



Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C
Data Types
Variables
Literals
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output
Std Library

Organization

Build Process

References

Summary of
module 01

Module 01: Programming in C++

Recap of C

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C
Data Types
Variables
Literals
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Revisit the concepts of C language
- Revisit C Standard Library components
- Revisit the Organization and Build Process for C programs
- Create the foundation for the concepts of C++ with backward compatibility to C



Module Outline

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Recap of C features
 - Data types
 - Variables
 - Literals
 - Operators
 - Expressions
 - Statements
 - Control Constructs – Conditional Flow & Loops
 - Arrays
 - Structures & Unions
 - Pointers
 - Functions
 - Input / Output
- C Standard Library
- Source Organization for a C program
- Build Process



Module 01: Lecture 01

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Recap of C features

- Data types
- Variables
- Literals
- Operators
- Expressions
- Statements
- Control Constructs – Conditional Flow & Loops



First C program

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

● Print "Hello World"

Source Program

```
#include <stdio.h>

int main() {
    printf("Hello World");
    printf("\n");
    return 0;
}
```

- stdio.h header included for input / output
- main function is used to start execution
- printf function is used to print the string "Hello World"



Data Types

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

Data types in C are used for declaring variables and deciding on storage and computations:

- **Built-in / Basic** data types are used to define raw data

- char
- int
- float
- double

Additionally, C99 defines:

- bool

All data items of a given type has the same size (in bytes). The size is implementation-defined.

- **Enumerated Type** data are internally of int type and operates on a select subset.



Data Types

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

Data types in C further include:

- **void:** The type specifier `void` indicates no type.
- **Derived data types** include:
 - Array
 - Structure – `struct` & `union`
 - Pointer
 - Function
 - String – C-Strings are really not a type; but can be made to behave as such using functions from `<string.h>` in standard library
- **Type modifiers** include:
 - `short`
 - `long`
 - `signed`
 - `unsigned`



Variables

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- A variable is a name given to a storage area

- Declaration of Variables:

- Each variable in C has a specific type, which determines the size and layout of the storage (memory) for the variable
- The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore

```
int      i, j, noOfData;
char     c, endOfSession;
float    f, velocity;
double   d, dist_in_light_years;
```



Variables

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

● Initialization of Variables:

- Initialization is setting an initial value to a variable at its definition

```
int      i = 10, j = 20, numberOfWorkDays = 22;
char     c = 'x';
float    weight = 4.5;
double   density = 0.0;
```



Literals

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Literals refer to fixed values of a built-in type
- Literals can be of any of the basic data types

```
212      // (int) Decimal literal
0173     // (int) Octal literal
0b1010   // (int) Binary literal
0xF2     // (int) Hexadecimal literal
3.14     // (double) Floating-point literal
'x'      // (char) Character literal
"Hello"  // (char *) String literal
```

- In C99, literals are constant values having const types as:

```
212      // (const int) Decimal literal
0173     // (const int) Octal literal
0b1010   // (const int) Binary literal
0xF2     // (const int) Hexadecimal literal
3.14     // (const double) Floating-point literal
'x'      // (const char) Character literal
"Hello"  // (const char *) String literal
```



Operators

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- An operator denotes a specific operation. C has the following types of operators:

- Arithmetic Operators: + - * / % ++ --
- Relational Operators: == != > < >= <=
- Logical Operators: && || !
- Bit-wise Operators: & | ~ << >>
- Assignment Operators: = += -= *= /= ...
- Miscellaneous Operators: . , sizeof & * ?:

- **Arity of Operators:** Number of operand(s) for an operator

- +, -, *, & operators can be *unary* (1 operand) or *binary* (2 operands)
- ==, !=, >, <, >=, <=, &&, ||, +=, -=, *=, =, /=, &, |, <<, >> can work only as *binary* (2 operands) operators
- sizeof ! ~ ++ -- can work only as *unary* (1 operand) operators
- ?: works as *ternary* (3 operands) operator. The condition is the first operand and the if true logic and if false logic corresponds to the other two operands.



Operators

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- **Operator Precedence:** Determines which operator will be performed first in a chain of different operators

The precedence of all the operators mentioned above is in the following order: (left to right – Highest to lowest precedence)

(), [], ++, −, + (unary), −(unary), !~, *, &, sizeof, *, /, %, +, −, < <, > >, ==, !=, *=, =, /=, &, |, &&, | |, ?:, =, +=, -=, *=, =, /=, < <=, > >=

- **Operator Associativity:** Indicates in what order operators of equal precedence in an expression are applied

- Consider the expression $a \sim b \sim c$. If the operator \sim has left associativity, this expression would be interpreted as $(a \sim b) \sim c$. If the operator has right associativity, the expression would be interpreted as $a \sim (b \sim c)$.

- Right-to-Left: ?:, =, +=, -=, *=, =, /=, <<=, >>=, −, +−, !~, *, &, sizeof
- Left-to-Right: *, /, %, +, −, <<, >>, ==, !=, *=, =, /=, &, |, &&, ||



Expressions

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Every expression has a value
 - A literal is an expression
 - A variable is an expression
 - One, two or three expression/s connected by an operator (of appropriate arity) is an expression
 - A function call is an expression

- Examples:

- For

```
int i = 10, j = 20, k;
int f(int x, int y) { return x + y; }
```

- Expression are:

```
2.5          // Value = 2.5
i            // Value 10
-i           // Value -10
i - j        // Value -10
k = 5         // Value 5
f(i, j)       // Value 30
i + j == i * 3 // Value true
(i == j)? 1: 2 // Value 2
```



Statement

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- A statement is a command for a specific action. It has no value
 - A ; (semicolon) is a (null) statement
 - An expression terminated by a ; (semicolon) is a statement
 - A list of one or more statements enclosed within a pair of curly braces { and } or block is a compound statement
 - Control constructs like if, if-else, switch, for, while, do-while, goto, continue, break, return are statements
- Example: *Expression statements*

Expressions	Statements
i + j	i + j;
k = i + j	k = i + j;
funct(i,j)	funct(i,j);
k = funct(i,j)	k = funct(i,j);

- Example: *Compound statements*

```
{  
    int i = 2, j = 3, t;  
  
    t = i;  
    i = j;  
    j = t;  
}
```



Control Constructs

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- These statements control the flow based on conditions:

- Selection-statement:* if, if-else, switch
- Labeled-statement:* Statements labeled with identifier, case, or default
- Iteration-statement:* for, while, do-while
- Jump-statement:* goto, continue, break, return

- Examples:

```
if (a < b) {  
    int t;  
  
    t = a;  
    a = b;  
    b = t;  
}
```

```
int sum = 0;  
for(i = 0; i < 5; ++i) {  
    int j = i * i;  
    sum += j;  
}
```

```
if (x < 5)  
    x = x + 1;  
else {  
    x = x + 2;  
    --y;  
}
```

```
while (n) {  
    sum += n;  
    if (sum > 20)  
        break;  
    --n;  
}
```

```
switch (i) {  
    case 1: x = 5;  
    break;  
    case 3: x = 10;  
    default: x = 15;  
}
```

```
int f(int x, int y)  
{  
    return x + y;  
}
```



Module 01: End of Lecture 01

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Recap of C features

- Data types
- Variables
- Literals
- Operators
- Expressions
- Statements
- Control Constructs – Conditional Flow & Loops



Module 01: Lecture 02

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Recap of C features

- Arrays
- Structures
- Unions
- Pointers



Arrays

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- An array is a collection of data items, all of the same type, accessed using a common name

- Declare Arrays:

```
#define SIZE 10
int name[SIZE];      // SIZE must be an integer constant greater than zero
double balance[10];
```

- Initialize Arrays:

```
int primes[5] = {2, 3, 5, 7, 11}; // Size = 5

int primes[] = {2, 3, 5, 7, 11};
int sizeOfPrimes = sizeof(primes)/sizeof(int); // size is 5 by initialization

int primes[5] = {2, 3};           // Size = 5, last 3 elements set to 0
```

- Access Array elements:

```
int primes[5] = {2, 3};
int EvenPrime = primes[0]; // Read 1st element
primes[2] = 5;           // Write 3rd element
```

- Multidimensional Arrays:

```
int mat[3][4];

for(i = 0; i < 3; ++i)
    for(j = 0; j < 4; ++j)
        mat[i][j] = i + j;
```



Structures

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- A structure is a collection of data items of different types. Data items are called *members*. The size of a structure is the sum of the size of its members.

- Declare Structures:**

```
struct Complex { // Complex Number
    double re;    // Real component
    double im;    // Imaginary component
} c;           // c is a variable of struct Complex type
printf("size = %d\n", sizeof(struct Complex)); // Prints: size = 16

typedef struct _Books {
    char title[50];   // data member
    char author[50];  // data member
    int book_id;      // data member
} Books; // Books is an alias for struct _Books type
```

- Initialize Structures:**

```
struct Complex x = {2.0, 3.5}; // Both members
struct Complex y = {4.2};     // Only the first member
```

- Access Structure members:**

```
struct Complex x = {2.0, 3.5};
double norm = sqrt(x.re*x.re + x.im*x.im); // Using . (dot) operator

Books book;
book.book_id = 6495407;
strcpy(book.title, "C Programming");
```



Unions

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- A union is a special structure that allocates memory only for the largest data member and holds only one member at a time

- Declare Union:

```
typedef union _Packet { // Mixed Data Packet
    int     iData;        // integer data
    double dData;        // floating point data
    char   cData;        // character data
} Packet;
printf("size = %d\n", sizeof(Packet)); // Prints: size = 8
```

- Initialize Union:

```
Packer p = {10}; // Initialize only with a value of the type of first member
printf("iData = %d\n", p.iData); // Prints: iData = 10
```

- Access Union members:

```
p.iData = 2;
printf("iData = %d\n", p.iData); // Prints: iData = 2
p.dData = 2.2;
printf("dData = %lf\n", p.dData); // Prints: dData = 2.200000
p.cData = 'a';
printf("cData = %c\n", p.cData); // Prints: cData = a

p.iData = 97;
printf("iData = %d\n", p.iData); // Prints: iData = 97
printf("dData = %lf\n", p.dData); // Prints: dData = 2.199999
printf("cData = %c\n", p.cData); // Prints: cData = a
```



Pointers

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- A pointer is a variable whose value is a memory address
- The type of a pointer is determined by the type of its pointee

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch;    // pointer to a character
```

- Using a pointer:

```
int main() {
    int i = 20;    // variable declaration
    int *ip;        // pointer declaration
    ip = &i;        // store address of i in pointer

    printf("Address of variable: %p\n", &i); // Prints: Address of variable : 00A8F73C
    printf("Value of pointer: %p\n", ip);     // Prints: Value of pointer : 00A8F73C
    printf("Value of pointee: %d\n", *ip);    // Prints: Value of pointee : 20
    return 0;
}
```



Pointers

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

• Pointer-Array Duality

```
int a[] = {1, 2, 3, 4, 5};  
int *p;  
  
p = a;  
printf("a[0] = %d\n", *p);      // a[0] = 1  
printf("a[1] = %d\n", *(p+1)); // a[1] = 2  
printf("a[2] = %d\n", *(p+2)); // a[2] = 3  
  
p = &a[2];  
*p = -10;  
printf("a[2] = %d\n", a[2]);    // a[2] = -10
```

• Pointer to a structure

```
struct Complex { // Complex Number  
    double re;    // Real component  
    double im;    // Imaginary component  
} c = { 0.0, 0.0 };  
  
struct Complex *p = &c;  
  
(*p).re = 2.5;  
p->im = 3.6;  
  
printf("re = %lf\n", c.re); // re = 2.500000  
printf("im = %lf\n", c.im); // im = 3.600000
```

• malloc-free

```
int *p = (int *)malloc(sizeof(int));  
  
*p = 0x8F7E1A2B;  
printf("%X\n", *p);    // 8F7E1A2B  
  
unsigned char *q = p;  
printf("%X\n", *q++); // 2B  
printf("%X\n", *q++); // 1A  
printf("%X\n", *q++); // 7E  
printf("%X\n", *q++); // 8F  
  
free(p);
```

• Dynamically allocated arrays

```
int *p = (int *)malloc(sizeof(int)*3);  
  
p[0] = 1; p[1] = 2; p[2] = 3;  
  
printf("p[1] = %d\n", *(p+1)); // p[1] = 2  
free(p);
```



Module 01: End of Lecture 02

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

● Recap of C features

- Arrays
- Structures
- Unions
- Pointers



Module 01: Lecture 03

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Recap of C features
 - Functions
 - Input / Output
- C Standard Library
- Source Organization for a C program
- Build Process



Functions

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- A function performs a specific task or computation
 - Has 0, 1, or more parameters / arguments. Every argument has a type (void for no argument)
 - May or may not return a result. Return value has a type (void for no result)
 - Function declaration:

```
// Function Prototype / Header / Signature
// Name of the function: funct
// Parameters: x and y. Types of parameters: int
// Return type: int

int funct(int x, int y);
```

- Function definition:

```
// Function Implementation
int funct(int x, int y)

// Function Body
{
    return (x + y);
}
```



Functions

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- **Call-by-value** mechanism for passing arguments. The value of an actual parameter copied to the formal parameter
- **Return-by-value** mechanism to return the value, if any.

```
int funct(int x, int y) {
    ++x; ++y;                      // Formal parameters changed
    return (x + y);
}

int main() {
    int a = 5, b = 10, z;

    printf("a = %d, b = %d\n", a, b); // prints: a = 5, b = 10

    z = funct(a, b); // function call by value
                      // a copied to x. x becomes 5
                      // b copied to y. y becomes 10
                      // x in funct changes to 6 (++x)
                      // y in funct changes to 11 (++y)
                      // return value (x + y) copied to z

    printf("funct = %d\n", z); // prints: funct = 17

    // Actual parameters do not change on return (call-by-value)
    printf("a = %d, b = %d\n", a, b); // prints: a = 5, b = 10

    return 0;
}
```



Functions

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- A function may be recursive (call itself)

- Has recursive step/s
- Has exit condition/s

- Example:

```
// Factorial of n
unsigned int factorial(unsigned int n) {
    if (n > 0)
        return n * factorial(n - 1); // Recursive step
    else
        return 1; // Exit condition
}

// Number of 1's in the binary representation of n
unsigned int nOnes(unsigned int n) {
    if (n == 0)
        return 0; // Exit condition
    else // Recursive steps
        if (n % 2 == 0)
            return nOnes(n / 2);
        else
            return nOnes(n / 2) + 1;
}
```



Function pointers

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

```
#include <stdio.h>
struct GeoObject {
    enum { CIR = 0, REC, TRG } gCode;
    union {
        struct Cir { double x, y, r; } c;
        struct Rec { double x, y, w, h; } r;
        struct Trg { double x, y, b, h; } t;
    };
};

typedef void(*DrawFunc) (struct GeoObject);

void drawCir(struct GeoObject go) {
    printf("Circle: (%lf, %lf, %lf)\n",
           go.c.x, go.c.y, go.c.r);
}

void drawRec(struct GeoObject go) {
    printf("Rect: (%lf, %lf, %lf, %lf)\n",
           go.r.x, go.r.y, go.r.w, go.r.h);
}

void drawTrg(struct GeoObject go) {
    printf("Triag: (%lf, %lf, %lf, %lf)\n",
           go.t.x, go.t.y, go.t.b, go.t.h);
}
```

```
DrawFunc DrawArr[] = { // Array of func. ptrs
    drawCir, drawRec, drawTrg };

int main() {
    struct GeoObject go;
    go.gCode = CIR;
    go.c.x = 2.3; go.c.y = 3.6;
    go.c.r = 1.2;
    DrawArr[go.gCode](go); // Call by ptr

    go.gCode = REC;
    go.r.x = 4.5; go.r.y = 1.9;
    go.r.w = 4.2; go.r.h = 3.8;
    DrawArr[go.gCode](go); // Call by ptr

    go.gCode = TRG;
    go.t.x = 3.1; go.t.y = 2.8;
    go.t.b = 4.4; go.t.h = 2.7;
    DrawArr[go.gCode](go); // Call by ptr

    return 0;
}
```

```
Circle: (2.300000, 3.600000, 1.200000)
Rect: (4.500000, 1.900000, 4.200000, 3.800000)
Triag: (3.100000, 2.800000, 4.400000, 2.700000)
```



Input / Output

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- `int printf(const char *format, ...)` writes to `stdout` by the format and returns the number of characters written
- `int scanf(const char *format, ...)` reads from `stdin` by the format and returns the number of characters read
- Use `%s`, `%d`, `%c`, `%lf`, to print/scan string, int, char, double

```
#include <stdio.h>

int main() {

    char str[100];
    int i;

    printf("Enter a value :");           // prints a constant string
    scanf("%s %d", str, &i);           // scans a string value and an integer value
    printf("You entered: %s %d ", str, i); // prints string and integer
    return 0;
}
```



Input / Output

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- To write to or read from file:

```
#include <stdio.h>

int main() {

    FILE *fp = NULL;
    int i;

    fp = fopen("Input.dat", "r");
    fscanf(fp, "%d", &i);           // scan from Input.dat
    fclose(fp);

    fp = fopen("Output.dat", "w");
    fprintf("%d^2 = %d\n", i, i*i); // prints to Output.dat
    fclose(fp);

    return 0;
}
```



C Standard Library

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

● Common Library Components:

Component	Data Types, Manifest Constants, Macros, Functions, ...
stdio.h	Formatted and un-formatted file input and output including functions <ul style="list-style-type: none">• printf, scanf, fprintf, fscanf, sprintf, sscanf, feof, etc.
stdlib.h	Memory allocation, process control, conversions, pseudo-random numbers, searching, sorting <ul style="list-style-type: none">• malloc, free, exit, abort, atoi, strtold, rand, bsearch, qsort, etc.
string.h	Manipulation of C strings and arrays <ul style="list-style-type: none">• strcat, strcpy, strcmp, strlen, strtok, memcpy, memmove, etc.
math.h	Common mathematical operations and transformations <ul style="list-style-type: none">• cos, sin, tan, acos, asin, atan, exp, log, pow, sqrt, etc.
errno.h	Macros for reporting and retrieving error conditions through error codes stored in a static memory location called errno <ul style="list-style-type: none">• EDOM (parameter outside a function's domain – sqrt(-1)),• ERANGE (result outside a function's range), or• EILSEQ (an illegal byte sequence), etc.



Source Organization for a C program

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C
Data Types
Variables
Literals
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output
Std Library

Organization

Build Process

References

Summary of
module 01

Header Files

- A header file has extension .h and contains C function declarations and macro definitions to be shared between several source files
- There are two types of header files:
 - Files that the programmer writes
 - Files from standard library
- Header files are included using the #include pre-processing directive
 - `#include <file>` for system header files
 - `#include "file"` for header files of your own program



Source Organization for a C program

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

● Example:

```
// Solver.h -- Header files
int quadraticEquationSolver(double, double, double, double*, double*);  
  
// Solver.c -- Implementation files
#include "Solver.h"  
  
int quadraticEquationSolver(double a, double b, double c, double* r1, double* r2) {
    // ...
    // ...
    // ...
    return 0;
}  
  
// main.c -- Application files
#include "Solver.h"  
  
int main() {
    double a, b, c;
    double r1, r2;  
  
    int status = quadraticEquationSolver(a, b, c, &r1, &r2);
    return 0;
}
```



Build Flow

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

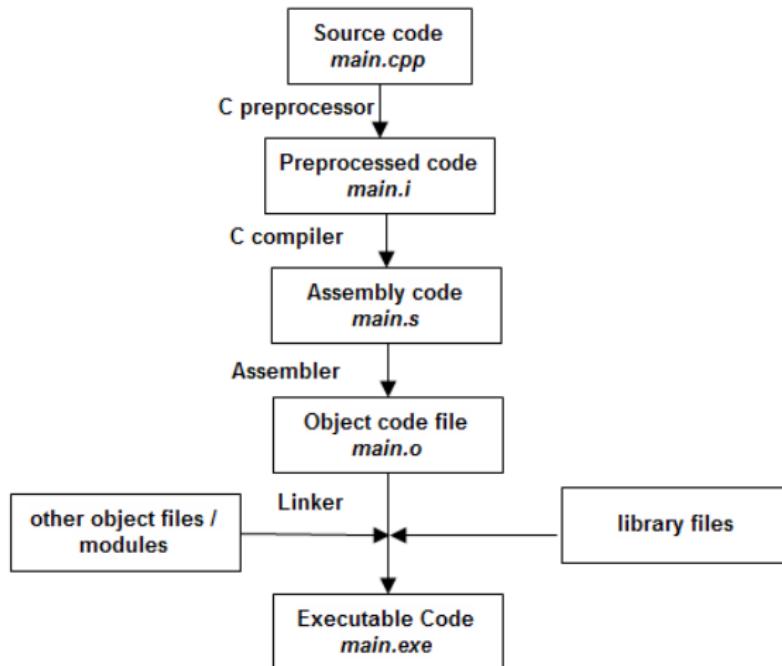
Std Library

Organization

Build Process

References

Summary of
module 01





Build Process

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- C Pre-processor (CPP) substitutes and includes functions, headers and macros before compilation

```
int sum(int, int);
int main() {
    int a = sum(1,2);
    return a;
}
```

- The compiler translates the pre-processed C code into assembly language, which is a machine level code that contains instructions that manipulate the memory and processor directly
- The linker links our program with the pre-compiled libraries for using their functions
- In the running example, `function.c` and `main.c` are first compiled and then linked

