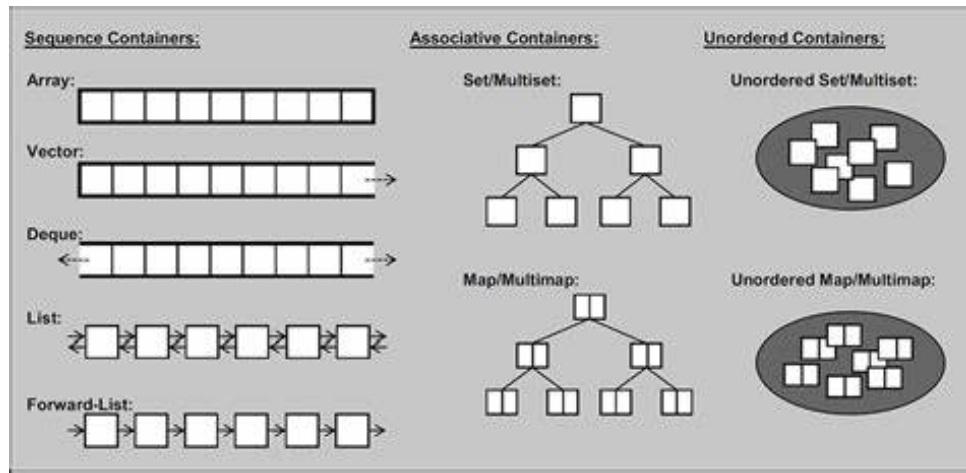
### The standard library <vector>

- A vector is very easy to use.
  - To create a vector, indicate the type of items:
 

```
vector<int> v1; // integer vector
vector<double> v2; // double vector
vector<Warrior> v3; // Warrior vector
```
- Member functions that modifies a vector:
  - `push_back()`, `pop_back()`, `insert()`, `erase()`, `swap()`, `=`, etc.
- Member functions for one to access a vector element:
  - `[], front()`, `back()`, etc.
- Member functions related to the capacity:
  - `size()`, `max_size()`, `resize()`, etc.

Templates
Exception handling

Programming Design – Templates, Vectors, and Exceptions
28 / 55
Ling-Chieh Kung (NTU IM)



## Exception Handling

Object Oriented Programming in C++

### Exception Handling in C++

- The process of converting system error messages into user friendly error message is known as **Exception handling**. This is one of the powerful feature of C++ to handle run time error and maintain normal flow of C++ application.
- **Exception**
- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's Instructions.

Lecture Slides By Adil Aslam

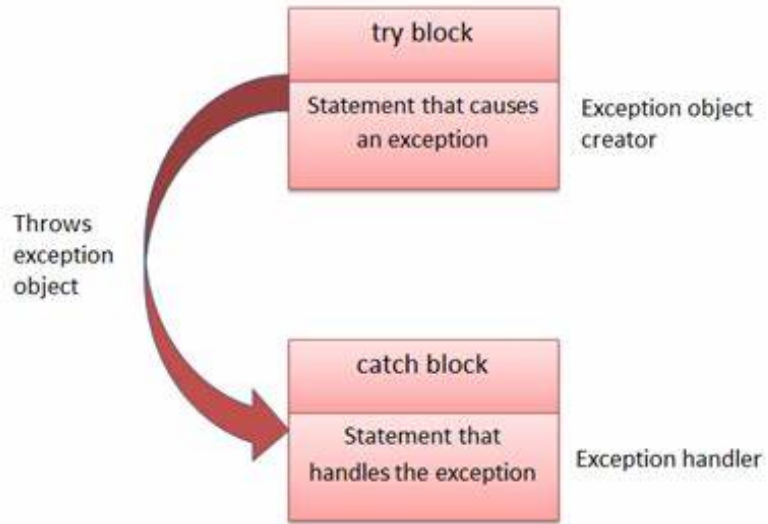
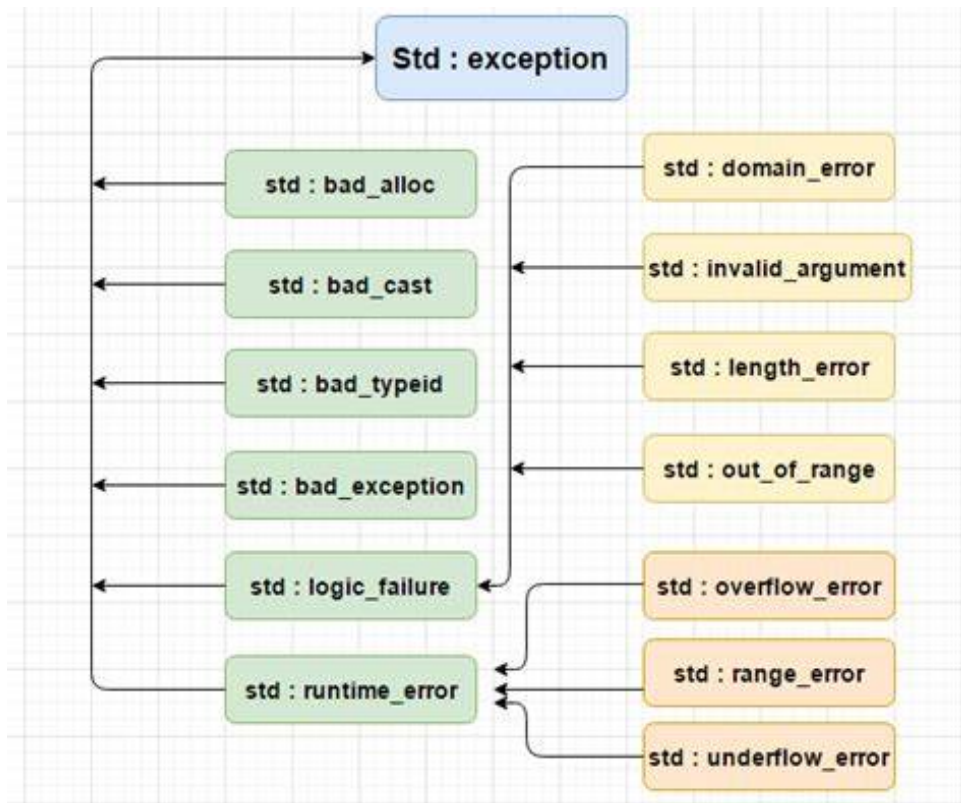


Fig: Exception Handling Mechanism



Templates The standard library <vector> Exception handling

## 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
  
```

Programming Design – Templates, Vectors, and Exceptions 47 / 55 Ling-Chieh Kung (NTU IM)

Templates The standard library <vector> Exception handling

## 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.

Programming Design – Templates, Vectors, and Exceptions 49 / 55 Ling-Chieh Kung (NTU IM)

Templates The standard library <vector> Exception handling

## 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;
}
  
```

Programming Design – Templates, Vectors, and Exceptions 50 / 55 Ling-Chieh Kung (NTU IM)



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 --