```
int sum(int a,int b) { return a+b; }

int main() {
    int a = sum(1,2); // as files are linked, uses functions directly
    return a;
}
```



Tools

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Development IDE: Code::Blocks 16.01
- Compiler: -std=c++98 and -std=c99



References

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Kernighan, Brian W., and Dennis M. Richie. *The C Programming Language*. Vol. 2. Englewood Cliffs: Prentice-Hall, 1988.
- King, Kim N., and Kim King. *C programming: A Modern Approach*. Norton, 1996.



Module Summary

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C

Data Types

Variables

Literals

Operators

Expressions

Statements

Control Flow

Arrays

Structures

Unions

Pointers

Functions

Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

- Revised the concept of variables and literals in C
- Revised the various data types and operators of C
- Re-iterated through the control constructs of C
- Re-iterated through the concepts of functions and pointers of C
- Re-iterated through the program organization of C and the build process.



Instructor and TAs

Module 01

Partha Pratim
Das

Objectives &
Outline

Recap of C
Data Types
Variables
Literals
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output

Std Library

Organization

Build Process

References

Summary of
module 01

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 02

Partha Pratim
Das

Objectives &
Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

Module 02: Programming in C++

Programs with IO & Loop

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 02

Partha Pratim
Das

Objectives &
Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

- Understand differences between C and C++ programs
- Appreciate the ease of programming in C++



Module Outline

Module 02

Partha Pratim
Das

Objectives &
Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

- Contrast differences between C and C++ programs for:
 - I/O
 - Variables
 - Using math library
 - Standard Library – Headers
 - Loop
 - bool type



Program 02.01: Hello World

Module 02

Partha Pratim
Das

Objectives & Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

	C Program	C++ Program
	<pre>// FileName:HelloWorld.c: #include <stdio.h> int main() { printf("Hello World in C"); printf("\n"); return 0; }</pre>	<pre>// FileName:HelloWorld.cpp: #include <iostream> int main() { std::cout << "Hello World in C++"; std::cout << std::endl; return 0; }</pre>
	Hello World in C	Hello World in C++
	<ul style="list-style-type: none">• IO Header is stdio.h• printf to print to console• Console is stdout file• printf is a variadic function• \n to go to the new line• \n is escaped newline character	<ul style="list-style-type: none">• IO Header is iostream• operator<< to stream to console• Console is std::cout ostream (in std namespace)• operator<< is a binary operator• std::endl (in std namespace) to go to the new line• std::endl is stream manipulator (newline) functor



Program 02.02: Add two numbers

Module 02

Partha Pratim
Das

Objectives & Outline

Hello World
Add numbers

Square Root
Standard Library
Sum Numbers
Using bool

Summary

C Program	C++ Program
<pre>// FileName:Add_Num.c: #include <stdio.h> int main() { int a, b; int sum; printf("Input two numbers:\n"); scanf("%d%d", &a, &b); sum = a + b; printf("Sum of %d and %d", a, b); printf(" is: %d\n", sum); return 0; }</pre>	<pre>// FileName:Add_Num_c++.cpp: #include <iostream> int main() { int a, b; std::cout << "Input two numbers:\n"; std::cin >> a >> b; int sum = a + b; // Declaration of sum std::cout << "Sum of " << a << " and " << b << " is: " << sum << std::endl; return 0; }</pre>
<pre>Input two numbers: 3 4 Sum of 3 and 4 is: 7</pre>	<pre>Input two numbers: 3 4 Sum of 3 and 4 is: 7</pre>
<ul style="list-style-type: none">• <code>scanf</code> to <code>scan (read)</code> from console• Console is <code>stdin</code> file• <code>scanf</code> is a variadic function• Addresses of <code>a</code> and <code>b</code> needed in <code>scanf</code>• All variables <code>a</code>, <code>b</code> & <code>sum</code> declared first (C89)• Formatting (<code>%d</code>) needed for variables	<ul style="list-style-type: none">• <code>operator>></code> to <code>stream</code> from console• Console is <code>std::cin</code> <code>istream</code> (in <code>std</code> namespace)• <code>operator>></code> is a binary operator• <code>a</code> and <code>b</code> can be directly used in <code>operator>></code> operator• <code>sum</code> may be declared when needed• Formatting is derived from type (<code>int</code>) of variables



Program 02.03: Square Root of a number

Module 02

Partha Pratim
Das

Objectives &
Outline

Hello World
Add numbers

Square Root
Standard Library
Sum Numbers
Using bool

Summary

C Program	C++ Program
<pre>// FileName:Sqrt.c: #include <stdio.h> #include <math.h> int main() { double x; double sqrt_x; printf("Input number:\n"); scanf("%lf", &x); sqrt_x = sqrt(x); printf("Sq. Root of %lf is:", x); printf("\n%lf", sqrt_x); return 0; }</pre>	<pre>// FileName:Sqrt_c++.cpp: #include <iostream> #include <cmath> using namespace std; int main() { double x; cout << "Input number:" << endl; cin >> x; double sqrt_x = // Declaration of sqrt_x sqrt(x); cout << "Sq. Root of " << x; cout << " is: " << sqrt_x << endl; return 0; }</pre>
<pre>Input number: 2 Square Root of 2.000000 is: 1.414214</pre>	<pre>Input number: 2 Square Root of 2 is: 1.41421</pre>
<ul style="list-style-type: none">Math Header is <code>math.h</code> (C Standard Library)Formatting (<code>%lf</code>) needed for variables<code>sqrt</code> function from C Standard LibraryDefault precision in print is 6	<ul style="list-style-type: none">Math Header is <code>cmath</code> (C Standard Library in C++)Formatting is derived from type (<code>double</code>) of variables<code>sqrt</code> function from C Standard LibraryDefault precision in print is 5 (different)



namespace std for C++ Standard Library

Module 02

Partha Pratim
Das

Objectives & Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

C Standard Library	C++ Standard Library
<ul style="list-style-type: none">• All names are global• <code>stdout</code>, <code>stdin</code>, <code>printf</code>, <code>scanf</code>	<ul style="list-style-type: none">• All names are within <code>std</code> namespace• <code>std::cout</code>, <code>std::cin</code>• Use <code>using namespace std;</code> to get rid of writing <code>std::</code> for every standard library name
W/o using	W/ using
<pre>#include <iostream> int main() { std::cout << "Hello World in C++" << std::endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; int main() { cout << "Hello World in C++" << endl; return 0; }</pre>



Standard Library Header Conventions

Module 02

Partha Pratim
Das

Objectives &
Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

	C Header	C++ Header
C Program	Use .h. Example: <code>#include <stdio.h></code> <i>Names in global namespace</i>	Not applicable
C++ Program	Prefix c, no .h. Example: <code>#include <cstdio></code> <i>Names in std namespace</i>	No .h. Example: <code>#include <iostream></code>

- Any C standard library header is to be used in C++ with a prefix 'c' and without the .h. These symbols will be in std namespace. Like:

```
#include <cmath> // In C it is <math.h>
...
std::sqrt(5.0); // Use with std::
```

It is possible that a C++ program include a C header as in C. Like:

```
#include <math.h> // Not in std namespace
...
sqrt(5.0); // Use without std::
```

This, however, is not preferred.

- Using .h with C++ header files, like iostream.h, is disastrous. These are deprecated. It is dangerous, yet true, that some compilers do not error out on such use. Exercise caution.**



Program 02.04: Sum n natural numbers

Module 02

Partha Pratim
Das

Objectives & Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

C Program	C++ Program
<pre>// FileName:Sum_n.c: #include <stdio.h> int main() { int n; int i; int sum = 0; printf("Input limit:\n"); scanf("%d", &n); for (i = 0; i <= n; ++i) sum = sum + i; printf("Sum of %d", n); printf(" numbers is: %d\n", sum); return 0; }</pre>	<pre>// FileName:Sum_n_c++.cpp: #include <iostream> using namespace std; int main() { int n; int sum = 0; cout << "Input limit:" << endl; cin >> n; for (int i = 0; i <= n; ++i) // Local Decl. sum = sum + i; cout << "Sum of " << n ; cout << " numbers is: " << sum << endl; return 0; }</pre>
<pre>Input limit: 10 Sum of 10 numbers is: 55</pre>	<pre>Input limit: 10 Sum of 10 numbers is: 55</pre>
<ul style="list-style-type: none">• i must be declared at the beginning (C89)	<ul style="list-style-type: none">• i declared locally in for loop



Program 02.05: Using bool

Module 02

Partha Pratim
Das

Objectives & Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

	C Program	C++ Program
	<pre>// FileName:bool.c: #include <stdio.h> #define TRUE 1 #define FALSE 0 int main() { int x = TRUE; printf ("bool is %d\n", x); return 0; }</pre>	<pre>// FileName:bool.c: #include <stdio.h> #include <stdbool.h> int main() { bool x = true; printf ("bool is %d\n", x); return 0; }</pre>
bool is 1	bool is 1	bool is 1
<ul style="list-style-type: none">Using int and #define for boolMay use _Bool (C99)	<ul style="list-style-type: none">stdbool.h included for bool_Bool type & macros (C99): bool which expands to _Bool true which expands to 1 false which expands to 0	<ul style="list-style-type: none">No additional headers required <p>bool is a built-in type true is a literal false is a literal</p>



Module Summary

Module 02

Partha Pratim
Das

Objectives &
Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool
Summary

- Understanding differences between C and C++ for:
 - IO
 - Variable declaration
 - Standard Library
- C++ gives us more flexibility in terms of basic declaration and input / output
- Many C constructs and functions are simplified in C++ which helps to increase the ease of programming



Instructor and TAs

Module 02

Partha Pratim
Das

Objectives &
Outline

Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

Module 03: Programming in C++

Arrays and Strings

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

- Understand array usage in C and C++
- Understand vector usage in C++
- Understand string functions in C and string type in C++



Module Outline

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

- **Arrays and Vectors**
 - Fixed size arrays – in C and C++
 - Arbitrary size arrays – in C and C++
 - vectors in C++
- **Strings in C and C++**
 - string functions in C and C++
 - string type in C++
 - String manipulation in C++



Program 03.01: Fixed Size Array

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size
Array
Arbitrary Size
Array
Vectors

Strings

Summary

C Program	C++ Program
<pre>// File Name:Array_Fixed_Size.c: #include <stdio.h> int main() { short age[4]; age[0] = 23; age[1] = 34; age[2] = 65; age[3] = 74; printf("%d ", age[0]); printf("%d ", age[1]); printf("%d ", age[2]); printf("%d ", age[3]); return 0; }</pre>	<pre>//FileName:Array_Fixed_Size_c++.cpp: #include <iostream> int main() { short age[4]; age[0] = 23; age[1] = 34; age[2] = 65; age[3] = 74; std::cout << age[0] << " "; std::cout << age[1] << " "; std::cout << age[2] << " "; std::cout << age[3] << " "; return 0; }</pre>
23 34 65 74	23 34 65 74

- No difference between arrays in C and C++



Arbitrary Size Array

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

This can be implemented in C (C++) in the following ways:

- **Case 1:** Declaring a large array with size greater than the size given by users in all (most) of the cases
 - Hard-code the maximum size in code
 - Declare a manifest constant for the maximum size
- **Case 2:** Using `malloc (new[])` to dynamically allocate space at run-time for the array



Program 03.02: Fixed size large array in C

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

Hard-coded

```
// FileName:Array_Large_Size.c:  
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
    int arr[100], sum = 0, i;  
  
    printf("Enter no. of elements: ");  
    int count;  
    scanf("%d", &count);  
  
    for(i = 0; i < count; i++) {  
        arr[i] = i;  
        sum += arr[i];  
    }  
    printf("Array Sum: %d", sum);  
  
    return 0;  
}
```

Enter no. of elements: 10
Array Sum: 45

- Hard-coded size

Using manifest constant

```
// FileName:Array_Macro.c:  
#include <stdio.h>  
#include <stdlib.h>  
#define MAX 100  
  
int main() {  
    int arr[MAX], sum = 0, i;  
  
    printf("Enter no. of elements: ");  
    int count;  
    scanf("%d", &count);  
  
    for(i = 0; i < count; i++) {  
        arr[i] = i;  
        sum += arr[i];  
    }  
    printf("Array Sum: %d", sum);  
  
    return 0;  
}
```

Enter no. of elements: 10
Array Sum: 45

- Size by manifest constant



Program 03.03: Fixed large array / vector

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

C (array & constant)

```
// FileName:Array_Macro.c:  
#include <stdio.h>  
#include <stdlib.h>  
  
#define MAX 100  
  
int main() {  
    int arr[MAX], sum = 0, i;  
  
    printf("Enter no. of elements: ");  
    int count;  
    scanf("%d", &count);  
    for(i = 0; i < count; i++) {  
        arr[i] = i;  
        sum += arr[i];  
    }  
    printf("Array Sum: %d", sum);  
    return 0;  
}
```

Enter no. of elements: 10
Array Sum: 45

- MAX is the declared size of array
- No header needed
- arr declared as int []

C++ (vector & constant)

```
// FileName:Array_Macro_c++.cpp:  
#include <iostream>  
#include <vector>  
using namespace std;  
#define MAX 100  
  
int main() {  
    vector<int> arr(MAX); // Define-time size  
  
    cout << "Enter the no. of elements: ";  
    int count, j, sum = 0;  
    cin >> count;  
    for(int i = 0; i < count; i++) {  
        arr[i] = i;  
        sum += arr[i];  
    }  
    cout << "Array Sum: " << sum << endl;  
    return 0;  
}
```

Enter no. of elements: 10
Array Sum: 45

- MAX is the declared size of vector
- Header vector included
- arr declared as vector<int>



Program 03.04: Dynamically managed array size

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

C Program

```
// FileName:Array_Malloc.c:  
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
    printf("Enter no. of elements ");  
    int count, sum = 0, i;  
    scanf("%d", &count);  
  
    int *arr = (int*) malloc  
        (sizeof(int)*count);  
  
    for(i = 0; i < count; i++) {  
        arr[i] = i;  
        sum += arr[i];  
    }  
    printf("Array Sum:%d ", sum);  
    return 0;  
}
```

Enter no. of elements: 10
Array Sum: 45

C++ Program

```
// FileName:Array_Resize_c++.cpp:  
#include <iostream>  
#include <vector>  
using namespace std;  
  
int main() {  
    cout << "Enter the no. of elements: ";  
    int count, j, sum=0;  
    cin >> count;  
  
    vector<int> arr;      // Default size  
    arr.resize(count);   // Set resize  
  
    for(int i = 0; i < arr.size(); i++) {  
        arr[i] = i;  
        sum += arr[i];  
    }  
    cout << "Array Sum: " << sum << endl;  
    return 0;  
}
```

Enter no. of elements: 10
Array Sum: 45

- malloc allocates space using sizeof

- resize fixes vector size at run-time



Strings in C and C++

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

String manipulations in C and C++:

- C-String and `string.h` library
 - C-String is an array of `char` terminated by `NULL`
 - C-String is supported by functions in `string.h` in C standard library
- `string` type in C++ standard library
 - `string` is a type
 - With operators (like `+` for concatenation) behaves like a built-in type



Program 03.05: Concatenation of Strings

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

C Program

```
// FileName:Add_strings.c:  
#include <stdio.h>  
#include <string.h>  
  
int main() {  
    char str1[] =  
        {'H','E','L','L','O',' ','\0'};  
    char str2[] = "WORLD";  
  
    char str[20];  
    strcpy(str, str1);  
    strcat(str, str2);  
  
    printf("%s\n", str);  
  
    return 0;  
}
```

HELLO WORLD

C++ Program

```
// FileName:Add_strings_c++.cpp:  
#include <iostream>  
#include <string>  
using namespace std;  
  
int main(void) {  
    string str1 = "HELLO ";  
  
    string str2 = "WORLD";  
  
    string str = str1 + str2;  
  
    cout << str;  
  
    return 0;  
}
```

HELLO WORLD

- Need header `string.h`
- C-String is an array of characters
- String concatenation done with `strcat` function
- Need a copy into `str`
- `str` must be large to fit the result

- Need header `string`
- `string` is a data-type in C++ standard library
- Strings are concatenated like addition of int



More on Strings

Module 03

Partha Pratim

Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

Further,

- `operator=` can be used on strings in place of `strcpy` function in C.
- `operator<=`, `operator<`, `operator>=`, `operator>` operators can be used on strings in place of `strcmp` function in C



Module Summary

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

- Working with variable sized arrays is more flexible with vectors in C++
- String operations are easier with C++ standard library



Instructor and TAs

Module 03

Partha Pratim
Das

Objectives &
Outline

Arrays &
Vectors

Fixed Size Array
Arbitrary Size
Array
Vectors

Strings

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

Module 04: Programming in C++

Sorting and Searching

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

- Implementation of Sorting and Searching in C and C++



Module Outline

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

- Sorting in C and C++
 - Bubble Sort
 - Using Standard Library
- Searching in C and C++
 - Using Standard Library
- algorithm Library



Program 04.01: Bubble Sort

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

C Program

```
// FileName:Bubble_Sort.c:  
#include <stdio.h>  
  
int main() {  
    int data[] = {32, 71, 12, 45, 26};  
    int i, step, n = 5, temp;  
  
    for(step = 0; step < n - 1; ++step)  
        for(i = 0; i < n-step-1; ++i) {  
            if(data[i] > data[i+1]) {  
                temp = data[i];  
                data[i] = data[i+1];  
                data[i+1] = temp;  
            }  
        }  
  
    for(i = 0; i < n; ++i)  
        printf("%d ", data[i]);  
  
    return 0;  
}
```

12 26 32 45 71

C++ Program

```
// FileName:Bubble_Sort.cpp:  
#include <iostream>  
using namespace std;  
int main() {  
    int data[] = {32, 71, 12, 45, 26};  
    int n = 5, temp;  
  
    for(int step = 0; step < n - 1; ++step)  
        for(int i = 0; i < n-step-1; ++i) {  
            if (data[i] > data[i+1]) {  
                temp = data[i];  
                data[i] = data[i+1];  
                data[i+1] = temp;  
            }  
        }  
  
    for(int i = 0; i < n; ++i)  
        cout << data[i] << " ";  
  
    return 0;  
}
```

12 26 32 45 71

- Implementation is same in both C and C++ apart from the changes in basic header files, I/O functions explained in Module 02.



Program 04.02: Using sort from standard library

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

C Program (Desc order)

```
// FileName:qsort.c:  
#include <stdio.h>  
#include <stdlib.h>  
  
// compare Function Pointer  
int compare(const void *a, const void *b) {  
    return (*(int*)a < *(int*)b);  
}  
  
int main () {  
    int data[] = {32, 71, 12, 45, 26};  
  
    // Start ptr, # elements, size, func. ptr  
    qsort(data, 5, sizeof(int), compare);  
  
    for(int i = 0; i < 5; i++)  
        printf ("%d ", data[i]);  
  
    return 0;  
}
```

71 45 32 26 12

- sizeof int, array passed in qsort

C++ Program (Desc order)

```
// FileName:Algorithm_Cust_c++.cpp:  
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
// compare Function Pointer  
bool compare (int i, int j) {  
    return (i > j);  
}  
  
int main() {  
    int data[] = {32, 71, 12, 45, 26};  
  
    // Start ptr, end ptr, func. ptr  
    sort (data, data+5, compare);  
  
    for (int i = 0; i < 5; i++)  
        cout << data[i] << " ";  
  
    return 0;  
}
```

71 45 32 26 12

- Size need not be passed.



Program 04.03: Using default sort of algorithm

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting

Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

C++ Program

```
// FileName:Algorithm_Cust_c++.cpp:  
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
int main () {  
    int data[] = {32, 71, 12, 45, 26};  
  
    sort (data, data+5);  
  
    for (int i = 0; i < 5; i++)  
        cout << data[i] << " ";  
  
    return 0;  
}
```

12 26 32 45 71

- Sort using the default sort function of algorithm library which does the sorting in ascending order only.



Program 04.04: Binary Search

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

C Program

```
// FileName:Binary_Search.c:  
#include <stdio.h>  
#include <stdlib.h>  
  
// compare Function Pointer  
int compare (const void * a, const void * b) {  
    if ( *(int*)a < *(int*)b ) return -1;  
    if ( *(int*)a == *(int*)b ) return 0;  
    if ( *(int*)a > *(int*)b ) return 1;  
}  
  
int main () {  
    int data[] = {1, 2, 3, 4, 5};  
    int key = 3;  
  
    if (bsearch (&key, data, 5,  
                sizeof(int), compare))  
        cout << "found!\n";  
    else  
        cout << "not found.\n";  
  
    return 0;  
}
```

found!

C++ Program

```
// FileName:Binary_Search_c++.cpp:  
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
int main() {  
    int data[] = {1, 2, 3, 4, 5};  
    int key = 3;  
  
    if (binary_search (data, data+5, key))  
        cout << "found!\n";  
    else  
        cout << "not found.\n";  
  
    return 0;  
}
```

found!



The algorithm Library

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

The algorithm library of c++ helps us to easily implement commonly used complex functions. We discussed the functions for sort and search. Let us look at some more useful functions.

- Replace element in an array
- Rotates the order of the elements



Program 04.05: replace and rotate functions

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting

Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

Replace

```
// FileName:Replace.cpp:  
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
int main() {  
    int data[] = {1, 2, 3, 4, 5};  
  
    replace (data, data+5, 3, 2);  
  
    for(int i = 0; i < 5; ++i)  
        cout << data[i] << " ";  
  
    return 0;  
}
```

1 2 2 4 5

- 3rd element replaced with 2

Rotate

```
// FileName:Rotate.cpp:  
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
int main() {  
    int data[] = {1, 2, 3, 4, 5};  
  
    rotate (data, data+2, data+5);  
  
    for(int i = 0; i < 5; ++i)  
        cout << data[i] << " ";  
  
    return 0;  
}
```

3 4 5 1 2

- Array circular shifted around 3rd element.



Module Summary

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

- Flexibility of defining *customised* sort algorithms to be passed as parameter to sort and search functions defined in the `algorithm` library.
- Predefined optimised versions of these sort and search functions can also be used.
- There are a number of useful functions like rotate, replace, merge, swap, remove etc in `algorithm` library.



Instructor and TAs

Module 04

Partha Pratim
Das

Objectives &
Outline

Sorting
Bubble Sort
Standard Library

Searching
Standard Library

STL:
algorithm

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C

Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

Module 05: Programming in C++

Stack and its Applications

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C

Reverse a String
Eval Postfix

Stack in C++

Reverse a String
Eval Postfix

Summary

- Understanding implementation and use of stack in C
- Understanding stack in C++ standard library and its use



Module Outline

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C

Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

- Stack in C
 - Reverse a String
 - Evaluate a Postfix Expression
- Stack in C++
 - Reverse a String
 - Evaluate a Postfix Expression



Understanding Stack in C

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C

Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

- Stack is a LIFO (last-In-First-Out) container that can maintain a collection of arbitrary number of data items – all of the same type
- To create a stack in C we need to:
 - Decide on the data type of the elements
 - Define a structure (container) (with maximum size) for stack and declare a top variable in the structure
 - Write separate functions for push, pop, top, and isempty using the declared structure
- Note:
 - Change of the data type of elements, implies re-implementation for all the stack codes
 - Change in the structure needs changes in all functions
- Unlike sin, sqrt etc. function from C standard library, we do not have a ready-made stack that we can use



Common C programs using stack

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C
Reverse a String
Eval Postfix

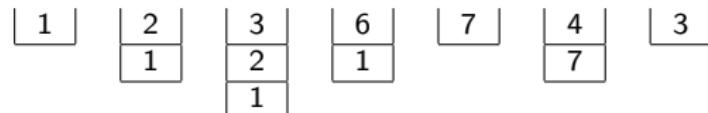
Stack in C++
Reverse a String
Eval Postfix

Summary

Some common C programs that use stack:

- Reversing a string
 - Input: ABCDE
 - Output: EDCBA
- Evaluation of postfix expression
 - Input: 1 2 3 * + 4 - (for $1 + 2 * 3 - 4$)
 - Output: 3

Stack states:



- Identification of palindromes (w/ and w/o center-marker)
- Conversion of an infix expression to postfix
- Depth-first Search (DFS)



Program 05.01: Reversing a string

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C
Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

```
// FileName: Reverse_String.c
#include <stdio.h>

typedef struct stack {
    char data [100];
    int top;
} stack;

int empty (stack *p) {
    return (p->top == -1);
}

int top (stack *p) {
    return p -> data [p->top];
}

void push (stack *p, char x) {
    p -> data [++(p -> top)] = x;
}

void pop (stack *p) {
    if (!empty(p)) {
        (p->top) = (p->top) - 1;
    }
}

void main() {
    stack s;
    s.top = -1;

    char ch, str[10] = "ABCDE";
    int i, len = sizeof(str);

    for(i = 0; i < len; i++) {
        push(&s, str[i]);
    }

    printf ("Reversed String: ");

    while (!empty(&s)){
        printf("%c ", top(&s));
        pop(&s);
    }
}
```

Reversed String: EDCBA



Program 05.02: Postfix Expression Evaluation

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C
Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

```
// FileName: PostFix_Evaluation.c
#include<stdio.h>

typedef struct stack {
    char data [100];
    int top;
} stack;

int empty (stack *p) {
    return (p->top == -1);
}

int top (stack *p) {
    return p -> data [p->top];
}

void push (stack *p, char x) {
    p -> data [++(p -> top)] = x;
}

void pop (stack *p) {
    if (!empty(p)) {
        (p->top) = (p->top) - 1;
    }
}

void main() {
    stack s;
    s.top = -1;

    // Postfix expression: 1 2 3 * + 4 -
    char postfix[] = {'1','2','3','*','+', '4', '-'};

    int i, op1, op2;

    for(i = 0; i < 7; i++) {
        char ch = postfix[i];
        if (isdigit(ch)) push(&s, ch-'0');
        else {
            op2 = top(&s); pop(&s);
            op1 = top(&s); pop(&s);
            switch (ch) {
                case '+':push(&s, op1 + op2);break;
                case '-':push(&s, op1 - op2);break;
                case '*':push(&s, op1 * op2);break;
                case '/':push(&s, op1 / op2);break;
            }
        }
    }
    printf("Evaluation %d\n", top(&s));
}
```

Evaluation 3



Understanding Stack in C++

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C
Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

- C++ standard library provide a ready-made stack for any type of elements
- To create a stack in C++ we need to:
 - Include the stack header
 - Instantiate a stack with proper element type (like `char`)
 - Use the functions of the stack objects for stack operations



Program 05.03: Reverse a String in C++

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C
Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

```
// FileName: Reverse_String_c++.cpp
#include<iostream>
#include<string.h>
#include<stack>
using namespace std;

int main() {
    char str[10] = "ABCDE";
    stack<char> s;
    int i;

    for(i = 0; i < strlen(str); i++)
        s.push(str[i]);

    cout << "Reversed String: ";

    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }

    return 0;
}
```

- No codes for creating stack
- No initialization
- Clean interface for stack functions
- Available in library – well-tested

```
// FileName: Reverse_String.c
int main() {
    char str[10] = "ABCDE";
    stack s; s.top = -1;
    int i;

    for(i = 0; i < strlen(str); i++)
        push(&s, str[i]);

    printf ("Reversed String: ");

    while (!empty(&s)){
        printf("%c ", top(&s));
        pop(&s);
    }

    return 0;
}
```

- Lot of code for creating stack
- top to be initialized
- Cluttered interface for stack functions
- Implemented by user – error-prone



Program 05.04: Postfix Evaluation in C++

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C

Reverse a String
Eval Postfix

Stack in C++

Reverse a String
Eval Postfix

Summary

```
// FileName:Postfix_Evaluation_c++.cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    // Postfix expression: 1 2 3 * + 4 -
    char postfix[] = {'1','2','3','*','+', '4', '-'}, ch;
    stack<int> s;

    for(int i = 0; i < 7; i++) {
        ch = postfix[i];
        if (isdigit(ch)) { s.push(ch-'0'); }
        else {
            int op1 = s.top(); s.pop();
            int op2 = s.top(); s.pop();
            switch(ch) {
                case '*': s.push(op2 * op1); break;
                case '/': s.push(op2 / op1); break;
                case '+': s.push(op2 + op1); break;
                case '-': s.push(op2 - op1); break;
            }
        }
    }
    cout << "\nEvaluation " << s.top();
    return 0;
}
```



Module Summary

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C
Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

- C++ standard library provides ready-made stack. It works like a data type
- Any type of element can be used for C++ stack
- Similar containers as available in C++ standard library include:
 - queue
 - deque
 - list
 - map
 - set
 - ... and more



Instructor and TAs

Module 05

Partha Pratim
Das

Objectives &
Outline

Stack in C
Reverse a String
Eval Postfix

Stack in C++
Reverse a String
Eval Postfix

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

Module 06: Programming in C++

Constants and Inline Functions

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

- Understand `const` in C++ and contrast with Manifest Constants
- Understand `inline` in C++ and contrast with Macros



Module Outline

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness

Advantages

Pointers

volatile

inline
functions

Macros

inline

Summary

- const-ness and cv-qualifier
 - Notion of const
 - Advantages of const
 - Natural Constants – π , e
 - Program Constants – array size
 - Prefer const to #define
 - const and pointer
 - const-ness of pointer / pointee. How to decide?
 - Notion of volatile
- inline functions
 - Macros with params
 - Advantages
 - Disadvantages
 - Notion of inline functions
 - Advantages



Program 06.01: Manifest constants in C

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions

Macros
inline

Summary

- Manifest constants are defined by `#define`
- Manifest constants are replaced by CPP (C Pre-Processor)

Source Program	Program after CPP
<pre>#include <iostream> #include <cmath> using namespace std; #define TWO 2 #define PI 4.0*atan(1.0) int main() { int r = 10; double peri = TWO * PI * r; cout << "Perimeter = " << peri << endl; return 0; }</pre>	<pre>// Contents of <iostream> header replaced by CPP // Contents of <cmath> header replaced by CPP using namespace std; // #define of TWO consumed by CPP // #define of PI consumed by CPP int main() { int r = 10; double peri = 2 * 4.0*atan(1.0) * r; // Replaced by CPP cout << "Perimeter = " << peri << endl; return 0; }</pre>
Perimeter = 314.159	Perimeter = 314.159
<ul style="list-style-type: none">TWO is a manifest constantPI is a manifest constantTWO & PI look like variables	<ul style="list-style-type: none">CPP replaces the token TWO by 2CPP replaces the token PI by 4.0*atan(1.0)Compiler sees them as constants



Notion of const-ness

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness

Advantages

Pointers

volatile

inline
functions

Macros
inline

Summary

- The value of a const variable cannot be changed after definition

```
const int n = 10; // n is an int type variable with value 10
                  // n is a constant
...
n = 5; // Is a compilation error as n cannot be changed
...
int m;
int *p = 0;
p = &m; // Hold m by pointer p
*p = 7; // Change m by p; m is now 7
...
p = &n; // Is a compilation error as n may be changed by *p = 5;
```

- Naturally, a const variable *must be initialized when defined*

```
const int n; // Is a compilation error as n must be initialized
```

- A variable of any data type can be declared as const

```
typedef struct _Complex {
    double re;
    double im;
} Complex;
const Complex c = {2.3, 7.5}; // c is a Complex type variable
                           // It is initialized with c.re = 2.3 and c.im = 7.5
                           // c is a constant
...
c.re = 3.5; // Is a compilation error as no part of c can be changed
```



Program 06.02: Compare #define and const

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness

Advantages

Pointers

volatile

inline
functions

Macros
inline

Summary

	Using #define	Using const
	<pre>#include <iostream> #include <cmath> using namespace std; #define TWO 2 #define PI 4.0*atan(1.0) int main() { int r = 10; double peri = TWO * PI * r; cout << "Perimeter = " << peri << endl; return 0; }</pre>	<pre>#include <iostream> #include <cmath> using namespace std; const int TWO = 2; const double PI = 4.0*atan(1.0); int main() { int r = 10; double peri = TWO * PI * r; // No replacement by CPP cout << "Perimeter = " << peri << endl; return 0; }</pre>
	Perimeter = 314.159	Perimeter = 314.159
	<ul style="list-style-type: none">• TWO is a manifest constant• PI is a manifest constant• TWO & PI look like variables• Types of TWO & PI may be indeterminate	<ul style="list-style-type: none">• TWO is a const variable initialized to 2• PI is a const variable initialized to 4.0*atan(1.0)• TWO & PI are variables• Type of TWO is const int• Type of PI is const double



Advantages of const

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &

cv-qualifier

const-ness

Advantages

Pointers

volatile

inline

functions

Macros

inline

Summary

- Natural Constants like π , e , Φ (*Golden Ratio*) etc. can be compactly defined and used

```
const double pi = 4.0*atan(1.0);           // pi = 3.14159
const double e = exp(1.0);                 // e = 2.71828
const double phi = (sqrt(5.0) + 1) / 2.0; // phi = 1.61803

const int TRUE = 1;                       // Truth values
const int FALSE = 0;

const int null = 0;                      // null value
```

Note: NULL is a manifest constant in C/C++ set to 0.

- Program Constants like number of elements, array size etc. can be defined at one place (at times in a header) and used all over the program

```
const int nArraySize = 100;
const int nElements = 10;

int main() {
    int A[nArraySize];                  // Array size
    for (int i = 0; i < nElements; ++i) // Number of elements
        A[i] = i * i;

    return 0;
}
```



Advantages of const

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages

Pointers
volatile

inline
functions

Macros
inline

Summary

● Prefer const over #define

Using #define	Using const
Manifest Constant	Constant Variable
<ul style="list-style-type: none">● Is not type safe● Replaced textually by CPP● Cannot be <i>watched</i> in debugger● Evaluated as many times as replaced	<ul style="list-style-type: none">● Has its type● Visible to the compiler● Can be <i>watched</i> in debugger● Evaluated only on initialization



const and Pointers

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &

cv-qualifier

const-ness

Advantages

Pointers

volatile

inline

functions

Macros

inline

Summary

- const-ness can be used with Pointers in one of the two ways:
 - **Pointer to Constant data** where the pointee (pointed data) cannot be changed
 - **Constant Pointer** where the pointer (address) cannot be changed
- Consider usual pointer-pointee computation (without const):

```
int m = 4;
int n = 5;
int * p = &n; // p points to n. *p is 5
...
n = 6;          // n and *p are 6 now
*p = 7;         // n and *p are 7 now. POINTEE changes
...
p = &m;          // p points to m. *p is 4. POINTER changes
*p = 8;         // m and *p are 8 now. n is 7. POINTEE changes
```



const and Pointers: *Pointer to Constant data*

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions

Macros
inline

Summary

Consider pointed data

```
int m = 4;
const int n = 5;
const int * p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
p = &m; // Okay
*p = 8; // Okay
```

Interestingly,

```
int n = 5;
const int * p = &n;
...
n = 6; // Okay
*p = 6; // Error: p points to a 'constant' data (n) that cannot be changed
```

Finally,

```
const int n = 5;
int * p = &n; // Error: If this were allowed, we would be able to change constant n
...
n = 6; // Error: n is constant and cannot be changed
*p = 6; // Would have been okay, if declaration of p were valid
```



const and Pointers: *Constant Pointer*

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier
const-ness
Advantages

Pointers

volatile

inline
functions

Macros
inline

Summary

Consider pointer

```
int m = 4, n = 5;
int * const p = &n;
...
n = 6; // Okay
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

By extension, both can be const

```
const int m = 4;
const int n = 5;
const int * const p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a 'constant' data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through *

```
int n = 5;
int * p = &n;           // non-const-Pointer to non-const-Pointee
const int * p = &n;     // non-const-Pointer to const-Pointee
int * const p = &n;     // const-Pointer to non-const-Pointee
const int * const p = &n; // const-Pointer to const-Pointee
```



const and Pointers: The case of C-string

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier
const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

Consider the example:

```
char * str = strdup("IIT, Kharagpur");
str[0] = 'N';                                // Edit the name
cout << str << endl;
str = strdup("JIT, Kharagpur"); // Change the name
cout << str << endl;
```

Output is:

```
NIT, Kharagpur
JIT, Kharagpur
```

To stop editing the name:

```
const char * str = strdup("IIT, Kharagpur");
str[0] = 'N';                                // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Change the name
```

To stop changing the name:

```
char * const str = strdup("IIT, Kharagpur");
str[0] = 'N';                                // Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

To stop both:

```
const char * const str = strdup("IIT, Kharagpur");
str[0] = 'N';                                // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```



Notion of volatile

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

● Variable Read-Write

- The value of a variable can be read and / or assigned at any point of time
- The value assigned to a variable does not change till a next assignment is made (value is persistent)

● const

- The value of a const variable can be set only at initialization – cannot be changed afterwards

● volatile

- *In contrast*, the value of a volatile variable may be different every time it is read – even if no assignment has been made to it
- A variable is taken as volatile if it can be changed by hardware, the kernel, another thread etc.
- cv-qualifier: A declaration may be prefixed with a qualifier – const or volatile



Using volatile

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

```
static int i;
void fun(void) {
    i = 0;
    while (i != 100);
}
```

This is an infinite loop! Hence the compiler should optimize as:

```
static int i;
void fun(void) {
    i = 0;
    while (1);           // Compiler optimizes
}
```

Now qualify i as volatile:

```
static volatile int i;
void fun(void) {
    i = 0;
    while (i != 100);   // Compiler does not optimize
}
```

Being volatile, i can be changed by hardware anytime. It waits till the value becomes 100 (possibly some hardware writes to a port).



Program 06.03: Macros with Parameters

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions

Macros
inline

Summary

- Macros with Parameters are defined by `#define`
- Macros with Parameters are replaced by CPP

Source Program	Program after CPP
<pre>#include <iostream> using namespace std; #define SQUARE(x) x * x int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>	<pre>// Contents of <iostream> header replaced by CPP using namespace std; // #define of SQUARE(x) consumed by CPP int main() { int a = 3, b; b = a * a; // Replaced by CPP cout << "Square = " << b << endl; return 0; }</pre>
Square = 9	Square = 9
<ul style="list-style-type: none">SQUARE(x) is a macro with one paramSQUARE(x) looks like a function	<ul style="list-style-type: none">CPP replaces the SQUARE(x) substituting x with aCompiler does not see it as function



Pitfalls of macros

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier
const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

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

#define SQUARE(x) x * x

int main() {
    int a = 3, b;

    b = SQUARE(a + 1); // Wrong macro expansion

    cout << "Square = " << b << endl;

    return 0;
}
```

Output is 7 in stead of 16 as expected. On the expansion line it gets:

```
b = a + 1 * a + 1;
```

To fix:

```
#define SQUARE(x) (x) * (x)
```

Now:

```
b = (a + 1) * (a + 1);
```



Pitfalls of macros

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness

Advantages

Pointers

volatile

inline
functions

Macros
inline

Summary

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

#define SQUARE(x) (x) * (x)

int main() {
    int a = 3, b;

    b = SQUARE(++a);

    cout << "Square = " << b << endl;

    return 0;
}
```

Output is 25 in stead of 16 as expected. On the expansion line it gets:

```
b = (++a) * (++a);
```

and a is incremented twice before being used! There is no easy fix.



inline Function

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

- An **inline** function is just another functions
- The function prototype is preceded by the keyword **inline**
- An **inline** function is expanded (inlined) at the site of its call and the overhead of passing parameters between caller and callee (or called) functions is avoided



Program 06.04: Macros as inline Functions

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages

Pointers

volatile

inline
functions

Macros
inline

Summary

- Define the function
- Prefix function header with `inline`
- Compile function body and function call together

Using macro	Using inline
<pre>#include <iostream> using namespace std; #define SQUARE(x) x * x int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; inline int SQUARE(int x) { return x * x; } int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>
Square = 9	Square = 9
<ul style="list-style-type: none"><code>SQUARE(x)</code> is a macro with one paramMacro <code>SQUARE(x)</code> is efficient<code>SQUARE(a + 1)</code> fails<code>SQUARE(++a)</code> fails<code>SQUARE(++a)</code> does not check type	<ul style="list-style-type: none"><code>SQUARE(x)</code> is a function with one param<code>inline SQUARE(x)</code> is equally efficient<code>SQUARE(a + 1)</code> works<code>SQUARE(++a)</code> works<code>SQUARE(++a)</code> checks type



Macros & inline Functions: Compare and Contrast

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions

Macros
inline

Summary

Macros

- Expanded at the place of calls
- Efficient in execution
- Code bloats
- Has syntactic and semantic pitfalls
- Type checking for parameters is not done
- Helps to write `max` / `swap` for all types
- Errors are not checked during compilation
- Not available to debugger

inline Functions

- Expanded at the place of calls
- Efficient in execution
- Code bloats
- No pitfall
- Type checking for parameters is robust
- Needs template for the same purpose
- Errors are checked during compilation
- Available to debugger in DEBUG build



Limitations of Function inlineing

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness

Advantages

Pointers

volatile

inline
functions

Macros

inline

Summary

- **inlineing** is a directive – compiler may not inline functions with large body
- **inline** functions may not be recursive
- Function body is needed for **inlineing** at the time of function call. Hence, implementation hiding is not possible. *Implement inline functions in header files*
- **inline** functions must not have two different definitions



Module Summary

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

- Revisit manifest constants from C
- Understand const-ness, its use and advantages over manifest constants
- Understand the interplay of const and pointer
- Understand the notion and use of volatile data
- Revisit macros with parameters from C
- Understand inline functions and their advantages over macros
- Limitations of inlineing



Instructor and TAs

Module 06

Partha Pratim
Das

Objectives &
Outline

const-ness &
cv-qualifier

const-ness
Advantages
Pointers
volatile

inline
functions
Macros
inline

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

Module 07: Programming in C++

Reference & Pointer

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

- Understand References in C++
- Compare and contrast References and Pointers



Module Outline

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

- Reference variable or Alias
 - Basic Notion
 - Call-by-reference in C++
- Example: Swapping two number in C
 - Using Call-by-value
 - Using Call-by-address
- Call-by-reference in C++ in contrast to Call-by-value in C
- Use of const in Alias / Reference
- Return-by-reference in C++ in contrast to Return-by-value in C
- Differences between References and Pointers



Reference

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

- A reference is an alias / synonym for an existing variable

```
int i = 15; // i is a variable  
int &j = i; // j is a reference to i
```

i	← variable
15	← memory content
200	← address
j	← alias or reference



Program 07.01: Behavior of Reference

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

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

int main() {
    int a = 10, &b = a; // b is reference of a

    // a and b have the same memory
    cout << "a = " << a << ", b = " << b << endl;
    cout << "&a = " << &a << ", &b = " << &b << endl;

    ++a; // Changing a appears as change in b
    cout << "a = " << a << ", b = " << b << endl;

    ++b; // Changing b also changes a
    cout << "a = " << a << ", b = " << b << endl;

    return 0;
}
```

```
a = 10, b = 10
&a = 002BF944, &b = 002BF944
a = 11, b = 11
a = 12, b = 12
```

- a and b have the same memory location and hence the same value
- Changing one changes the other and vice-versa



Pitfalls in Reference

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

Wrong declaration	Reason	Correct declaration
<code>int& i;</code>	no variable to refer to – must be initialized	<code>int& i = j;</code>
<code>int& j = 5;</code>	no address to refer to as 5 is a constant	<code>const int& j = 5;</code>
<code>int& i = j + k;</code>	only temporary address (result of $j + k$) to refer to	<code>const int& i = j + k;</code>



C++ Program 07.02: Call-by-reference

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

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

void Function_under_param_test{// Function prototype
    int &b, // Reference parameter
        c); // Value parameter

int main() {
    int a = 20;
    cout << "a = " << a << ", &a = " << &a << endl << endl;
    Function_under_param_test(a, a); // Function call

    return 0;
}

void Function_under_param_test(int &b, int c) { // Function definition
    cout << "b = " << b << ", &b = " << &b << endl << endl;
    cout << "c = " << c << ", &c = " << &c << endl << endl;
}

----- Output -----
a = 20, &a = 0023FA30
b = 20, &b = 0023FA30
c = 20, &c = 0023F95C
```

- Param b is call-by-reference while param c is call-by-value
- Actual param a and formal param b get the same value in called function
- Actual param a and formal param c get the same value in called function
- Actual param a and formal param b get the same value in called function
- However, actual param a and formal param c have *different* addresses in called function



C Program 07.03: Swap in C

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

	Call-by-value	Call-by-address
	<pre>#include <stdio.h> void swap(int, int); // Call-by-value int main() { int a = 10, b = 15; printf("a= %d & b= %d to swap\n", a, b); swap(a, b); printf("a= %d & b= %d on swap\n", a, b); return 0; } void swap(int c, int d){ int t; t = c; c = d; d = t; }</pre>	<pre>#include <stdio.h> void swap(int *, int *); // Call-by-address int main() { int a=10, b=15; printf("a= %d & b= %d to swap\n", a, b); swap(&a, &b); printf("a= %d & b= %d on swap\n", a, b); return 0; } void swap(int *x, int *y){ int t; t = *x; *x = *y; *y = t; }</pre>
	<ul style="list-style-type: none">• a= 10 & b= 15 to swap• a= 10 & b= 15 on swap	<ul style="list-style-type: none">• a= 10 & b= 15 to swap• a= 15 & b= 10 on swap
	<ul style="list-style-type: none">• Passing values of a=10 & b=15• In callee; c = 10 & d = 15• Swapping the values of c & d• No change for the values of a & b in caller• Swapping the value of c & d instead of a & b	<ul style="list-style-type: none">• Passing Address of a & b• In callee x = Addr(a) & y = Addr(b)• Values at the addresses is swapped• Changes for the values of a & b in caller• It is correct, but C++ has a better way out



Program 07.04: Swap in C & C++

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

C Program: Call-by-value – wrong

```
#include <stdio.h>

void swap(int, int); // Call-by-value
int main() {
    int a = 10, b = 15;
    printf("a= %d & b= %d to swap\n",a,b);
    swap(a, b);
    printf("a= %d & b= %d on swap\n",a,b);
    return 0;
}

void swap(int c, int d) {
    int t ;
    t = c ;
    c = d ;
    d = t ;
}
```

- a= 10 & b= 15 to swap
- a= 10 & b= 15 on swap

- Passing values of a=10 & b=15
- In callee; c=10 & d=15
- Swapping the values of c & d
- No change for the values of a & b
- Here c & d do not share address with a & b

C++ Program: Call-by-reference – right

```
#include <iostream>
using namespace std;
void swap(int&, int&); // Call-by-reference
int main() {
    int a = 10, b = 15;
    cout<<"a= "<<a<<" & b= "<<b<<"to swap"\n";
    swap(a, b);
    cout<<"a= "<<a<<" & b= "<<b<<"on swap"\n";
    return 0;
}

void swap(int &x, int &y) {
    int t ;
    t = x ;
    x = y ;
    y = t ;
}
```

- a= 10 & b= 15 to swap
- a= 15 & b= 10 on swap

- Passing values of a = 10 & b = 15
- In callee x = 10 & y = 15
- Swapping the value x & y
- Changes the values of a & b
- x & y having same address as a & b respectively



Program 07.05: Reference Parameter as const

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

- A reference parameter may get changed in the called function
- Use `const` to stop reference parameter being changed

const reference – bad	const reference – good
<pre>#include <iostream> using namespace std; int Ref_const(const int &x) { ++x; // Not allowed return (x); } int main() { int a = 10, b; b = Ref_const(a); cout << "a = " << a << " and" << " b = " << b; return 0; }</pre>	<pre>#include <iostream> using namespace std; int Ref_const(const int &x) { return (x + 1); } int main() { int a = 10, b; b = Ref_const(a); cout << "a = " << a << " and" << " b = " << b; return 0; }</pre>
<ul style="list-style-type: none">• Error: Increment of read only Reference 'x'• Compilation Error: Value of x can't be changed• Implies, 'a' can't be changed through 'x'	a = 10 and b = 11 <ul style="list-style-type: none">• No violation.



Program 07.06: Return-by-reference

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

- A function can return a value by reference
- C uses Return-by-value

Return-by-value	Return-by-reference
<pre>#include <iostream> using namespace std; int Function_Return_By_Val(int &x) { cout << "x = "<<x<<" &x = "<<&x<<endl; return (x); } int main() { int a = 10; cout << "a = "<<a<<" &a = "<<&a<<endl; const int& b = // const needed. Why? Function_Return_By_Val(a); cout << "b = "<<b<<" &b = "<<&b<<endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; int& Function_Return_By_Ref(int &x) { cout << "x = "<<x<<" &x = "<<&x<<endl; return (x); } int main() { int a = 10; cout << "a = "<<a<<" &a = "<<&a<<endl; const int& b = // const optional Function_Return_By_Ref(a); cout << "b = "<<b<<" &b = "<<&b<<endl; return 0; }</pre>
<pre>a = 10 &a = 00DCFD18 x = 10 &x = 00DCFD18 b = 10 &b = 00DCFD00</pre> <ul style="list-style-type: none"> • Returned variable is temporary • Has a different address 	<pre>a = 10 &a = 00A7F8FC x = 10 &x = 00A7F8FC b = 10 &b = 00A7F8FC</pre> <ul style="list-style-type: none"> • Returned variable is an alias of a • Has the same address



Program 07.07: Return-by-reference can get tricky

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

Return-by-reference

```
#include <iostream>
using namespace std;
int& Return_ref(int &x) {

    return (x);
}

int main() {
    int a = 10, b;
    b = Return_ref(a);
    cout << "a = " << a << " and b = "
        << b << endl;

    Return_ref(a) = 3; // Changes
                       // reference
    cout << "a = " << a;

    return 0;
}
```

a = 10 and b = 10
a = 3

- Note how a value is assigned to function call
- This can change a local variable

Return-by-reference – Risky!

```
#include <iostream>
using namespace std;
int& Return_ref(int &x) {
    int t = x;
    t++;
    return (t);
}

int main() {
    int a = 10, b;
    b = Return_ref(a);
    cout << "a = " << a << " and b = "
        << b << endl;

    Return_ref(a) = 3;

    cout << "a = " << a;

    return 0;
}
```

a = 10 and b = 11
a = 10

- We expect a to be 3, but it has not changed
- It returns reference to local. This is risky



I/O of a Function

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

- In C++ we can change values with a function as follows:

Orifice	Purpose	Mechanism
Value Parameter	Input	Call-by-value
Reference Parameter	In-Out	Call-by-reference
const Reference Parameter	Input	Call-by-reference
Return Value	Output	Return-by-value Return-by-reference



Recommended Mechanisms

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

● Call

- Pass parameters of built-in types by value
 - Recall: Array parameters are passed by reference in C
- Pass parameters of user-defined types by reference
 - Make a reference parameter `const` if it is not used for output

● Return

- Return built-in types by value
- Return user-defined types by reference
 - Return value is not copied back
 - May be faster than returning a value
 - Beware: Calling function can change returned object
 - Never return a local variables by reference



Difference between Reference and Pointer

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

Pointers	References
<ul style="list-style-type: none">Refers to an addressPointers can point to NULL. <pre>int *p = NULL; // p is not pointing</pre> <ul style="list-style-type: none">Pointers can point to different variables at different times <pre>int a, b, *p; p = &a; // p points to a ... p = &b // p points to b</pre>	<ul style="list-style-type: none">Refers to an addressReferences cannot be NULL <pre>int &j; //wrong</pre> <ul style="list-style-type: none">For a reference, its referent is fixed <pre>int a, c, &b = a; // Okay &b = c // Error</pre>
<ul style="list-style-type: none">NULL checking is required	<ul style="list-style-type: none">Makes code fasterDoes not require NULL checking
<ul style="list-style-type: none">Allows users to operate on the address – diff pointers, increment, etc.	<ul style="list-style-type: none">Does not allow users to operate on the address. All operations are interpreted for the referent
<ul style="list-style-type: none">Array of pointers can be defined	<ul style="list-style-type: none">Array of references not allowed



Module Summary

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference
Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

- Introduced reference in C++
- Studied the difference between call-by-value and call-by-reference
- Studied the difference between return-by-value and return-by-reference
- Discussed the difference between References and Pointers



Instructor and TAs

Module 07

Partha Pratim
Das

Objectives &
Outlines

Reference
variable

Call-by-
reference

Swap in C
Swap in C++
const Reference
Parameter

Return-by-
reference

I/O of a
Function

References vs.
Pointers

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

Module 08: Programming C++

Default Parameters & Function Overloading

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Understand default parameters
- Understand function overloading and Resolution



Module Outline

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Default parameter
 - Motivation
 - Call function with default parameter
 - Highlighted Points
 - Restrictions
- Function overloading
 - Meaning & Motivation
 - Necessity of function overloading in Contrast with C
- Static Polymorphism
 - Meaning
 - Overloading function
- Overload Resolution
- Default parameters and Function Overloading



Motivation: Example CreateWindow in MSDN

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

Declaration of CreateWindow	Calling CreateWindow
<pre>HWND WINAPI CreateWindow(_In_opt_ LPCTSTR lpClassName, _In_opt_ LPCTSTR lpWindowName, _In_ DWORD dwStyle, _In_ int x, _In_ int y, _In_ int nWidth, _In_ int nHeight, _In_opt_ HWND hWndParent, _In_opt_ HMENU hMenu, _In_opt_ HINSTANCE hInstance, _In_opt_ LPVOID lpParam);</pre>	<pre>hWnd = CreateWindow(ClsName, WndName, WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);</pre>

- There are 11 Number of parameters in `CreateWindow()`
- Out of these 11, 7 parameters (4 are `CWUSEDEFAULT` and 3 are `NULL`) usually get fixed values in a call
- Instead of using these 7 fixed valued Parameters at the time of calling, we could have avoided those by assigning those value much earlier in function formal parameter
- C++ allows us to do so through the mechanism called **Default parameters**



Program 08.01: Function with a default parameter

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

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

int IdentityFunction(int a = 10) { // Default value for the parameter
    return (a);
}

int main() {
    int x = 5, y;

    y = IdentityFunction(x); // Usual function call
    cout << "y = " << y << endl;

    y = IdentityFunction(); // Uses default parameter
    cout << "y = " << y << endl;
}

-----
y = 5
y = 10
```



Program 08.02: Function with 2 default parameters

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

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

int Add(int a = 10, int b = 20) {
    return (a + b);
}

int main(){
    int x = 5, y = 6, z;

    z = Add(x, y); // Usual function call -- a = x = 5 & b = y = 6
    cout << "Sum = " << z << endl;

    z = Add(x);      // One parameter defaulted -- a = x = 5 & b = 20
    cout << "Sum = " << z << endl;

    z = Add();        // Both parameter defaulted -- a = 10 & b = 20
    cout << "Sum = " << z << endl;
}

-----
Sum = 11
Sum = 25
Sum = 30
```



Default Parameter: Highlighted Points

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- C++ allows programmer to assign default values to the function parameters
- Default values are specified while prototyping the function
- Default parameters are required while calling functions with fewer arguments or without any argument
- Better to use default value for less used parameters
- Default arguments may be expressions also



Restrictions on default parameters

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- All parameters to the right of a parameter with default argument must have default arguments (function f)
- Default arguments cannot be re-defined (function g)
- All non-defaulted parameters needed in a call (call g())

```
#include <iostream>

void f(int, double = 0.0, char *);  
// Error C2548: 'f': missing default parameter for parameter 3

void g(int, double = 0, char * = NULL); // OK  
void g(int, double = 1, char * = NULL);  
// Error C2572: 'g': redefinition of default parameter : parameter 3  
// Error C2572: 'g': redefinition of default parameter : parameter 2

int main() {  
    int i = 5; double d = 1.2; char c = 'b';  
  
    g(); // Error C2660: 'g': function does not take 0 arguments  
    g(i);  
    g(i, d);  
    g(i, d, &c);  
    return 0;  
}
```



Restrictions on default parameters

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Default parameters should be supplied only in a header file and not in the definition of a function

```
// Header file: myFunc.h
void g(int, double, char = 'a');

-----
// Source File: myFunc.cpp
#include <iostream>
using namespace std;
#include "myFunc.h"

void g(int i, double d, char c) {
    cout << i << ' ' << d << ' ' << c << endl;
}

-----
// Application File: Apps.cpp
#include <iostream>
#include "myFunc.h"
// void g(int, double, char = 'a');

void g(int i, double f = 0.0, char ch); // OK a new overload
void g(int i = 0, double f, char ch); // OK a new overload
int main() {
    int i = 5; double d = 1.2; char c = 'b';

    g();           // Prints: 0 0 a
    g(i);         // Prints: 5 0 a
    g(i, d);      // Prints: 5 1.2 a
    g(i, d, c);   // Prints: 5 1.2 b
    return 0;
}
```



Function overloads: Matrix Multiplication in C

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Similar functions with different data types & algorithms

```
typedef struct { int data[10][10]; } Mat;      // 2D Matrix
typedef struct { int data[1][10]; } VecRow; // Row Vector
typedef struct { int data[10][1]; } VecCol; // Column Vector

void Multiply_M_M (Mat a,      Mat b,      Mat* c)    { /* c = a * b */ }
void Multiply_M_VC (Mat a,      VecCol b,    VecCol* c) { /* c = a * b */ }
void Multiply_VR_M (VecRow a,   Mat b,      VecRow* c) { /* c = a * b */ }
void Multiply_VC_VR(VecCol a,   VecRow b,    Mat* c)  { /* c = a * b */ }
void Multiply_VR_VC(VecRow a,   VecCol b,   int* c)   { /* c = a * b */ }

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply_M_M (m1, m2, &rm); // rm <- m1 * m2
    Multiply_M_VC (m1, cv, &rcv); // rcv <- m1 * cv
    Multiply_VR_M (rv, m2, &rrv); // rrv <- rv * m2
    Multiply_VC_VR(cv, rv, &rm); // rm <- cv * rv
    Multiply_VR_VC(rv, cv, &r); // r <- rv * cv
    return 0;
}
```

- 5 multiplication functions share same functionality but different argument types
- C treats them as 5 separate functions
- C++ has an elegant solution



Function overloads: Matrix Multiplication in C++

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Functions having similar functionality but different in details.

```
typedef struct { int data[10][10]; } Mat;      // 2D Matrix
typedef struct { int data[1][10]; } VecRow; // Row Vector
typedef struct { int data[10][1]; } VecCol; // Column Vector

void Multiply(const Mat& a,      const Mat& b,      Mat& c)    { /* c = a * b */ };
void Multiply(const Mat& a,      const VecCol& b,  VecCol& c) { /* c = a * b */ };
void Multiply(const VecRow& a,    const Mat& b,      VecRow& c) { /* c = a * b */ };
void Multiply(const VecCol& a,    const VecRow& b,  Mat& c)  { /* c = a * b */ };
void Multiply(const VecRow& a,    const VecCol& b,  int& c)   { /* c = a * b */ };

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm); // rm <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm); // rm <-- cv * rv
    Multiply(rv, cv, r); // r <-- rv * cv
    return 0;
}
```

- These 5 functions having different argument types are treated as one function (`Multiply`) in C++
- This is called **Function Overloading or Static Polymorphism**



Program 08.03/04: Function Overloading

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Define multiple functions having the same name
- Binding happens at compile time

Same # of Parameters	Different # of Parameters
<pre>#include <iostream> using namespace std; int Add(int a, int b) { return (a + b); } double Add(double c, double d) { return (c + d); } int main() { int x = 5, y = 6, z; z = Add(x, y); // int Add(int, int) cout << "int sum = " << z; double s = 3.5, t = 4.25, u; u = Add(s, t); // double Add(double, double) cout << "double sum = " << u << endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; int Area(int a, int b) { return (a * b); } int Area(int c) { return (c * c); } int main(){ int x = 10, y = 12, z = 5, t; t = Area(x, y); // int Add(int, int) cout << "Area of Rectangle = " << t; int z = 5, u; u = Area(z); // int Add(int) cout << " Area of Square = " << u << endl; return 0; }</pre>
<pre>int sum = 11 double sum = 7.75</pre> <ul style="list-style-type: none"> Same Add function Same # of parameters but different types 	<pre>Area of Rectangle = 12 Area of Square = 25</pre> <ul style="list-style-type: none"> Same Area function Different # of parameters



Program 08.05: Restrictions in Function Overloading

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Two functions having the same signature but different return types cannot be overloaded

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

int Area(int a, int b) { return (a * b); }
double Area(int a, int b) { return (a * b); }
// Error C2556: 'double Area(int,int)': overloaded function differs only by return type
//           from 'int Area(int,int)'
// Error C2371: 'Area': redefinition; different basic types

int main() {
    int x = 10, y = 12, z = 5, t;
    double f;

    t = Area(x, y);
    // Error C2568: '=': unable to resolve function overload
    // Error C3861: 'Area': identifier not found

    cout << "Multiplication = " << t << endl;

    f = Area(y, z); // Errors C2568 and C3861 as above
    cout << "Multiplication = " << f << endl;

    return 0;
}
```



Function Overloading – Summary of Rules

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- The same function name may be used in several definitions
- Functions with the same name must have different number of formal parameters and/or different types of formal parameters
- Function selection is based on the number and the types of the actual parameters at the places of invocation
- Function selection (Overload Resolution) is performed by the compiler
- Two functions having the same signature but different return types will result in a compilation error due to *attempt to re-declare*
- Overloading allows **Static Polymorphism**



Overload Resolution

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- To resolve overloaded functions with one parameter
 - Identify the set of *Candidate Functions*
 - From the set of candidate functions identify the set of *Viable Functions*
 - Select the *Best viable function* through (*Order is important*)
 - Exact Match
 - Promotion
 - Standard type conversion
 - User defined type conversion



Overload Resolution: Exact Match

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- lvalue-to-rvalue conversion
 - Most common
- Array-to-pointer conversion

Definitions: `int ar[10];`
`void f(int *a);`

Call: `f(ar)`
- Function-to-pointer conversion

Definitions: `typedef int (*fp) (int);`
`void f(int, fp);`
`int g(int);`

Call: `f(5, g)`
- Qualification conversion
 - Converting pointer (only) to const pointer



Overload Resolution: Promotion & Conversion

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Examples of Promotion
 - char to int; float to double
 - enum to int / short / unsigned int / ...
 - bool to int
- Examples of Standard Conversion
 - integral conversion
 - floating point conversion
 - floating point to integral conversion

The above 3 may be dangerous!

 - pointer conversion
 - bool conversion



Example: Overload Resolution with one parameter

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- In the context of a list of function prototypes:

```
int g(double);           // F1
void f();                // F2
void f(int);             // F3
double h(void);          // F4
int g(char, int);        // F5
void f(double, double = 3.4); // F6
void h(int, double);     // F7
void f(char, char *);    // F8
```

The call site to resolve is:

```
f(5.6);
```

- Resolution:

- Candidate functions (by name): F2, F3, F6, F8
- Viable functions (by # of parameters): F3, F6
- Best viable function (by type double – Exact Match): F6



Example: Overload Resolution fails

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Consider the overloaded function signatures:

```
int fun(float a) {...}           // Function 1
int fun(float a, int b) {...}    // Function 2
int fun(float x, int y = 5) {...} // Function 3

int main() {
    float p = 4.5, t = 10.5;
    int s = 30;

    fun(p, s); // CALL - 1
    fun(t);    // CALL - 2
    return 0;
}
```

- CALL - 1: Matches Function 2 & Function 3
- CALL - 2: Matches Function 1 & Function 3
- Results in ambiguity



Program 08.06/07: Default Parameter & Function Overload

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Compilers deal with default parameters as a special case of function overloading

Default Parameters	Function Overload
<pre>#include <iostream> using namespace std; int f(int a = 1, int b = 2); int main() { int x = 5, y = 6; f(); // a = 1, b = 2 f(x); // a = x = 5, b = 2 f(x, y); // a = x = 5, b = y = 6 return 0; }</pre> <ul style="list-style-type: none">Function <code>f</code> has 2 parameters overloaded<code>f</code> can have 3 possible forms of call	<pre>#include <iostream> using namespace std; int f(); int f(int); int f(int, int); int main() { int x = 5, y = 6; f(); // int f(); f(x); // int f(int); f(x, y); // int f(int, int); return 0; }</pre> <ul style="list-style-type: none">Function <code>f</code> is overloaded with up to 3 parameters<code>f</code> can have 3 possible forms of callNo overload here use default parameters



Program 08.08: Default Parameter & Function Overload

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Function overloading can use default parameter
- However, with default parameters, the overloaded functions should still be resolvable

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

int Area(int a, int b = 10) { return (a * b); }
double Area(double c, double d) { return (c * d); }

int main() {
    int x = 10, y = 12, t;
    double z = 20.5, u = 5.0, f;

    t = Area(x);      // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // t = 100

    f = Area(z, y); // Binds double Area(double, double)
    cout << "Area = " << f << endl; // f = 102.5

    return 0;
}
```



Program 08.09: Default Parameter & Function Overload

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Function overloading with default parameters may fail

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();      // Error C2668: 'f': ambiguous call to overloaded function
              // More than one instance of overloaded function "f"
              // matches the argument list:
              //     function "f()"
              //     function "f(int = 0)"

    f(x);    // int f(int);
    f(x, y); // int f(int, int);

    return 0;
}
```



Module Summary

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

- Introduced the notion of Default parameters and discussed several examples
- Identified the necessity of function overloading
- Introduced static Polymorphism and discussed examples and restrictions
- Discussed an outline for Overload resolution
- Discussed the mix of default Parameters and function overloading



Instructor and TAs

Module 08

Partha Pratim
Das

Objectives &
Outline

Default
Parameter

Function
Overloading

Overload
Resolution

Default
Parameters in
Overloading

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples
String
Enum

Operator
Overloading
Rules

Summary

Module 09: Programming in C++

Operator Overloading

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples

String
Enum

Operator
Overloading
Rules

Summary

- Understand the Operator Overloading



Module Outline

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples

String
Enum

Operator
Overloading
Rules

Summary

- Basic Differences between Operators & Functions
- Operator Overloading
- Examples of Operator Overloading
 - `operator+` for String & Enum
- Operator Overloading Rules



Operator & Function

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples

String
Enum

Operator
Overloading
Rules

Summary

- What is the difference between an operator & a function?

```
unsigned int Multiply(unsigned x, unsigned y) {  
    int prod = 0;  
    while (y-- > 0) prod += x;  
    return prod;  
}  
  
int main() {  
    unsigned int a = 2, b = 3;  
  
    // Computed by '*' operator  
    unsigned int c = a * b;           // c is 6  
  
    // Computed by Multiply function  
    unsigned int d = Multiply(a, b); // d is 6  
  
    return 0;  
}
```

- Same computation by an operator and a function



Difference between Operator & Functions

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples
String
Enum

Operator
Overloading
Rules

Summary

Operator	Function
<ul style="list-style-type: none">Usually written in infix notationExamples: Infix: <code>a + b; a ? b : c;</code> Prefix: <code>++a;</code> Postfix: <code>a++;</code>Operates on one or more operands, typically up to 3 (Unary, Binary or Ternary)Produces one resultOrder of operations is decided by precedence and associativityOperators are pre-defined	<ul style="list-style-type: none">Always written in prefix notationExamples: Prefix: <code>max(a, b);</code> <code>qsort(int[], int, int,</code> <code>void (*) (void*, void*));</code>Operates on zero or more argumentsProduces up to one resultOrder of application is decided by depth of nestingFunctions can be defined as needed



Operator Functions in C++

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples

String
Enum

Operator
Overloading
Rules

Summary

- Introduces a new keyword: `operator`
- Every operator is associated with an operator function that defines its behavior

Operator Expression	Operator Function
<code>a + b</code>	<code>operator+(a, b)</code>
<code>a = b</code>	<code>operator=(a, b)</code>
<code>c = a + b</code>	<code>operator=(c, operator+(a, b))</code>

- Operator functions are implicit for predefined operators of built-in types and cannot be redefined
- An operator function may have a signature as:

```
MyType a, b; // An enum or struct
```

```
MyType operator+(MyType, MyType); // Operator function
```

```
a + b // Calls operator+(a, b)
```

- C++ allows users to define an operator function and overload it



Program 09.01: String Concatenation

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples
String
Enum

Operator
Overloading
Rules

Summary

Concatenation by string functions

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str;
} String;
int main(){
    String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das" );
    name.str = (char *) malloc(
        strlen(fName.str) +
        strlen(lName.str) + 1);
    strcpy(name.str, fName.str);
    strcat(name.str, lName.str);

    cout << "First Name: " <<
        fName.str << endl;
    cout << "Last Name: " <<
        lName.str << endl;
    cout << "Full Name: " <<
        name.str << endl;
    return 0;
}
-----
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Concatenation operator

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
String operator+(const String& s1, const String& s2)
{
    String s;
    s.str = (char *) malloc(strlen(s1.str) +
                           strlen(s2.str) + 1);
    strcpy(s.str, s1.str);
    strcat(s.str, s2.str);
    return s;
}
int main() {
    String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das");
    name = fName + lName; // Overload operator +
    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
    return 0;
}
-----
First Name: Partha
Last Name: Das
Full Name: Partha Das
Partha Pratim Das
```



Program 09.02: A new semantics for operator +

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples
String
Enum

Operator
Overloading
Rules

Summary

	w/o Overloading +	Overloading operator +
	<pre>#include <iostream> using namespace std; enum E {C0 = 0, C1 = 1, C2 = 2}; int main() { E a = C1, b = C2; int x = -1; x = a + b; cout << x << endl; return 0; } ----- 3</pre>	<pre>#include <iostream> using namespace std; enum E {C0 = 0, C1 = 1, C2 = 2}; E operator+(const E& a, const E& b) { unsigned int uia = a, uib = b; unsigned int t = (uia + uib) % 3; return (E) t; } int main() { E a = C1, b = C2; int x = -1; x = a + b; cout << x << endl; return 0; } ----- 0</pre>
	<ul style="list-style-type: none">• Implicitly converts enum E values to int• Adds by operator+ of int• Result is outside enum E range	<ul style="list-style-type: none">• operator + is overloaded for enum E• Result is a valid enum E value



Operator Overloading – Summary of Rules

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples

String
Enum

Operator
Overloading
Rules

Summary

- No new operator such as `**`, `<>`, or `&|` can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be changed
 - Preserves arity
 - Preserves precedence
 - Preserves associativity
- These operators can be overloaded:
`[] + - * / % & | ~ ! = += -= *= /= %= = &= |=`
`<< >> >>= <<= == != < > <= >= && || ++ -- , ->* -> () []`
- For unary prefix operators, use: `MyType& operator++(MyType& s1)`
- For unary postfix operators, use: `MyType operator++(MyType& s1, int)`
- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), `sizeof`, and `?:` (ternary conditional) cannot be overloaded
- The overloads of operators `&&`, `||`, and `,` (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of `operator->` must either return a raw pointer or return an object (by reference or by value), for which `operator->` is in turn overloaded



Overloading disallowed for

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples

String
Enum

Operator
Overloading
Rules

Summary

Operator	Reason
• dot (.)	• It will raise question whether it is for object reference or overloading
• Scope Resolution (::)	• It performs a (compile time) scope resolution rather than an expression evaluation.
• Ternary (? :)	• overloading <code>expr1 ? expr2 : expr3</code> would not be able to guarantee that only one of <code>expr2</code> and <code>expr3</code> was executed
• sizeof	• <code>sizeof</code> cannot be overloaded because built-in operations, such as incrementing a pointer into an array implicitly depends on it



Do not overload these operators

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples
String
Enum

Operator
Overloading
Rules

Summary

Operator	Reason
• && and	• In evaluation, the second operand is not evaluated if the result can be deduced solely by evaluating the first operand. However, this evaluation is not possible for overloaded versions of these operators
• Comma (,)	• This operator guarantees that the first operand is evaluated before the second operand. However, if the comma operator is overloaded, its operand evaluation depends on C++'s function parameter mechanism, which does not guarantee the order of evaluation
• Ampersand (&)	• The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares operator &() as a member function, then the behavior is undefined



Module Summary

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples

String
Enum

Operator
Overloading
Rules

Summary

- Introduced operator overloading
- Explained the rules of operator overloading



Instructor and TAs

Module 09

Partha Pratim
Das

Objectives &
Outline

Operators &
Functions

Operator
Overloading

Examples

String
Enum

Operator
Overloading
Rules

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C
`malloc & free`

Memory
Management
in C++
`new & delete`
Array
Placement new
Restrictions

Overloading
`new & delete`

Summary

Module 10: Programming in C++

Dynamic Memory Management

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

`ppd@cse.iitkgp.ernet.in`

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C

malloc & free

Memory
Management
in C++

new & delete
Array
Placement new
Restrictions

Overloading
new & delete

Summary

- Understand the dynamic memory management in C++



Module Outline

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C
`malloc & free`

Memory
Management
in C++
`new & delete`
Array
Placement new
Restrictions

Overloading
`new & delete`

Summary

- Memory management in C
 - `malloc()` & `free()`
- Memory management in C++
 - `new` and `delete`
 - Array `new[]` and `delete[]`
 - Placement `new()`
 - Restrictions
- Overloading `new` and `delete`



Program 10.01/02: malloc() & free(): C & C++

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C
malloc & free

Memory
Management
in C++
new & delete

Array
Placement new
Restrictions

Overloading
new & delete

Summary

C Program	C++ Program
#include <stdio.h> #include <stdlib.h> int main() { int *p = (int *)malloc(sizeof(int)); *p = 5; printf("%d", *p); free(p); return 0; } ----- 5	#include <iostream> #include <cstdlib> using namespace std; int main() { int *p = (int *)malloc(sizeof(int)); *p = 5; cout << *p; free(p); return 0; } ----- 5

- Dynamic memory management functions in stdlib.h header for C (cstdlib header for C++)
- malloc() allocates the memory on heap
- sizeof(int) needs to be provided
- Pointer to allocated memory returned as void * – needs cast to int *
- Allocated memory is released by free() from heap
- calloc() and realloc() also available in both languages



Program 10.02/03: operator new & delete: Dynamic memory management in C++

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C
malloc & free

Memory
Management
in C++

new & delete

Array

Placement new
Restrictions

Overloading
new & delete

Summary

- C++ introduces operators **new** and **delete** to dynamically allocate and de-allocate memory:

malloc() & free()	Operators new & delete
<pre>#include <iostream> #include <cstdlib> using namespace std; int main() { int *p = (int *)malloc(sizeof(int)); *p = 5; cout << *p; free(p); return 0; }-----5</pre>	<pre>#include <iostream> using namespace std; int main() { int *p = new int(5); cout << *p; delete p; return 0; }-----5</pre>
<ul style="list-style-type: none">● Function malloc() for allocation on heap● sizeof(int) needs to be provided● Allocated memory returned as void *● Casting to int * needed● Cannot be initialized● Function free() for de-allocation from heap● Library feature – header cstdlib needed	<ul style="list-style-type: none">● Operator new for allocation on heap● No size specification needed, type suffices● Allocated memory returned as int *● No casting needed● Can be initialized● Operator delete for de-allocation from heap● Core language feature – no header needed



Program 10.02/04: Functions: operator new() & operator delete()

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C
malloc & free

Memory
Management
in C++
new & delete

Array
Placement new
Restrictions

Overloading
new & delete

Summary

- C++ also allows `operator new` and `operator delete` functions to dynamically allocate and de-allocate memory:

malloc() & free()	new & delete
<pre>#include <iostream> #include <cstdlib> using namespace std; int main() { int *p = (int *)malloc(sizeof(int)); *p = 5; cout << *p; free(p); return 0; }-----5</pre>	<pre>#include <iostream> #include <cstdlib> using namespace std; int main(){ int *p = (int *)operator new(sizeof(int)); *p = 5; cout << *p; operator delete(p); return 0; }-----5</pre>
<ul style="list-style-type: none">● Function <code>malloc()</code> for allocation on heap● Function <code>free()</code> for de-allocation from heap	<ul style="list-style-type: none">● Function <code>operator new()</code> for allocation on heap● Function <code>operator delete()</code> for de-allocation from heap

There is a major difference between `operator new` and function `operator new()`. We explore this angle more after we learn about classes



Program 10.05/06: Operators new[] & delete[]: Dynamically managed Arrays in C++

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C
malloc & free

Memory
Management
in C++
new & delete
Array

Placement new
Restrictions

Overloading
new & delete

Summary

malloc() & free()

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

int main() {
    int *a = (int *)malloc(sizeof(int)* 3);
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
            << a[i] << "      ";
    cout << endl;

    free(a);

    return 0;
}
-----
a[0] = 10      a[1] = 20      a[2] = 30
```

- Allocation by malloc() on heap
- # of elements implicit in size passed to malloc()
- Release by free() from heap

new[] & delete[]

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

int main() {
    int *a = new int[3];
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
            << a[i] << "      ";
    cout << endl;

    delete [] a;

    return 0;
}
-----
a[0] = 10      a[1] = 20      a[2] = 30
```

- Allocation by operator new[] (**different from** operator new)
- # of elements explicitly passed to operator new[]
- Release by operator delete[] (**different from** operator delete) from heap



Program 10.07: Operator new(): Placement new in C++

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C

malloc & free

Memory
Management
in C++

new & delete
Array

Placement new
Restrictions

Overloading
new & delete

Summary

```
#include <iostream> using namespace std;
int main() {
    unsigned char buf[sizeof(int)* 2]; // Buffer on stack

    // placement new in buffer buf
    int *pInt = new (buf) int (3); int *qInt = new (buf+sizeof(int)) int (5);

    int *pBuf = (int *) (buf + 0); int *qBuf = (int *) (buf + sizeof(int));
    cout << "Buf Addr  Int Addr" << endl;
    cout << pBuf << "  " << pInt << endl << qBuf << "  " << qInt << endl;
    cout << "1st Int  2nd Int" << endl;
    cout << *pBuf << "           " << *qBuf << endl;

    int *rInt = new int(7); // heap allocation
    cout << "Heap Addr  3rd Int" << endl;
    cout << rInt << "           " << *rInt << endl;
    delete rInt;           // delete integer from heap

    // No delete for placement new

    return 0;
}
-----
Buf Addr  Int Addr
001BFC50  001BFC50
001BFC54  001BFC54
1st Int  2nd Int
3      5
Heap Addr  3rd Int
003799B8  7
```

- Placement new operator takes a buffer address to place objects
- These are not dynamically allocated on heap –
may be allocated on stack
- Allocations by Placement new operator must not be deleted



Mixing malloc, operator new, etc

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C

malloc & free

Memory
Management
in C++

new & delete
Array

Placement new
Restrictions

Overloading
new & delete

Summary

- Allocation and De-Allocation must correctly match. Do not free the space created by new using free(). And do not use delete if memory is allocated through malloc(). These may result in memory corruption

Allocator	De-allocator
malloc()	free()
operator new	operator delete
operator new[]	operator delete[]
operator new()	No delete

- Passing NULL pointer to delete operator is secure
- Prefer to use only new and delete in a C++ program
- The new operator allocates exact amount of memory from Heap
- new returns the given pointer type – no need to typecast
- new, new[] and delete, delete[] have separate semantics



Program 10.08: Overloading operator new

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C

malloc & free

Memory
Management
in C++

new & delete

Array

Placement new
Restrictions

Overloading
new & delete

Summary

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void* operator new(size_t n) { // Definition of new
    cout << "Overloaded new" << endl;
    void *ptr;
    ptr = malloc(n);           // Memory allocated to ptr
    return ptr;
}
void operator delete(void *p) { // definition of delete
    cout << "Overloaded delete" << endl;
    free(p);                  // Allocated memory released
}
int main() {
    int *p = new int; // calling overloaded operator new
    *p = 30;          // Assign value to the location
    cout << "The value is :\t" << *p << endl;
    delete p;         // calling overloaded operator delete
    return 0;
}
```

Overloaded new

The value is : 30

Overloaded delete

- operator new overloaded
- The first parameter of overloaded operator new must be size_t
- The return type of overloaded operator new must be void *
- The first parameter of overloaded operator delete must be void *
- The return type of overloaded operator delete must be void
- More parameters may be used for overloading
- operator delete should not be overloaded (usually) with extra parameters



Program 10.09: Overloading operator new[]

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C
malloc & free

Memory
Management
in C++
new & delete
Array
Placement new
Restrictions

Overloading
new & delete

Summary

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

void* operator new [] (size_t os, char setv) { // Fill the allocated array with setv
    void *t = operator new(os);
    memset(t, setv, os);
    return t;
}

void operator delete[] (void *ss) {
    operator delete(ss);
}

int main() {
    char *t = new('#')char[10]; // Allocate array of 10 elements and fill with '#'

    cout << "p = " << (int) (t) << endl;
    for (int k = 0; k < 10; ++k)
        cout << t[k];

    delete [] t;
    return 0;
}
-----
p = 19421992
#####
```

- **operator new[] overloaded with initialization**
- **The first parameter of overloaded operator new[] must be size_t**
- **The return type of overloaded operator new[] must be void ***
- **Multiple parameters may be used for overloading**
- **operator delete [] should not be overloaded (usually) with extra parameters**



Module Summary

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C
`malloc & free`

Memory
Management
in C++
`new & delete`

Array
Placement new
Restrictions

Overloading
`new & delete`

Summary

- Introduced `new` and `delete` for dynamic memory management in C++
- Understood the difference between `new`, `new[]` and `delete`, `delete[]`
- Compared memory management in C with C++
- Explored the overloading of `new`, `new[]` and `delete`, `delete[]` operators



Instructor and TAs

Module 10

Partha Pratim
Das

Objectives &
Outline

Memory
Management
in C

malloc & free

Memory
Management
in C++

new & delete
Array
Placement new
Restrictions

Overloading
new & delete

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

Module 11: Programming in C++

Classes and Objects

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

- Understand the concept of classes and objects in C++



Module Outline

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex

Rectangle

Stack

Member
Functions

Complex

Rectangle

Stack

this pointer

State of an
Object

Complex

Rectangle

Stack

Summary

- Classes
- Objects
- Data Members of a class
- Member functions of a class
- **this** Pointer
- State of an Object



Classes

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex

Rectangle

Stack

Member

Functions

Complex

Rectangle

Stack

this pointer

State of an
Object

Complex

Rectangle

Stack

Summary

- A class is an implementation of a *type*. It is the only way to implement **User-defined Data Type (UDT)**
- A class contains **data members / attributes**
- A class has **operations / member functions / methods**
- A class defines a **namespace**
- Thus, classes offer **data abstraction / encapsulation** of **Object Oriented Programming**
- Classes are similar to structures that aggregate data logically
- A class is defined by **class keyword**
- Classes provide **access specifiers** for members to enforce **data hiding** that separates **implementation** from **interface**
 - **private** - accessible inside the definition of the class
 - **public** - accessible everywhere
- A class is a **blue print** for its instances (objects)



Objects

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex

Rectangle

Stack

Member

Functions

Complex

Rectangle

Stack

this pointer

State of an
Object

Complex

Rectangle

Stack

Summary

- An **object** of a class is an **instance** created according to its **blue print**. Objects can be automatically, statically, or dynamically created
- A object comprises **data members** that specify its **state**
- A object supports **member functions** that specify its **behavior**
- Data members of an object can be accessed by " ." (dot) operator on the object
- Member functions are invoked by " ." (dot) operator on the object
- An implicit **this** pointer holds the address of an object. This serves the **identity** of the object in C++
- **this** pointer is implicitly passed to methods



Program 11.01/02: Complex Numbers: Attributes

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex

Rectangle

Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

	C Program	C++ Program
	<pre>// File Name:Complex_object.c: #include <stdio.h> typedef struct Complex { // struct double re, im; // Data members } Complex; int main() { // Variable n1 declared, initialized Complex n1 = {4.2, 5.3}; printf("%d %d", n1.re, n1.im); // Use return 0; } ----- 4.2 5.3</pre>	<pre>// File Name:Complex_object_c++.cpp: #include <iostream> using namespace std; class Complex { public: // class double re, im; // Data members }; int main() { // Object n1 declared, initialized Complex n1 = {4.2, 5.3}; cout << n1.re << " " << n1.im; // Use return 0; } ----- 4.2 5.3</pre>
	<ul style="list-style-type: none">● struct is a keyword in C for data aggregation● The struct Complex is defined as composite data type containing two double (re, im) data members● struct Complex is a derived data type used to create Complex type variable n1● Data members are accessed using '.' operator● struct only aggregates	<ul style="list-style-type: none">● class is a new keyword in C+ for data aggregation● The class Complex is defined as composite data type containing two double (re, im) data members● class Complex is User-defined Data Type (UDT) used to create Complex type object n1● Data members are accessed using '.' operator.● class aggregates and helps to do more for building a UDT



Program 11.03/04: Points and Rectangles: Attributes

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex

Rectangle

Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

C Program	C++ Program
<pre>// File Name:Rectangle_object.c: #include <stdio.h> typedef struct { // struct Point int x; int y; } Point; typedef struct { // Rect uses Point Point TL; // Top-Left Point BR; // Bottom-Right } Rect; int main() { Rect r = {{0,2}, {5,7}}; // r.TL <- {0,2}; r.BR <- {5,7} // r.TL.x <- 0; r.TL.y <- 2 // Members of structure r accessed printf("[%d %d) (%d %d]", r.TL.x, r.TL.y, r.BR.x, r.BR.y); return 0; } ----- [(0 2) (5 7)]</pre>	<pre>// File Name:Rectangle_object_c++.cpp: #include <iostream> using namespace std; class Point { public: // class Point int x; int y; // Data members }; class Rect { public: // Rect uses Point Point TL; // Top-Left Point BR; // Bottom-Right }; int main() { Rect r = {{0,2}, {5,7}}; // r.TL <- {0,2}; r.BR <- {5,7} // r.TL.x <- 0; r.TL.y <- 2 // Rectangle Object r accessed cout << "[" << r.TL.x << " " << r.TL.y << ") (" << r.BR.x << " " << r.BR.y << "]")"; return 0; } ----- [(0 2) (5 7)]</pre>

- Data members of user-defined data types



Program 11.05/06: Stacks: Attributes

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

C Program

```
// File Name:Stack_object.c:  
#include <stdio.h>  
  
typedef struct Stack { // struct Stack  
    char data [100];  
    int top;  
} Stack;  
  
// Codes for push, pop, top, empty  
  
int main() {  
    // Variable s declared  
    Stack s;  
    s.top = -1;  
  
    // Using stack for solving problems  
  
    return 0;  
}
```

C++ Program

```
// File Name:Stack_object_c++.cpp:  
#include <iostream>  
using namespace std;  
  
class Stack { public: // class Stack  
    char data [100];  
    int top;  
};  
  
// Codes for push, pop, top, empty  
  
int main() {  
    // Object s declared  
    Stack s;  
    s.top = -1;  
  
    // Using stack for solving problems  
  
    return 0;  
}
```

- Data members of mixed data types



Classes

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

- A class is an implementation of a *type*. It is the only way to implement **User-defined Data Type (UDT)**
- A class contains **data members / attributes**.
- A class defines a **namespace**
- Thus, classes offer **data abstraction / encapsulation** of **Object Oriented Programming**
- Classes are similar to structures that aggregate data logically
- A class is a **blue print** for its instances (objects)



Objects

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

- An **object** of a class is an **instance** created according to its **blue print**. Objects can be automatically, statically, or dynamically created
- A object comprises **data members** that specify its **state**
- Data members of an object can be accessed by ":" (dot) operator on the object



Program 11.07/08: Complex Numbers: Methods

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex

Rectangle

Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

C Program

```
// File Name:Complex_func.c:  
#include <stdio.h>  
#include <math.h>  
  
typedef struct Complex {  
    double re, im;  
} Complex;  
  
// Norm of Complex Number - global fn.  
double norm(Complex c) {  
    return sqrt(c.re*c.re + c.im*c.im);  
}  
// Print number with Norm - global fn.  
void print(Complex c) {  
    printf("|\%lf+j%lf| = ", c.re, c.im);  
    printf("%lf", norm(c)); // Call global  
}  
  
int main() { Complex c = { 4.2, 5.3 };  
  
    // Call global fn. with 'c' as param  
    print(c);  
  
    return 0;  
}  
----  
|4.200000+j5.300000| = 6.762396
```

C++ Program

```
// File Name:Complex_func_c++.cpp:  
#include <iostream>  
#include <cmath>  
using namespace std;  
  
class Complex { public:  
    double re, im;  
  
    // MEMBER FUNCTIONS / METHODS  
    // Norm of Complex Number - method  
    double norm() {  
        return sqrt(re*re + im*im);  
    }  
    // Print number with Norm - method  
    void print() {  
        cout << "|\\" << re << "+j" << im << "| = ";  
        cout << norm(); // Call method  
    }  
}; // End of class Complex  
int main() { Complex c = { 4.2, 5.3 };  
  
    // Invoke method print of 'c'  
    c.print();  
  
    return 0;  
}  
----  
|4.2+j5.3| = 6.7624
```



Program 11.09/10: Rectangles: Methods

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

Using struct

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

typedef struct {
    int x; int y;
} Point;
typedef struct {
    Point TL; // Top-Left
    Point BR; // Bottom-Right
} Rect;

// Global function
void computeArea(Rect r) {
    cout << abs(r.TL.x - r.BR.x) *
        abs(r.BR.y - r.TL.y);
}

int main() {
    Rect r = { { 0, 2 }, { 5, 7 } };

    // Global fn. call
    computeArea(r);

    return 0;
}
-----
```

Using class

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

class Point { public:
    int x; int y;
};

class Rect { public:
    Point TL; // Top-Left
    Point BR; // Bottom-Right

    // Method
    void computeArea() {
        cout << abs(TL.x - BR.x) *
            abs(BR.y - TL.y);
    }
};

int main() {
    Rect r = { { 0, 2 }, { 5, 7 } };

    // Method invocation
    r.computeArea();

    return 0;
}
-----
```



Program 11.11/12: Stacks: Methods

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

Using struct

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

typedef struct Stack {
    char data_[100]; int top_;
} Stack;
bool empty(const Stack& s)
{ return (s.top_ == -1); }
char top(const Stack& s)
{ return s.data_[s.top_]; }
void push(Stack& s, char x)
{ s.data_[++(s.top_)] = x; }
void pop(Stack& s) { --(s.top_); }

int main()
{
    Stack s; s.top_ = -1;
    char str[10] = "ABCDE"; int i;

    for (i = 0; i < 5; ++i) push(s, str[i]);

    cout << "Reversed String: ";
    while (!empty(s)) {
        cout << top(s); pop(s);
    }
    return 0;
}
-----
Reversed String: EDCBA
```

Using class

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

class Stack { public:
    char data_[100]; int top_;
    // METHODS
    bool empty() { return (top_ == -1); }

    char top() { return data_[top_]; }

    void push(char x) { data_[++top_] = x; }

    void pop() { --top_; }

};

int main()
{
    Stack s; s.top_ = -1;
    char str[10] = "ABCDE"; int i;

    for (i = 0; i < 5; ++i) s.push(str[i]);

    cout << "Reversed String: ";
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
-----
Reversed String: EDCBA
```



Classes

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

- A class has **operations / member functions / methods**
- A class defines a **namespace**
- Thus, classes offer **data abstraction / encapsulation of Object Oriented Programming**



Objects

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

- An **object** of a class is an **instance** created according to its **blue print**. Objects can be automatically, statically, or dynamically created
- A object supports **member functions** that specify its **behavior**
- Member functions are invoked by “.” (dot) operator on the object



Program 11.13: this Pointer

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

- An implicit this pointer holds the address of an object
- this pointer serves as the **identity** of the object in C++
- Type of this pointer for a class X object: X * const this;
- this pointer is accessible *only in methods*

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

class X { public: int m1, m2;
    void f(int k1, int k2) {                      // Sample Method
        m1 = k1;                                // Implicit access w/o 'this' pointer
        this->m2 = k2;                            // Explicit access w/ 'this' pointer
        cout << "Id   = " << this << endl; // Identity (address) of the object
    }
};

int main() {
    X a;
    a.f(2, 3);
    cout << "Addr = " << &a << endl;      // Address (identity) of the object
    cout << "a.m1 = " << a.m1 << " a.m2 = " << a.m2 << endl;
    return 0;
}
-----
Id   = 0024F918
Addr = 0024F918
a.m1 = 2 a.m2 = 3
```



this Pointer

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

- this pointer is implicitly passed to methods

In Source Code	In Binary Code
class X { void f(int, int); ... }	void X::f(X * const this, int, int);
X a; a.f(2, 3);	X::f(&a, 2, 3); // &a = this

- Use of this pointer

- Distinguish member from non-member

```
class X { public: int m1, m2;
           void f(int k1, int k2) {
               m1 = k1;           // this->m1 (member) is valid; this->k1 is invalid
               this->m2 = k2; // m2 (member) is valid; this->k2 is invalid
           }
       };
```

- Explicit Use

```
// Link the object
class DoublyLinkedListNode { public: DoublyLinkedListNode *prev, *next; int data;
                           void append(DoublyLinkedListNode *x) { next = x; x->prev = this; }
                           }
                           ---
                           // Return the object
Complex& inc() { ++re; ++im; return *this; }
```



State of an Object: Complex

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex

Rectangle

Stack

Member

Functions

Complex

Rectangle

Stack

this pointer

State of an
Object

Complex

Rectangle

Stack

Summary

- The state of an object is determined by the combined value of all its data members. Consider class Complex:

```
class Complex { public:  
    double re_, im_; // ordered tuple of data members decide the state at any time  
  
    double get_re { return re_; }  
    void set_re(double re) { re_ = re; }  
    double get_im { return im_; }  
    void set_im(double im) { im_ = im; }  
}  
  
Complex c1 = {4.2, 5.3};  
// STATE 1 of c1 = {4.2, 5.3} // Denotes a tuple / sequence
```

- A method may change the state:

```
Complex c = {4.2, 5.3};  
// STATE 1 of c = {4.2, 5.3}  
  
c.set_re(6.4);  
// STATE 2 of c = {6.4, 5.3}  
  
c.get_re();  
// STATE 2 of c = {6.4, 5.3} // No change of state  
  
c.set_im(7.8);  
// STATE 3 of c = {6.4, 7.8}
```



State of an Object: Rectangle

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

- Consider class Point and class Rect:

```
 Data members of Rect class: Point TL; Point BR; // Point class type object
 Data members of Point class: int x; int y

 Rectangle r = {{0, 5}, {5, 0}}; // Initialization
 // STATE 1 of r = {{0, 5}, {5, 0}}
 { r.TL.x = 0; r.TL.y = 5; r.BR.x = 5; r.BR.y = 0 }

 r.TL.y = 9;
 // STATE 2 of r = {{0, 9}, {5, 0}}

 r.computeArea();
 // STATE 2 of r = {{0, 9}, {5, 0}} // No change in state

 Point p = {3, 4};
 r.BR = p;
 // STATE 3 of r = {{0, 9}, {3, 4}}
```



State of an Object: Stack

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object
Complex
Rectangle
Stack

Summary

Consider class Stack:

```
Data members of Stack class: char data[5] and int top;

Stack s;
// STATE 1 of s = {{?, ?, ?, ?, ?}, ?} // No data member is initialized

s.top_ = -1;
// STATE 2 of s = {{?, ?, ?, ?, ?}, -1}

s.push('b');
// STATE 3 of s = {{'b', ?, ?, ?, ?}, 0}

s.push('a');
// STATE 4 of s = {{'b', 'a', ?, ?, ?}, 1}

s.empty();
// STATE 4 of s = {{'b', 'a', ?, ?, ?}, 1} // No change of state

s.push('t');
// STATE 5 of s = {{'b', 'a', 't', ?, ?}, 2}

s.top();
// STATE 5 of s = {{'b', 'a', 't', ?, ?}, 2} // No change of state

s.pop();
// STATE 6 of s = {{'b', 'a', 't', ?, ?}, 1}
```



Module Summary

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex

Rectangle

Stack

Member
Functions

Complex

Rectangle

Stack

this pointer

State of an
Object

Complex

Rectangle

Stack

Summary

- We have covered the following:

	<pre>class Complex { public: double re_, im_;</pre>
Class	<pre> double norm() { // Norm of Complex Number return sqrt(re_ * re_ + im_ * im_); } };</pre>
Attributes	<pre>Complex::re_, Complex::re_im_</pre>
Method	<pre>double Complex::norm();</pre>
Object	<pre>Complex c = {2.6, 3.9};</pre>
Access	<pre>c.re_ = 4.6; cout << c.im_;</pre> <pre>cout << c.norm;</pre>
this Pointer	<pre>double Complex::norm() { cout << this; return ... }</pre>



Instructor and TAs

Module 11

Partha Pratim
Das

Objectives &
Outline

Classes

Objects

Data Members

Complex
Rectangle
Stack

Member
Functions

Complex
Rectangle
Stack

this pointer

State of an
Object

Complex
Rectangle
Stack

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

Module 12: Programming in C++

Access Specifiers

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

- Understand access specifiers in C++ classes to control the visibility of members
- Learn to design with Information Hiding



Module Outline

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers
public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

- Access specifiers
 - public Access Specifier
 - private Access Specifier
- Information Hiding
 - Stack with public data
 - Stack with private data
- Get-Set Idiom



Program 12.01/02: Complex Number: Access Specification

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

Public data, Public method

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

class Complex { public: double re, im;
public: double norm() {
            return sqrt(re*re + im*im);
        }
    };
void print(const Complex& t) { // Global fn.
    cout << t.re << "+j" << t.im << endl;
}

int main() {
    Complex c = { 4.2, 5.3 }; // Okay

    print(c);
    cout << c.norm();
    return 0;
}
```

- **public data can be accessed by any function**
- **norm (method) can access (re, im)**
- **print (global) can access (re, im)**
- **main (global) can access (re, im) & initialize**

Private data, Public method

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

class Complex { private: double re, im;
public: double norm() {
            return sqrt(re*re + im*im);
        }
    };
void print(const Complex& t) { // Global fn.
    cout << t.re << "+j" << t.im << endl;
    // 'Complex::re': cannot access private
    // member declared in class 'Complex'

    // 'Complex::im': cannot access private
    // member declared in class 'Complex'
}
int main() {
    Complex c = { 4.2, 5.3 }; // Error
    // 'initializing': cannot convert from
    // 'initializer-list' to 'Complex'
    print(c);
    cout << c.norm();
    return 0;
}
```

- **private data can be accessed *only* by methods**
- **norm (method) can access (re, im)**
- **print (global) cannot access (re, im)**
- **main (global) cannot access (re, im) to initialize**



Access Specifiers

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

- Classes provide **access specifiers** for members (data as well as function) to enforce **data hiding** that separates **implementation** from **interface**
 - **private** - accessible inside the definition of the class
 - member functions of the same class
 - **public** - accessible everywhere
 - member functions of the same class
 - member function of a different class
 - global functions
- The keywords **public** and **private** are the *Access Specifiers*
- Unless specified, the access of the members of a class is considered **private**
- A class may have multiple access specifier. The effect of one continues till the next is encountered



Information Hiding

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers
public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

- The private part of a class (*attributes* and *methods*) forms its **implementation** because the class alone should be concerned with it and have the right to change it
- The public part of a class (*attributes* and *methods*) constitutes its **interface** which is available to all others for using the class
- Customarily, we put all *attributes* in private part and the *methods* in public part. This ensures:
 - The **state** of an object can be changed only through one of its *methods* (with the knowledge of the class)
 - The **behavior** of an object is accessible to others through the *methods*
- This is known as **Information Hiding**



Information Hiding

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

- For the sake of efficiency in design, we at times, put *attributes* in public and / or *methods* in private. In such cases:
 - The public *attributes* *should not* decide the *state* of an object, and
 - The private *methods* cannot be part of the behavior of an object

We illustrate information hiding through two implementations a stack



Program 12.03/04: Stack: Implementations using public data

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

Using dynamic array

```
#include <iostream> #include <cstdlib>
using namespace std;
class Stack { public:
    char *data_; int top_;
public:
    int empty() { return (top_ == -1); }
    void push(char x) {data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() {
    Stack s; char str[10] = "ABCDE";
    s.data_ = new char[100]; // Exposed Init
    s.top_ = -1;           // Exposed Init
    for(int i = 0; i < 5; ++i)
        s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    } // Outputs: EDCBA -- Reversed string
    delete [] s.data_;      // Exposed De-Init
    return 0;
}
```

Using vector

```
#include <iostream> #include <vector>
using namespace std;
class Stack { public:
    vector<char> data_; int top_;
public:
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() {
    Stack s; char str[10] = "ABCDE";
    s.data_.resize(100); // Exposed Init
    s.top_ = -1;         // Exposed Init
    for(int i = 0; i < 5; ++i)
        s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    } // Outputs: EDCBA -- Reversed string
    return 0;
}
```

- public data reveals the *internals* of the stack (no information hiding)
- Spills data structure codes (Exposed Init / De-Init) into the application (main)
- To switch from array to vector or vice-versa the application needs to change



Program 12.03/04: Stack: Implementations using public data – Risks

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

Using dynamic array

```
#include <iostream> #include <cstdlib>
using namespace std;
class Stack { public:
    char *data_; int top_;
public:
    int empty() { return (top_ == -1); }
    void push(char x) {data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() {
    Stack s; char str[10] = "ABCDE";
    s.data_ = new char[100]; // Exposed Init
    s.top_ = -1;           // Exposed Init
    for(int i=0; i<5; ++i) s.push(str[i]);
    s.top_ = 2; // STACK GETS INCONSISTENT
    while (!s.empty()) {
        cout << s.top(); s.pop();
    } // Outputs: CBA -- WRONG!!!
    delete [] s.data_;      // Exposed De-Init
    return 0;
}
```

Using vector

```
#include <iostream> #include <vector>
using namespace std;
class Stack { public:
    vector<char> data_; int top_;
public:
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() {
    Stack s; char str[10] = "ABCDE";
    s.data_.resize(100); // Exposed Init
    s.top_ = -1;           // Exposed Init
    for(int i=0; i<5; ++i) s.push(str[i]);
    s.top_ = 2; // STACK GETS INCONSISTENT
    while (!s.empty()) {
        cout << s.top(); s.pop();
    } // Outputs: CBA -- WRONG!!!
    return 0;
}
```

- Application may intentionally or inadvertently tamper the value of `top_` – this corrupts the stack!
- `s.top_ = 2;` destroys consistency of the stack and causes wrong output



Program 12.05/06: Stack: Implementations using private data – Safe

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

Using dynamic array

```
#include <iostream>
using namespace std;
class Stack { private:
    char *data_; int top_;
public:
    // Initialization
    Stack(): data_(new char[100]), top_(-1) {}
    // De-Initialization
    ~Stack() { delete[] data_; }
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() {
    Stack s; char str[10] = "ABCDE";
    for (int i=0; i<5; ++i) s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
```

Using vector

```
#include <iostream>
#include <vector>
using namespace std;
class Stack { private:
    vector<char> data_; int top_;
public:
    // Initialization
    Stack(): top_(-1) { data_.resize(100); }
    // De-Initialization
    ~Stack() {};
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() {
    Stack s; char str[10] = "ABCDE";
    for (int i=0; i<5; ++i) s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
```

- private data hides the *internals* of the stack (information hiding)
- Data structure codes contained within itself with initialization and de-initialization
- To switch from array to vector or vice-versa the application needs *no change*
- Application cannot tamper stack – any direct access to top_ or data_ is compilation error!



Interface and Implementation

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

Interface	Implementation	Application
<pre>// File: Stack.h class Stack { private: // Implementation char *data_; int top_; public: // Interface Stack(); ~Stack(); int empty(); void push(char x); void pop(); char top(); };</pre>	<pre>// File: Stack.h class Stack { private: // Implementation char *data_; int top_; public: // Interface Stack(); ~Stack(); int empty(); void push(char x); void pop(); char top(); }; // File: Stack.cpp // Implementation Stack::Stack(): data_(new char[100]), top_(-1) {} Stack::~Stack() { delete[] data_; } int Stack::empty() { return (top_ == -1); } void Stack::push(char x) { data_[++top_] = x; } void Stack::pop() { --top_; } char Stack::top() { return data_[top_]; }</pre>	

```
#include "Stack.h"
int main() {
    Stack s; char str[10] = "ABCDE";
    for (int i = 0; i < 5; ++i) s.push(str[i]);
    while (!s.empty()) { cout << s.top(); s.pop(); }
    return 0;
}
```



Get–Set Methods: Idiom for fine-grained Access Control

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

- As noted, we put all *attributes* in private and the *methods* in public. This restricts the access to data completely
- To fine-grain the access to data we provide selective public member functions to *read* (get) and / or *write* (set) data

```
class MyClass { // private
    int readWrite_; // Like re_, im_ in Complex -- common aggregated members

    int readOnly_; // Like DateOfBirth, Emp_ID, RollNo -- should not need a change

    int writeOnly_; // Like Password -- reset if forgotten

    int invisible_; // Like top_, data_ in Stack -- keeps internal state

    public:
        // get and set methods both to read as well as write readWrite_ member
        int getReadWrite() { return readWrite_; }
        void setReadWrite(int v) { readWrite_ = v; }

        // Only get method to read readOnly_ member - no way to write it
        int getReadOnly() { return readOnly_; }

        // Only set method to write writeOnly_ member - no way to read it
        void setWriteOnly(int v) { writeOnly_ = v; }

        // No method accessing invisible_ member directly - no way to read or write it
}
```



Module Summary

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers
public and
private

Information
Hiding
Stack (public)
Stack (private)

Get-Set Idiom

Summary

- Access Specifiers helps to control visibility of data members and methods of a class
- The private access specifier can be used to hide information about the implementation details of the data members and methods
- Get, Set methods are defined to provide an interface to use and access the data members



Instructor and TAs

Module 12

Partha Pratim
Das

Objectives &
Outline

Access
Specifiers

public and
private

Information
Hiding

Stack (public)
Stack (private)

Get-Set Idiom

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655



Module 13

Partha Pratim
Das

Objectives &
Outline

Constructor
Parameterized
Overloaded

Destructor

Default
Constructor

Object
Lifetime

Automatic
Static
Dynamic

Summary

Module 13: Programming in C++

Constructors, Destructors & Object Lifetime

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan



Module Objectives

Module 13

Partha Pratim
Das

Objectives &
Outline

Constructor
Parameterized
Overloaded

Destructor

Default
Constructor

Object
Lifetime

Automatic
Static
Dynamic

Summary

- Understand Object Construction (Initialization)
- Understand Object Destruction (De-Initialization)
- Understand Object Lifetime



Module Outline

Module 13

Partha Pratim
Das

Objectives &
Outline

Constructor
Parameterized
Overloaded

Destructor

Default
Constructor

Object
Lifetime

Automatic
Static
Dynamic

Summary

- Constructors
 - Parameterized
 - Overloaded
- Destructor
- Default Constructor
- Object Lifetime
 - Automatic
 - Array
 - Dynamic



Program 13.01/02: Stack Initialization

Module 13

Partha Pratim
Das

Objectives &
Outline

Constructor
Parameterized
Overloaded

Destructor

Default
Constructor

Object
Lifetime
Automatic
Static
Dynamic

Summary

Public Data

```
#include <iostream>
using namespace std;
class Stack { public: // VULNERABLE DATA
    char data_[10]; int top_;
public:
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { char str[10] = "ABCDE";
    Stack s;
    s.top_ = -1; // Exposed initialization
    for (int i = 0; i < 5; ++i)
        s.push(str[i]);
    // s.top_ = 2; // RISK - CORRUPTS STACK
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
```

Private Data

```
#include <iostream>
using namespace std;
class Stack { private: // PROTECTED DATA
    char data_[10]; int top_;
public:
    void init() { top_ = -1; }
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { char str[10] = "ABCDE";
    Stack s;
    s.init(); // Clean initialization
    for (int i = 0; i < 5; ++i)
        s.push(str[i]);
    // s.top_ = 2; // Compile error - SAFE
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
```

- Spills data structure codes into application
- public data reveals the *internals*
- To switch container, application needs to change
- Application may corrupt the stack!

- No code in application, but init() to be called
- private data protects the *internals*
- Switching container is seamless
- Application cannot corrupt the stack



Program 13.02/03: Stack Initialization

Module 13

Partha Pratim
Das

Objectives &
Outline

Constructor
Parameterized
Overloaded

Destructor

Default
Constructor

Object
Lifetime

Automatic
Static
Dynamic

Summary

Using init()

```
#include <iostream>
using namespace std;
class Stack { private: // PROTECTED DATA
    char data_[10]; int top_;
public:
    void init() { top_ = -1; }
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { char str[10] = "ABCDE";
    Stack s;
    s.init(); // Clean initialization

    for (int i = 0; i < 5; ++i)
        s.push(str[i]);
    // s.top_ = 2; // Compile error - SAFE
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
```

- **init() serves no visible purpose – application may forget to call**
- If application misses to call `init()`, we have a corrupt stack

Using Constructor

```
#include <iostream>
using namespace std;
class Stack { private: // PROTECTED DATA
    char data_[10]; int top_;
public:
    Stack() : top_(-1) {} // Initialization
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call

    for (int i = 0; i < 5; ++i)
        s.push(str[i]);

    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
```

- **Can initialization be made a part of instantiation?**
- Yes. **Constructor** is implicitly called at instantiation as set by the compiler



Program 13.04/05: Stack: Constructor

Module 13

Partha Pratim
Das

Objectives &
Outline

Constructor
Parameterized
Overloaded

Destructor

Default
Constructor

Object
Lifetime

Automatic
Static
Dynamic

Summary

Automatic Array	Dynamic Array
<pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_; // Automatic public: Stack(); // Constructor // More Stack methods }; Stack::Stack(): // Initialization List top_(-1) { cout << "Stack::Stack() called" << endl; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call for (int i=0; i<5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; } ----- Stack::Stack() called EDCBA</pre>	<pre>#include <iostream> using namespace std; class Stack { private: char *data_; int top_; // Dynamic public: Stack(); // Constructor // More Stack methods }; Stack::Stack(): data_(new char[10]), // Init top_(-1) { // List cout << "Stack::Stack() called" << endl; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call for (int i=0; i<5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; } ----- Stack::Stack() called EDCBA</pre>

- `top_` initialized to `-1` in initialization list
- `data_[10]` initialized by default (automatic)
- `Stack::Stack()` called automatically when control passes `Stack s;` – Guarantees initialization



Constructor: Contrasting with Member Functions

Module 13

Partha Pratim
Das

Objectives &
Outline

Constructor
Parameterized
Overloaded

Destructor

Default
Constructor

Object
Lifetime

Automatic
Static
Dynamic

Summary

Constructor

- Is a member function with `this` pointer
- Name is same as the name of the class

```
class Stack { public:  
    Stack();  
};
```

- Has no return type
- `Stack::Stack();` // Not even void
- No return; hence has no return statement
- `Stack::Stack(): top_(-1)
{ } // Returns implicitly`

- Initializer list to initialize the data members

```
Stack::Stack(): // Initializer list  
    data_(new char[10]), // Init data_  
    top_(-1)           // Init top_  
{ }
```

- Implicit call by instantiation / operator new
- `Stack s;` // Calls `Stack::Stack()`
- May have any number of parameters
- Can be overloaded

Member Function

- Has implicit `this` pointer
 - Any name different from name of class
- ```
class Stack { public:
 int empty();
};
```

- Must have a return type
- `int Stack::empty();`
- Must have at least one return statement
- `int Stack::empty()  
{ return (top_ == -1); }`

```
void pop()
{ --top_; } // Implicit return
```

- Not applicable

- Explicit call by the object
- `s.empty();` // Calls `Stack::empty(&s)`
- May have any number of parameters
- Can be overloaded



# Program 13.06: Complex: Parameterized Constructor

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

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

class Complex { private: double re_, im_;
public:
 Complex(double re, double im): // Ctor w/ params
 re_(re), im_(im) // Params used to initialize
 {}
 double norm() { return sqrt(re_*re_ + im_*im_); }

 void print() {
 cout << "(" << re_ << "j" << im_ << ")" = ";
 cout << norm() << endl;
 }
};

int main() {
 Complex c(4.2, 5.3), // Complex::Complex(4.2, 5.3)
 d = { 1.6, 2.9 }; // Complex::Complex(1.6, 2.9)

 c.print();
 d.print();

 return 0;
}

|4.2+j5.3| = 6.7624
|1.6+j2.9| = 3.3121
```



# Program 13.07: Complex: Constructor with default parameters

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

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

class Complex { private: double re_, im_;
public:
 Complex(double re = 0.0, double im = 0.0) : // Ctor w/ default params
 re_(re), im_(im) // Params used to initialize
 {}
 double norm() { return sqrt(re_*re_ + im_*im_); }

 void print() { cout << "(" << re_ << "+" << im_ << "j" << ")" = " << norm() << endl; }
};

int main() {
 Complex c1(4.2, 5.3), // Complex::Complex(4.2, 5.3) -- both parameters explicit
 c2(4.2), // Complex::Complex(4.2, 0.0) -- second parameter default
 c3; // Complex::Complex(0.0, 0.0) -- both parameters default

 c1.print();
 c2.print();
 c3.print();

 return 0;
}

|4.2+j5.3| = 6.7624
|4.2+j0| = 4.2
|0+j0| = 0
```



# Program 13.08: Stack: Constructor with default parameters

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

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

class Stack { private: char *data_; int top_;
public:
 Stack(size_t = 10); // Size of data_ defaulted
 int empty() { return (top_ == -1); }
 void push(char x) { data_[++top_] = x; }
 void pop() { --top_; }
 char top() { return data_[top_]; }
};
Stack::Stack(size_t s) : data_(new char[s]), // Array of size s allocated
 top_(-1)
{ cout << "Stack created with max size = " << s << endl; }

int main() {
 char str[] = "ABCDE";
 Stack s(strlen(str)); // Create a stack large enough for the problem

 for (int i = 0; i<5; ++i) s.push(str[i]);
 while (!s.empty()) {
 cout << s.top(); s.pop();
 }
 return 0;
}

Stack created with max size = 5
EDCBA
```



# Program 13.09: Complex: Overloaded Constructors

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime  
Automatic  
Static  
Dynamic

Summary

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

class Complex { private: double re_, im_;
public:
 Complex(double re, double im): re_(re), im_(im) {} // Two parameters
 Complex(double re): re_(re), im_(0.0) {} // One parameter
 Complex(): re_(0.0), im_(0.0) {} // No parameter

 double norm() { return sqrt(re_*re_ + im_*im_); }

 void print() { cout << "(" << re_ << "+j" << im_ << ")" = " << norm() << endl; }
};

int main() {
 Complex c1(4.2, 5.3), // Complex::Complex(4.2, 5.3)
 c2(4.2), // Complex::Complex(4.2)
 c3; // Complex::Complex()

 c1.print();
 c2.print();
 c3.print();

 return 0;
}

|4.2+j5.3| = 6.7624
|4.2+j0| = 4.2
|0+j0| = 0
```



# Program 13.10/11: Stack: Destructor

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

## Automatic Array

```
#include <iostream> using namespace std;
class Stack { private:
 char *data_; int top_; // Dynamic
public: Stack(); // Constructor
 void de_init() { delete [] data_; }
 // More Stack methods
};
Stack::Stack(): data_(new char[10]), top_(-1)
{ cout << "Stack::Stack() called\n"; }

int main() { char str[10] = "ABCDE";
 Stack s; // Init by Stack::Stack() call

 // Reverse string using Stack
 de_init();
 return 0;
}

Stack::Stack() called
EDCBA
```

## Dynamic Array

```
#include <iostream> using namespace std;
class Stack { private:
 char *data_; int top_; // Dynamic
public: Stack(); // Constructor
 ~Stack(); // Destructor
 // More Stack methods
};
Stack::Stack(): data_(new char[10]), top_(-1)
{ cout << "Stack::Stack() called\n"; }
Stack::~Stack()
{
 cout << "\nStack::~Stack() called\n";
 delete data_;
}
int main() { char str[10] = "ABCDE";
 Stack s; // Init by Stack::Stack() call

 // Reverse string using Stack
 return 0;
} // De-Init by Stack::~Stack() call

Stack::Stack() called
EDCBA
Stack::~Stack() called
```

- Dynamically allocated data\_ leaks unless released before program loses scope of s
- Application may forget to call de\_init(); Also, when should de\_init() be called?

- Can de-initialization (release of data\_) be a part of scope rules?
- Yes. Destructor is implicitly called at end of scope



# Destructor: Contrasting with Member Functions

## Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

| Destructor                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Member Function                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>Is a member function with <code>this</code> pointer</li><li>Name is <code>~</code> followed by the name of the class<pre>class Stack { public:<br/>    ~Stack();<br/>};</pre></li><li>Has no return type<pre>Stack::~Stack(); // Not even void</pre></li><li>No return; hence has no return statement<pre>Stack::~Stack()<br/>{ } // Returns implicitly</pre></li><li>Implicitly called at end of scope or by operator <code>delete</code>. May be called explicitly by the object (rare)<pre>{<br/>    Stack s;<br/>    // ...<br/>} // Calls Stack::~Stack(&amp;s)</pre></li><li>No parameter is allowed - unique for the class</li><li>Cannot be overloaded</li></ul> | <ul style="list-style-type: none"><li>Has implicit <code>this</code> pointer</li><li>Any name different from name of class<pre>class Stack { public:<br/>    int empty();<br/>};</pre></li><li>Must have a return type<pre>int Stack::empty();</pre></li><li>Must have at least one return statement<pre>int Stack::empty()<br/>{ return (top_ == -1); }</pre></li><li>Explicit call by the object<pre>s.empty(); // Calls Stack::empty(&amp;s)</pre></li><li>May have any number of parameters</li><li>Can be overloaded</li></ul> |



# Default Constructor / Destructor

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

## ● Constructor

- A constructor with no parameter is called a *Default Constructor*
- If no constructor is provided by the user, the compiler supplies a *free default constructor*
- Compiler-provided (default) constructor, understandably, cannot initialize the object to proper values. It has no code in its body
- Default constructors (free or user-provided) are required to define arrays of objects

## ● Destructor

- If no destructor is provided by the user, the compiler supplies a *free default destructor*
- Compiler-provided (default) destructor has no code in its body



# Program 13.12: Complex: Default Constructor

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

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

class Complex {
private: double re_, im_; // private data
public:
 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "(" << re_ << "+j" << im_ << ")" = " << norm() << endl; }
 void set(double re, double im) { re_ = re; im_ = im; }
};

int main() {
 Complex c; // Free constructor from compiler
 // Initialization with garbage

 c.print(); // Print initial value - garbage
 c.set(4.2, 5.3); // Set proper components
 c.print(); // Print values set

 return 0;
} // Free destructor from compiler

|-9.25596e+061+j-9.25596e+061| = 1.30899e+062
|4.2+j5.3| = 6.7624
```

- User has provided no constructor / destructor
- Compiler provides default (free) constructor / destructor
- Compiler-provided constructor does nothing – components have garbage values
- Compiler-provided destructor does nothing



# Program 13.13: Complex: Default Constructor

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

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

class Complex { private: double re_, im_;
public:
 Complex(): re_(0.0), im_(0.0) // Default Ctor
 { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
 ~Complex() // Dtor
 { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }
 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "(" << re_ << "+j" << im_ << ")" = " << norm() << endl; }
 void set(double re, double im) { re_ = re; im_ = im; }
};

int main() {
 Complex c; // Default constructor -- user provided

 c.print(); // Print initial values
 c.set(4.2, 5.3); // Set components
 c.print(); // Print values set

 return 0;
} // Destuctor

Ctor: (0, 0)
|0+j0| = 0
|4.2+j5.3| = 6.7624
Dtor: (4.2, 5.3)
```

- User has provided a default constructor



# Object Lifetime: When is an Object ready? How long can it be used?

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

| Application                                                                                                                                                                                                         | Class Code                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>void MyFunc() // E1: Allocation of c on Stack {     ...     Complex c; // E2: Ctor called     ...     c.norm(); // E5: Use     ...     return; // E7: Dtor called } // E9: De-Allocation of c from Stack</pre> | <pre>Complex::Complex(double re = 0.0, // Ctor                   double im = 0.0):     re_(re), im_(im) // E3: Initialization { // E4: Object Lifetime STARTS     cout &lt;&lt; "Ctor:" &lt;&lt; endl; }  double Complex::norm() // E6 { return sqrt(re_*re_ + im_*im_); }  Complex::~Complex() // Dtor {     cout &lt;&lt; "Dtor:" &lt;&lt; endl; } // E8: Object Lifetime ENDS</pre> |

## Event Sequence and Object Lifetime

|    |                                                                                                |
|----|------------------------------------------------------------------------------------------------|
| E1 | MyFunc called. Stackframe allocated. c is a part of Stackframe                                 |
| E2 | Control to pass Complex c. Ctor Complex::Complex(&c) called with the address of c on the frame |
| E3 | Control on Initializer list of Complex::Complex(). Data members initialized (constructed)      |
| E4 | Object Lifetime STARTS for c. Control reaches the start of the body of Ctor. Ctor executes     |
| E5 | Control at c.norm(). Complex::norm(&c) called. Object is being used                            |
| E6 | Complex::norm() executes                                                                       |
| E7 | Control to pass return. Dtor Complex::~Complex(&c) called                                      |
| E8 | Dtor executes. Control reaches the end of the body of Dtor. Object Lifetime ENDS for c         |
| E9 | return executes. Stackframe including c de-allocated. Control returns to caller                |



# Object Lifetime

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

## ● Execution Stages

- Memory Allocation and Binding
- Constructor Call and Execution
- Object Use
- Destructor Call and Execution
- Memory De-Allocation and De-Binding

## ● Object Lifetime

- Starts with execution of Constructor Body
  - Must *follow* Memory Allocation
  - As soon as Initialization ends and control enters Constructor Body
- Ends with execution of Destructor Body
  - As soon as control leaves Destructor Body
  - Must *precede* Memory De-allocation
- For Objects of *Built-in / Pre-Defined Types*
  - No Explicit Constructor / Destructor
  - Lifetime spans from object definition to end of scope



# Program 13.14: Complex: Object Lifetime: Automatic

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor

Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_;
public:
 Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
 { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }

 ~Complex() // Dtor
 { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }

 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};

int main() {
 Complex c(4.2, 5.3), d(2.4); // Complex::Complex() called -- c, then d -- objects ready

 c.print(); // Using objects
 d.print();

 return 0;
} // Scope over, objects no more available.
 // Complex::~Complex() called -- d then c
 // Note the reverse order!

Ctor: (4.2, 5.3)
Ctor: (2.4, 0)
|4.2+j5.3| = 6.7624
|2.4+j0| = 2.4
Dtor: (2.4, 0)
Dtor: (4.2, 5.3)
```



# Program 13.15: Complex: Object Lifetime: Automatic: Array of Objects

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_;
public:
 Complex(double re = 0.0, double im = 0.0) : re_(re), im_(im) // Ctor
 { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
 ~Complex() // Dtor
 { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }

 void opComplex(double i) { re_ += i; im_ += i; } // Some operation with Complex

 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};

int main() {
 Complex c[3]; // Default ctor Complex::Complex() called thrice -- c[0], c[1], c[2]

 for (int i = 0; i < 3; ++i) { c[i].opComplex(i); c[i].print(); } // Use array
 return 0;
} // Scope over. Complex::~Complex() called thrice -- c[2], c[1], c[0] -- reverse order

Ctor: (0, 0)
Ctor: (0, 0)
Ctor: (0, 0)
|0+j0| = 0
|1+j1| = 1.41421
|2+j2| = 2.82843
Dtor: (2, 2)
Dtor: (1, 1)
Dtor: (0, 0)
```



# Program 13.16: Complex: Object Lifetime: Static

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

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

class Complex { private: double re_, im_;
public:
 Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
 { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
 ~Complex() // Dtor
 { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }
 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "(" << re_ << "+j" << im_ << ")" << " = " << norm() << endl; }
};

Complex c(4.2, 5.3); // Static (global) object
 // Constructed before main starts
 // Destrued after main ends

int main() {
 cout << "main() Starts" << endl;
 Complex d(2.4); // Ctor for d

 c.print(); // Use static object
 d.print(); // Use local object

 return 0;
} // Dtor for d

// Dtor for c
```

----- OUTPUT -----  
Ctor: (4.2, 5.3)  
main() Starts  
Ctor: (2.4, 0)  
|4.2+j5.3| = 6.7624  
|2.4+j0| = 2.4  
Dtor: (2.4, 0)  
Dtor: (4.2, 5.3)



# Program 13.17: Complex: Object Lifetime: Dynamic

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_;
public:
 Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
 { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
 ~Complex() // Dtor
 { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }
 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "(" << re_ << "+j" << im_ << ")" << " = " << norm() << endl; }
};

int main() { unsigned char buf[100]; // Buffer for placement of objects
 Complex* pc = new Complex(4.2, 5.3); // operator new: allocates memory, calls Ctor
 Complex* pd = new Complex[2]; // operator new []: allocates memory,
 // calls default Ctor twice
 Complex* pe = new (buf) Complex(2.6, 3.9); // operator placement new: only calls Ctor
 // no allocation of memory, uses buf
 // Use objects
 pc->print();
 pd[0].print(); pd[1].print();
 pe->print();

 // Release of objects - can be done in any order
 delete pc; // delete: calls Dtor, release memory
 delete [] pd; // delete[]: calls 2 Dtor's, release mem
 pe->~Complex(); // No delete: explicit call to Dtor
 // Use with extreme care
 return 0;
}
```

----- OUTPUT -----

Ctor: (4.2, 5.3)  
Ctor: (0, 0)  
Ctor: (0, 0)  
Ctor: (2.6, 3.9)  
|4.2+j5.3| = 6.7624  
|0+j0| = 0  
|0+j0| = 0  
|2.6+j3.9| = 4.68722  
Dtor: (4.2, 5.3)  
Dtor: (0, 0)  
Dtor: (0, 0)  
Dtor: (2.6, 3.9)



# Module Summary

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

- Objects are initialized by Constructors
- Constructors can be Parameterized and can be Overloaded
- Default Constructor does not take any parameter. It is necessary for defining arrays of objects
- Objects are cleaned-up by Destructors. Destructor for a class is unique
- Compiler provides *free* Default Constructor and Destructor, if not provided by the program
- Objects have a well-defined lifetime spanning from execution of the beginning of the body of a constructor to the execution till the end of the body of the destructor
- Memory for an object must be available before its construction and can be released only after its destruction



# Instructor and TAs

Module 13

Partha Pratim  
Das

Objectives &  
Outline

Constructor  
Parameterized  
Overloaded

Destructor

Default  
Constructor

Object  
Lifetime

Automatic  
Static  
Dynamic

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor  
Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

# Module 14: Programming in C++

## Copy Constructor and Copy Assignment Operator

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor  
Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

- More on Object Lifetime
- Understand Copy Construction
- Understand Copy Assignment Operator
- Understand Shallow and Deep Copy



# Module Outline

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

- Lifetime Examples
- Copy Constructor
  - Input Parameters
  - Call-by-Value
  - Initialization List
  - Copy with Pointers – Shallow and Deep Copy
- Copy Assignment Operator
  - Input Parameters
  - Return Type
  - Copy with Pointers – Shallow and Deep Copy
  - Self-copy



# Program 14.01: Order of Initialization – Order of Data Members

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor  
Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

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

int init_m1(int m) { // Func. to init m1_
 cout << "Init m1_: " << m << endl;
 return m;
}
int init_m2(int m) { // Func. to init m2_
 cout << "Init m2_: " << m << endl;
 return m;
}
class X {
 int m1_; // Initialize 1st
 int m2_; // Initialize 2nd
public:
 X(int m1, int m2) :
 m1_(init_m1(m1)), // Called 1st
 m2_(init_m2(m2)) // Called 2nd
 { cout << "Ctor: " << endl; }
 ~X() { cout << "Dtor: " << endl; }
};
int main() { X a(2, 3); return 0; }

Init m1_: 2
Init m2_: 3
Ctor:
Dtor:
```

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

int init_m1(int m) { // Func. to init m1_
 cout << "Init m1_: " << m << endl;
 return m;
}
int init_m2(int m) { // Func. to init m2_
 cout << "Init m2_: " << m << endl;
 return m;
}
class X {
 int m2_; // Order of data members swapped
 int m1_;
public:
 X(int m1, int m2) :
 m1_(init_m1(m1)), // Called 2nd
 m2_(init_m2(m2)) // Called 1st
 { cout << "Ctor: " << endl; }
 ~X() { cout << "Dtor: " << endl; }
};
int main() { X a(2, 3); return 0; }

Init m2_: 3
Init m1_: 2
Ctor:
Dtor:
```

- Order of initialization does not depend on the order in the initialization list. It depends on the order of data members in the definition



# Program 14.02/03: A Simple String Class

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members

Free Copy

Constructor

Copy

Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

| C Style                                                                                                                                                                                                                                                                                                                                                                                                                    | C++ Style                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#include &lt;iostream&gt; using namespace std;  struct String {     char *str_; // Container     size_t len_; // Length };  void print(const String&amp; s) {     cout &lt;&lt; s.str_ &lt;&lt; ":"          &lt;&lt; s.len_ &lt;&lt; endl; }  int main() {     String s;      // Init data members     s.str_ = strdup("Partha");     s.len_ = strlen(s.str_);      print(s);      return 0; } ----- Partha: 6</pre> | <pre>#include &lt;iostream&gt; using namespace std;  class String {     char *str_; // Container     size_t len_; // Length public:     String(char *s) : str_(strdup(s)),    // Uses malloc()                     len_(strlen(str_))     { cout &lt;&lt; "ctor: "; print(); }     ~String() {         cout &lt;&lt; "dtor: "; print();         free(str_); // To match malloc() in strdup()     }     void print() { cout &lt;&lt; "(" &lt;&lt; str_ &lt;&lt; ":"                      &lt;&lt; len_ &lt;&lt; ")" &lt;&lt; endl; }     size_t len() { return len_; } }; int main() {     String s = "Partha"; // Ctor called     s.print();     return 0; } ----- ctor: (Partha: 6) (Partha: 6) dtor: (Partha: 6)</pre> |

- Note the order of initialization between str\_ and len\_. What if we swap them?



# Program 14.04: A Simple String Class – Fails for wrong order of data members

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

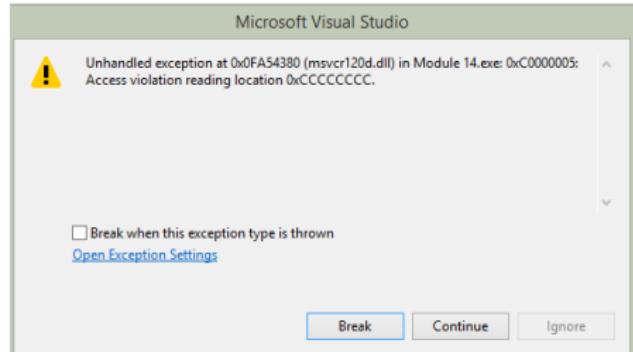
Summary

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

class String {
 size_t len_; // Swapped members cause program crash (unhandled exception)
 char *str_;
public:
 String(char *s) : str_(strdup(s)), len_(strlen(str_)) { cout << "ctor: "; print(); }
 ~String() { cout << "dtor: "; print(); free(str_); }
 void print() { cout << "(" << str_ << ":" << len_ << ")" << endl; }
};

int main() {
 String s = "Partha";
 s.print();
 return 0;
}
```

- **len\_ precedes str\_ in list of data members**
- **len\_(strlen(str\_)) is executed before str\_(strdup(s))**
- **When strlen(str\_) is called str\_ is still uninitialized**
- **Causes the program to crash as shown in the message box**





# Program 14.05: A Simple Date Class

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date

Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members

Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

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

char monthNames[] [4] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
char dayNames[] [10] = { "Monday", "Tuesday", "Wednesday", "Thursday",
 "Friday", "Saturday", "Sunday" };

class Date {
 enum Month { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
 enum Day { Mon, Tue, Wed, Thr, Fri, Sat, Sun };
 typedef unsigned int UINT;
 UINT date_; Month month_; UINT year_;
public:
 Date(UINT d, UINT m, UINT y) : date_(d), month_((Month)m), year_(y)
 { cout << "ctor: "; print(); }
 ~Date() { cout << "dtor: "; print(); }
 void print() { cout << date_ << "/" << monthNames[month_ - 1] << "/" << year_ << endl; }
 bool validDate() { /* Check validity */ return true; } // Not implemented
 Day day() { /* Compute day from date using time.h */ return Mon; } // Not implemented
};

int main() {
 Date d(30, 7, 1961);
 d.print();
 return 0;
}

ctor: 30/Jul/1961
30/Jul/1961
dtor: 30/Jul/1961
```



# Program 14.06: Point and Rect Classes: Lifetime of Data Members or Embedded Objects

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value

Signature

Data members

Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

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

class Point {
 int x_;
 int y_;
public:
 Point(int x, int y):
 x_(x), y_(y)
 { cout << "Point ctor: ";
 print(); cout << endl; }
 ~Point() { cout << "Point dtor: ";
 print(); cout << endl; }
 void print()
 { cout << "(" << x_ << ", "
 << y_ << ")"; }
};

int main()
{
 Rect r(0, 2, 5, 7);

 cout << endl; r.print(); cout << endl;
 cout << endl;
 return 0;
}
```

- Attempt is to construct a Rect object
- That, in turn, needs constructions of Point data members (or embedded objects) – TL\_ and BR\_ respectively
- Destruction, initiated at the end of scope of destructor's body, naturally follows a reverse order

```
class Rect {
 Point TL_;
 Point BR_;
public:
 Rect(int tlx, int tly, int brx, int bry):
 TL_(tlx, tly), BR_(brx, bry)
 { cout << "Rect ctor: ";
 print(); cout << endl; }
 ~Rect() { cout << "Rect dtor: ";
 print(); cout << endl; }
 void print()
 { cout << "["; TL_.print(); cout
 << " "; BR_.print(); cout << "]"; }
};
```

```

Point ctor: (0, 2)
Point ctor: (5, 7)
Rect ctor: [(0, 2) (5, 7)]
[(0, 2) (5, 7)]
```

```
Rect dtor: [(0, 2) (5, 7)]
Point dtor: (5, 7)
Point dtor: (0, 2)
```



# Program 14.07: Name & Address Classes

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members  
Free Copy

Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

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

#include "String.h"
#include "Date.h"

class Name { String firstName_, lastName_;
public:
 Name(const char* fn, const char* ln) : firstName_(fn), lastName_(ln)
 { cout << "Name ctor: "; print(); cout << endl; }
 ~Name() { cout << "Name dtor: "; print(); cout << endl; }
 void print()
 { firstName_.print(); cout << " "; lastName_.print(); }
};

class Address {
 unsigned int houseNo_;
 String street_, city_, pin_;
public:
 Address(unsigned int hn, const char* sn, const char* cn, const char* pin) :
 houseNo_(hn), street_(sn), city_(cn), pin_(pin)
 { cout << "Address ctor: "; print(); cout << endl; }
 ~Address() { cout << "Address dtor: "; print(); cout << endl; }
 void print()
 { cout << houseNo_ << " ";
 street_.print(); cout << " ";
 city_.print(); cout << " ";
 pin_.print();
 }
};
```



# Program 14.07: CreditCard Class

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

```
class CreditCard { typedef unsigned int UINT;
 char cardNumber_[17]; // 16-digit (character) card number as C-string
 Name holder_;
 Address addr_;
 Date issueDate_, expiryDate_;
 UINT cvv_;

public:
 CreditCard(const char* cNumber, const char* fn, const char* ln,
 unsigned int hn, const char* sn, const char* cn, const char* pin,
 UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear, UINT cvv) :
 holder_(fn, ln), addr_(hn, sn, cn, pin),
 issueDate_(1, issueMonth, issueYear),
 expiryDate_(1, expiryMonth, expiryYear), cvv_(cvv)
 { strcpy(cardNumber_, cNumber); cout << "CC ctor: "; print(); cout << endl; }
 ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; }
 void print() {
 cout << cardNumber_ << " ";
 holder_.print(); cout << " ";
 addr_.print(); cout << " ";
 issueDate_.print(); cout << " ";
 expiryDate_.print(); cout << " ";
 cout << cvv_;
 }
};

int main() {
 CreditCard cc("5321711934640027", "Sharlock", "Holmes",
 221, "Baker Street", "London", "NW1 6XE", 7, 2014, 12, 2016, 811);
 cout << endl; cc.print(); cout << endl << endl;
 return 0;
}
```



# Program 14.07: CreditCard Class: Lifetime Chart

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

## Construction of Objects

```
String: Sherlock
String: Holmes
Name: Sherlock Holmes
String: Baker Street
String: London
String: NW1 6XE
Address: 221 Baker Street London NW1 6XE
Date: 1/Jul/2014
Date: 1/Dec/2016
```

```
CC: 5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811
```

## Use of Object

```
5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811
```

## Destruction of Objects

```
~CC: 5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811
~Date: 1/Dec/2016
~Date: 1/Jul/2014
~Address: 221 Baker Street London NW1 6XE
~String: NW1 6XE
~String: London
~String: Baker Street
~Name: Sherlock Holmes
~String: Holmes
~String: Sherlock
```



# Copy Constructor

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

- We know:

```
Complex c1 = {4.2, 5.9}; // or c1(4.2, 5.9)
invokes
```

```
Constructor Complex::Complex(double, double);
```

- Which constructor is invoked for?

```
Complex c2(c1);
```

Or for?

```
Complex c2 = c1;
```

- It is the **Copy Constructor** that takes an object of the same type and constructs a copy:

```
Complex::Complex(const Complex &);
```



# Program 14.08: Complex: Copy Constructor

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members

Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_;
public:
 Complex(double re, double im) : re_(re), im_(im) // Constructor
 { cout << "Complex ctor: "; print(); }
 Complex(const Complex& c) : re_(c.re_), im_(c.im_) // Copy Constructor
 { cout << "Complex copy ctor: "; print(); }
 ~Complex() { cout << "Complex dtor: "; print(); }
 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "(" << re_ << "+" << im_ << ")" = " << norm() << endl; }
};
int main() {
 Complex c1(4.2, 5.3), // Constructor - Complex(double, double)
 c2(c1), // Copy Constructor - Complex(const Complex&)
 c3 = c2; // Copy Constructor - Complex(const Complex&)

 c1.print(); c2.print(); c3.print();
 return 0;
}

Complex ctor: (4.2+5.3) = 6.7624 // Ctor: c1
Complex copy ctor: (4.2+5.3) = 6.7624 // CCtor: c2 of c1
Complex copy ctor: (4.2+5.3) = 6.7624 // CCtor: c3 of c2
(4.2+5.3) = 6.7624 // c1
(4.2+5.3) = 6.7624 // c2
(4.2+5.3) = 6.7624 // c3
Complex dtor: (4.2+5.3) = 6.7624 // Dtor: c3
Complex dtor: (4.2+5.3) = 6.7624 // Dtor: c2
Complex dtor: (4.2+5.3) = 6.7624 // Dtor: c1
```



# Why do we need Copy Constructor?

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

- Consider the **function call mechanisms** in C++:
  - *Call-by-reference*: Set a reference to the actual parameter as a formal parameter. Both the formal parameter and the actual parameter share the same location (object)
  - *Return-by-reference*: Set a reference to the computed value as a return value. Both the computed value and the return value share the same location (object)
  - *Call-by-value*: Make a copy (clone) of the actual parameter as a formal parameter. This needs a **Copy Constructor**
  - *Return-by-value*: Make a copy (clone) of the computed value as a return value. This needs a **Copy Constructor**
- **Copy Constructor** is needed for **initializing the data members** of a UDT from an existing value



# Program 14.09: Complex: Call by value

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members

Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

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

class Complex { double re_, im_;
public:
 Complex(double re, double im) : re_(re), im_(im) // Constructor
 { cout << "ctor: "; print(); }
 Complex(const Complex& c) : re_(c.re_), im_(c.im_) // Copy Constructor
 { cout << "copy ctor: "; print(); }
 ~Complex() { cout << "dtor: "; print(); }
 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "(" << re_ << "+" << im_ << ")" << " = " << norm() << endl; }
};

void Display(Complex c_param) { // Call by value
 cout << "Display: "; c_param.print();
}

int main() {
 Complex c(4.2, 5.3); // Constructor - Complex(double, double)

 Display(c); // Copy Constructor called to copy c to c_param

 return 0;
}

ctor: (4.2+5.3) = 6.7624 // Ctor of c in main()
copy ctor: (4.2+5.3) = 6.7624 // Ctor c_param as copy of c, call Display()
Display: (4.2+5.3) = 6.7624 // c_param
dtor: (4.2+5.3) = 6.7624 // Dtor c_param on exit from Display()
dtor: (4.2+5.3) = 6.7624 // Dtor of c on exit from main()
```



# Signature of Copy Constructors

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor  
Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature  
Summary

- Signature of a *Copy Constructor* can be one of:

```
MyClass(const MyClass& other); // Common
 // Source cannot be changed
MyClass(MyClass& other); // Occasional
 // Source needs to change
MyClass(volatile const MyClass& other); // Rare
MyClass(volatile MyClass& other); // Rare
```

- None of the following are copy constructors, though they can copy:

```
MyClass(MyClass* other);
MyClass(const MyClass* other);
```

- Why the parameter to a copy constructor must be passed as Call-by-Reference?

```
MyClass(MyClass other);
```

The above is an infinite loop as the call to copy constructor itself needs to make copy for the Call-by-Value mechanism.



# Program 14.10: Point and Rect Classes: Default, Copy and Overloaded Constructors

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members

Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

```
#include <iostream>
using namespace std;
class Point { int x_; int y_; public:
 Point(int x, int y) : x_(x), y_(y) // Constructor (Ctor)
 { cout << "Point ctor: "; print(); cout << endl; }
 Point() : x_(0), y_(0) // Default Constructor (DCtor)
 { cout << "Point ctor: "; print(); cout << endl; }
 Point(const Point& p) : x_(p.x_), y_(p.y_) // Copy Constructor (CCtor)
 { cout << "Point cctor: "; print(); cout << endl; }
 ~Point() { cout << "Point dtor: "; print(); cout << endl; } // Destructor (Dtor)
 void print() { cout << "(" << x_ << ", " << y_ << ")"; }
};

class Rect { Point TL_; Point BR_; public:
 Rect(int tlx, int tly, int brx, int bry):
 TL_(tlx, tly), BR_(brx, bry) // Ctor - Uses Ctor for Point
 { cout << "Rect ctor: "; print(); cout << endl; }
 Rect(const Point& p_tl, const Point& p_br): TL_(p_tl), BR_(p_br) // Ctor
 { cout << "Rect ctor: "; print(); cout << endl; } // Uses CCtor for Point
 Rect(const Point& p_tl, int brx, int bry): TL_(p_tl), BR_(brx, bry) // Ctor
 { cout << "Rect ctor: "; print(); cout << endl; } // CCtor for Point
 Rect() { cout << "Rect ctor: "; print(); cout << endl; } // Default Ctor
 Rect(const Rect& r): TL_(r.TL_), BR_(r.BR_) // Copy Ctor
 { cout << "Rect cctor: "; print(); cout << endl; }
 ~Rect() { cout << "Rect dtor: "; print(); cout << endl; } // Dtor
 void print() { cout << "["; TL_.print(); cout << " "; BR_.print(); cout << "]"; }
};

● When parameter (tlx, tly) is set to TL_ by TL_(tlx, tly): parameterized Ctor of Point is involved
● When parameter p_tl is set to TL_ by TL_(p_tl): CCtor of Point is involved
● When TL_ is set by default in DCtor of Rect: DCtor of Point is involved
● When member r.TL_ is set to TL_ by TL_(r.TL_) in CCtor of Rect: CCtor of Point is involved
```



# Program 14.10: Rect Class: Trace of Object Lifetimes

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

| Code                                                                                                                                                                                                                                                                                 | Output                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | Lifetime                                                                                                                                                                                                                                                                                                                                                            | Remarks                                                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>int main() {     Rect r1(0, 2, 5, 7);     //Rect(int, int, int, int)      Rect r2(Point(3, 5),             Point(6, 9));     //Rect(Point&amp;, Point&amp;)      Rect r3(Point(2, 2), 6, 4);     //Rect(Point&amp;, int, int)      Rect r4;     //Rect()      return 0; }</pre> | <pre>Point ctor: (0, 2) Point ctor: (5, 7) Rect ctor: [(0, 2) (5, 7)]  Point ctor: (6, 9) Point ctor: (3, 5) Point cctor: (3, 5) Point cctor: (6, 9) Rect ctor: [(3, 5) (6, 9)] Point dtor: (3, 5) Point dtor: (6, 9)  Point ctor: (2, 2) Point cctor: (2, 2) Point ctor: (6, 4) Rect ctor: [(2, 2) (6, 4)] Point dtor: (2, 2)  Point ctor: (0, 0) Point ctor: (0, 0) Rect ctor: [(0, 0) (0, 0)]  Rect dtor: [(0, 0) (0, 0)] Point dtor: (0, 0) Point dtor: (0, 0)  Rect dtor: [(2, 2) (6, 4)] Point dtor: (6, 4) Point dtor: (2, 2)  Rect dtor: [(3, 5) (6, 9)] Point dtor: (6, 9) Point dtor: (3, 5)  Rect dtor: [(0, 2) (5, 7)] Point dtor: (5, 7) Point dtor: (0, 2)</pre> | <pre>Point r1.TL_ Point r1.BR_ Rect r1  Point t1 Point t2 r2.TL_ = t2 r2.BR_ = t1 Rect r2 ~Point t2 ~Point t1  Point t3 r3.TL_ = t3 Point r3.BR_ Rect r3 ~Point t3  Point r4.TL_ Point r4.BR_ Rect r4  ~Rect r4 ~Point r4.BR_ ~Point r4.TL_  ~Rect r3 ~Point r3.BR_ ~Point r3.TL_  ~Rect r2 ~Point r2.BR_ ~Point r2.TL_  ~Rect r1 ~Point r1.BR_ ~Point r1.TL_</pre> | <p>Second parameter<br/>First parameter<br/>Copy to r2.TL_<br/>Copy to r2.BR_</p> <p>First parameter<br/>Second parameter<br/>First parameter<br/>Copy to r3.TL_</p> <p>First parameter</p> |



# Free Copy Constructor

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members  
**Free Copy  
Constructor**

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

- If no copy constructor is provided by the user, the compiler supplies a *free* copy constructor
- Compiler-provided copy constructor, understandably, cannot initialize the object to proper values. It has no code in its body. It performs a *bit-copy*



# Program 14.09: Complex: Free Copy Constructor

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

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

class Complex { double re_, im_; public:
 Complex(double re, double im) : re_(re), im_(im) // Constructor
 { cout << "ctor: "; print(); }
 //Complex(const Complex& c) : re_(c.re_), im_(c.im_) // Copy Constructor
 //{ cout << "copy ctor: "; print(); }
 ~Complex() { cout << "dtor: "; print(); }
 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "(" << re_ << "+" << im_ << ")" = " << norm() << endl; }
};

void Display(Complex c_param) { cout << "Display: "; c_param.print(); }

int main()
{
 Complex c(4.2, 5.3); // Constructor - Complex(double, double)
 Display(c); // Free Copy Constructor called to copy c to c_param

 return 0;
}
```

|                                                                                                                                                                                    |                                                                                                                                                                           |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>User-defined CCtor</b><br>ctor:  4.2+j5.3  = 6.7624<br>copy ctor:  4.2+j5.3  = 6.7624<br>Display:  4.2+j5.3  = 6.7624<br>dtor:  4.2+j5.3  = 6.7624<br>dtor:  4.2+j5.3  = 6.7624 | <b>Free CCtor</b><br>ctor:  4.2+j5.3  = 6.7624<br>\\ No message from free CCtor<br>Display:  4.2+j5.3  = 6.7624<br>dtor:  4.2+j5.3  = 6.7624<br>dtor:  4.2+j5.3  = 6.7624 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- User has provided no copy constructor
- Compiler provides free copy constructor
- Compiler-provided copy constructor performs bit-copy - hence there is no message
- Correct in this case as members are of built-in type



# Program 14.11: String: User-defined Copy Constructor

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members

Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;
class String { public: char *str_; size_t len_;
 String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // ctor
 String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // cctor
 ~String() { free(str_); } // dtor
 void print() { cout << "(" << str_ << ":" << len_ << ")" << endl; }
};
void strToUpper(String a) { // Make the string uppercase
 for (int i = 0; i < a.len_; ++i) a.str_[i] = toupper(a.str_[i]);
 cout << "strToUpper: "; a.print();
}
int main() {
 String s = "Partha";
 s.print();
 strToUpper(s);
 s.print();
 return 0;
}

(Partha: 6)
strToUpper: (PARTHA: 6)
(Partha: 6)
```

- User has provided copy constructor. So Compiler does not provide free copy constructor
- When actual parameter s is copied to formal parameter a, space is allocated for a.str\_ and then it is copied from s.str\_. On exit from strToUpper, a is destructed and a.str\_ is deallocated. But in main, s remains intact and access to s.str\_ is valid.
- **Deep Copy:** While copying the object, the pointed object is copied in a fresh allocation. This is safe



# Program 14.11: String: Free Copy Constructor

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

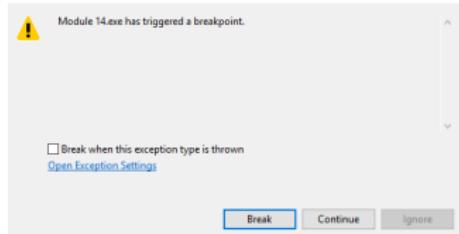
Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

```
#include <iostream>
using namespace std;
class String { public: char *str_; size_t len_;
 String(char *s) : str_(strdup(s)), len_(strlen(str_)) {} // ctor
 //String(const String& s) : str_(strdup(s.str_)), len_(s.len_) {} // cctor
 ~String() { free(str_); } // dtor
 void print() { cout << "(" << str_ << ":" << len_ << ")" << endl; }
 void strToUpper(String a) { // Make the string uppercase
 for (int i = 0; i < a.len_; ++i) a.str_[i] = toupper(a.str_[i]);
 cout << "strToUpper: "; a.print(); } Microsoft Visual Studio
int main() {
 String s = "Partha";
 s.print();
 strToUpper(s);
 s.print();

 return 0;
}
```



User-defined CCtor  
(Partha: 6)  
strToUpper: (PARTHA: 6)  
(Partha: 6)

Free CCtor  
(Partha: 6)  
strToUpper: (PARTHA: 6)  
(?????????????????????????: 6)

- User has provided no copy constructor. Compiler provides free copy constructor
- Free copy constructor performs bit-copy - hence no allocation is done for str\_ when actual parameter s is copied to formal parameter a. s.str\_ is merely copied to a.str\_ and both continue to point to the same memory. On exit from strToUpper, a is destructed and a.str\_ is deallocated. Hence in main access to s.str\_ is corrupted. Program crashes
- Shallow Copy: With bit-copy, only the pointer is copied - not the pointed object. This may be risky



# Program 14.12: Complex: Copy Assignment

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
 Complex(double re, double im) : re_(re), im_(im) { cout << "ctor: "; print(); }
 Complex(const Complex& c) : re_(c.re_), im_(c.im_) { cout << "cctor: "; print(); }
 ~Complex() { cout << "dtor: "; print(); }
 Complex& operator=(const Complex& c) // Copy Assignment Operator
 { re_ = c.re_; im_ = c.im_; cout << "copy: "; print(); return *this; }
 double norm() { return sqrt(re_*re_ + im_*im_); }
 void print() { cout << "(" << re_ << "j" << im_ << ")" << " | " << norm() << endl; }
};
int main() {
 Complex c1(4.2, 5.3), c2(7.9, 8.5); // Constructor - Complex(double, double)
 Complex c3(c2); // Constructor - Complex(const Complex& c)

 c1.print(); c2.print(); c3.print();
 c2 = c1; c2.print(); // Copy Assignment Operator
 c1 = c2 = c3; c1.print(); c2.print(); c3.print(); // Copy Assignment Chain
 return 0;
}
ctor: |4.2+j5.3| = 6.7624 // c1 - ctor copy: |7.9+j8.5| = 11.6043 // c2 <- c3
ctor: |7.9+j8.5| = 11.6043 // c2 - ctor copy: |7.9+j8.5| = 11.6043 // c1 <- c2
cctor: |7.9+j8.5| = 11.6043 // c3 - ctor |7.9+j8.5| = 11.6043 // c1
|4.2+j5.3| = 6.7624 // c1 |7.9+j8.5| = 11.6043 // c2
|7.9+j8.5| = 11.6043 // c2 |7.9+j8.5| = 11.6043 // c3
|7.9+j8.5| = 11.6043 // c3 dtor: |7.9+j8.5| = 11.6043 // c3 - dtor
copy: |4.2+j5.3| = 6.7624 // c2 <- c1 dtor: |7.9+j8.5| = 11.6043 // c2 - dtor
|4.2+j5.3| = 6.7624 // c2 dtor: |7.9+j8.5| = 11.6043 // c1 - dtor
```

- Copy assignment operator should return the object to make chain assignments possible



# Program 14.13: String: Copy Assignment

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

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

class String { public: char *str_; size_t len_;
 String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // ctor
 String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // cctor
 ~String() { free(str_); } // dtor
 String& operator=(const String& s) {
 free(str_); // Release existing memory
 str_ = strdup(s.str_); // Perform deep copy
 len_ = s.len_;
 return *this; // Return object for chain assignment
 }
 void print() { cout << "(" << str_ << ":" << len_ << ")" << endl; }
};

int main() { String s1 = "Football", s2 = "Cricket";
 s1.print(); s2.print();
 s2 = s1; s2.print();
 return 0;
}

(Football: 8)
(Cricket: 7)
(Football: 8)
```

- In copy assignment operator, `str_ = s.str_` should not be done for two reasons:
  - 1) Resource held by `str_` will leak
  - 2) Shallow copy will result with its related issues
- What happens if a self-copy `s1 = s1` is done?



# Program 14.13: String: Self Copy

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature

Data members  
Free Copy

Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

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

class String { public: char *str_; size_t len_;
 String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // ctor
 String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // cctor
 ~String() { free(str_); } // dtor
 String& operator=(const String& s) {
 free(str_); // Release existing memory
 str_ = strdup(s.str_); // Perform deep copy
 len_ = s.len_;
 return *this; // Return object for chain assignment
 }
 void print() { cout << "(" << str_ << ":" << len_ << ")" << endl; }
};

int main() { String s1 = "Football", s2 = "Cricket";
 s1.print(); s2.print();
 s1 = s1; s1.print();
 return 0;
}

(Football: 8)
(Cricket: 7)
(???????: 8) // Garbage is printed
```

- For self-copy `str_` and `s.str_` are the same pointers
- Hence, `free(str_)` first releases the memory, and then `strdup(s.str_)` tries to copy from released memory
- This may crash or produce garbage values
- Self-copy must be detected and protected



# Program 14.14: String: Self Copy – Safe

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect

Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

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

class String { public: char *str_; size_t len_;
 String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // ctor
 String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // cctor
 ~String() { free(str_); } // dtor
 String& operator=(const String& s) {
 if (this != &s) {
 free(str_);
 str_ = strdup(s.str_);
 len_ = s.len_;
 }
 return *this;
 }
 void print() { cout << "(" << str_ << ":" << len_ << ")" << endl; }
};

int main() { String s1 = "Football", s2 = "Cricket";
 s1.print(); s2.print();
 s1 = s1; s1.print();
 return 0;
}

(Football: 8)
(Cricket: 7)
(Football: 8)
```

- Check for self-copy (`this != &s`)
- In case of self-copy, do nothing



# Signature and Body of Copy Assignment Operator

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members

Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

- For class MyClass, typical copy assignment operator will be:

```
MyClass& operator=(const MyClass& s) {
 if (this != &s) {
 // Release resources held by *this
 // Copy members of s to members of *this
 }
 return *this;
}
```

- Signature of a *Copy Assignment Operator* can be one of:

```
MyClass& operator=(const MyClass& rhs); // Common
 // No change in Source
MyClass& operator=(MyClass& rhs); // Occasional
 // Change in Source
```

- The following *Copy Assignment Operators* are occasionally used:

```
MyClass& operator=(MyClass rhs);
const MyClass& operator=(const MyClass& rhs);
const MyClass& operator=(MyClass& rhs);
const MyClass& operator=(MyClass rhs);
MyClass operator=(const MyClass& rhs);
MyClass operator=(MyClass& rhs);
MyClass operator=(MyClass rhs);
```



# Module Summary

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value  
Signature  
Data members  
Free Copy  
Constructor

Copy  
Assignment  
Operator  
Copy Pointer  
Self-Copy  
Signature

Summary

## ● **Copy Constructors**

- A new object is created
- The new object is initialized with the value of data members of another object

## ● **Copy Assignment Operator**

- An object is already existing (and initialized)
- The members of the existing object are replaced by values of data members of another object

## ● **Deep and Shallow Copy for Pointer Members**

- Deep copy allocates new space for the contents and copies the pointed data
- Shallow copy merely copies the pointer value – hence, the new copy and the original pointer continue to point to the same data



# Instructor and TAs

Module 14

Partha Pratim  
Das

Objectives &  
Outline

Lifetime  
Examples

String  
Date  
Rect  
Name & Address  
CreditCard

Copy  
Constructor

Call by value

Signature

Data members

Free Copy  
Constructor

Copy  
Assignment  
Operator

Copy Pointer  
Self-Copy  
Signature

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

# Module 15: Programming in C++

Const-ness

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- Understand const-ness of objects in C++
- Understand the use of const-ness in class design



# Module Outline

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- Constant Objects
- Constant Member methods
- Constant Data members
  - Credit Card Example
- **mutable** Data members - logical and bitwise const-ness



# Constant Objects

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- Like objects of built-in type, objects of user-defined types can also be made constant
- If an object is constant, none of its data members can be changed
- The type of the this pointer of a constant object of class, say, MyClass is:

```
// Const Pointer to Const Object
const MyClass * const this;
```

instead of

```
// Const Pointer to non-Const Object
MyClass * const this;
```

as for a non-constant object of the same class

- A constant objects cannot invoke normal methods of the class lest these methods change the object
- Let us take an example



# Program 15.01: Example: Non-Constant Objects

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

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

class MyClass {
 int myPriMember_;
public:
 int myPubMember_;
 MyClass(int mPri, int mPub) : myPriMember_(mPri), myPubMember_(mPub) {}
 int getMember() { return myPriMember_; }
 void setMember(int i) { myPriMember_ = i; }
 void print() { cout << myPriMember_ << ", " << myPubMember_ << endl; }
};

int main() {
 MyClass myObj(0, 1); // Non-constant object

 cout << myObj.getMember() << endl;
 myObj.setMember(2);
 myObj.myPubMember_ = 3;
 myObj.print();

 return 0;
}

```

0  
2, 3

- It is okay to invoke methods for non-constant object **myObj**
- It is okay to make changes in non-constant object **myObj** by method (**setMember()**)
- It is okay to make changes in non-constant object **myObj** directly (**myPubMember\_**)



# Program 15.02: Example: Constant Objects

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

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

class MyClass {
 int myPriMember_;
public:
 int myPubMember_;
 MyClass(int mPri, int mPub) : myPriMember_(mPri), myPubMember_(mPub) {}
 int getMember() { return myPriMember_; }
 void setMember(int i) { myPriMember_ = i; }
 void print() { cout << myPriMember_ << ", " << myPubMember_ << endl; }
};

int main() {
 const MyClass myConstObj(5, 6); // Constant object

 cout << myConstObj.getMember() << endl; // Error 1
 myConstObj.setMember(7); // Error 2
 myConstObj.myPubMember_ = 8; // Error 3
 myConstObj.print(); // Error 4

 return 0;
}
```

- It is not allowed to invoke methods or make changes in constant object **myConstObj**
- Error (1, 2 & 4) on method invocation typically is:  
**cannot convert 'this' pointer from 'const MyClass' to 'MyClass &'**
- Error (3) on member update typically is:  
**'myConstObj' : you cannot assign to a variable that is const**
- With **const**, this pointer is **const MyClass \* const** while the methods expects **MyClass \* const**
- Consequently, we cannot print the data member of the class (even without changing it)
- Fortunately, constant objects can invoke (select) methods if they are **constant member functions**



# Constant Member Function

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- To declare a constant member function, we use the keyword `const` between the function header and the body. Like:

```
void print() const { cout << myMember_ << endl; }
```

- A constant member function expects a `this` pointer as:

```
const MyClass * const this;
```

and hence can be invoked by constant objects

- In a constant member function no data member can be changed. Hence,

```
void setMember(int i) const
{ myMember_ = i; } // data member cannot be changed
```

gives an error

- Interesting, *non-constant objects* can invoke *constant member functions* (by casting – we discuss later) and, of course, *non-constant member functions*
- Constant objects*, however, can **only** invoke *constant member functions*
- All member functions that do not need to change an object must be declared as constant member functions**



# Program 15.03: Example: Constant Member Functions

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

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

class MyClass {
 int myPriMember_;
public:
 int myPubMember_;
 MyClass(int mPri, int mPub) : myPriMember_(mPri), myPubMember_(mPub) {}
 int getMember() const { return myPriMember_; }
 void setMember(int i) { myPriMember_ = i; }
 void print() const { cout << myPriMember_ << ", " << myPubMember_ << endl; }
};

int main() {
 MyClass myObj(0, 1); // Non-constant object
 const MyClass myConstObj(5, 6); // Constant object

 cout << myObj.getMember() << endl;
 myObj.setMember(2);
 myObj.myPubMember_ = 3;
 myObj.print();

 cout << myConstObj.getMember() << endl;
 //myConstObj.setMember(7);
 //myConstObj.myPubMember_ = 8;
 myConstObj.print();
 return 0;
}
```

| Output |
|--------|
| 0      |
| 2, 3   |
| 5      |
| 5, 6   |

- Now **myConstObj** can invoke **getMember()** and **print()**, but cannot invoke **setMember()**
- Naturally **myConstObj** cannot update **myPubMember\_**
- myObj** can invoke all of **getMember()**, **print()**, and **setMember()**



# Constant Data members

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- Often we need part of an object, that is, one or more data members to be constant (non-changeable after construction) while the rest of the data members should be changeable. For example:
  - For an Employee: employee ID and DoB should be non-changeable while designation, address, salary etc. should be changeable
  - For a Student: roll number and DoB should be non-changeable while year of study, address, gpa etc. should be changeable
  - For a Credit Card: card number and name of holder should be non-changeable while date of issue, date of expiry, address, cvv number gpa etc. should be changeable
- Do this by making the non-changeable data members as constant
- To make a data member constant, we need to put the `const` keyword before the declaration of the member in the class
- **A constant data member cannot be changed even in a non-constant object**
- **A constant data member must be initialized on the initialization list**



# Program 15.04: Example: Constant Data Member

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#include <iostream>
using namespace std;
class MyClass {
 const int cPriMem_;
 int priMem_;
public:
 const int cPubMem_;
 int pubMem_;
 MyClass(int cPri, int ncPri, int cPub, int ncPub) :
 cPriMem_(cPri), priMem_(ncPri), cPubMem_(cPub), pubMem_(ncPub) {}
 int getcPri() { return cPriMem_; }
 void setcPri(int i) { cPriMem_ = i; } // Error 1: Assignment to constant data member
 int getPri() { return priMem_; }
 void setPri(int i) { priMem_ = i; }
};
int main() {
 MyClass myObj(1, 2, 3, 4);

 cout << myObj.getcPri() << endl; myObj.setcPri(6);
 cout << myObj.getPri() << endl; myObj.setPri(6);

 cout << myObj.cPubMem_ << endl;
 myObj.cPubMem_ = 3; // Error 2: Assignment to constant data member

 cout << myObj.pubMem_ << endl; myObj.pubMem_ = 3;
 return 0;
}
```

- It is not allowed to make changes to constant data members in **myObj**
- Error 1:l-value specifies **const** object
- Error 2:'**myObj**' : you cannot assign to a variable that is **const**



# Credit Card Example

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

We now illustrate constant data members with a complete example of CreditCard class with the following supporting classes:

- String class
- Date class
- Name class
- Address class



# Program 15.05: String Class: In header file with copy

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __STRING_H
#define __STRING_H
#include <iostream>
#include <cstring>
using namespace std;

class String { char *str_; size_t len_;
public:
 String(const char *s) : str_(strdup(s)), len_(strlen(str_)) // ctor
 { cout << "String ctor: "; print(); cout << endl; }
 String(const String& s) : str_(strdup(s.str_)), len_(strlen(str_)) // cctor
 { cout << "String cctor: "; print(); cout << endl; }
 String& operator=(const String& s) {
 if (this != &s) {
 free(str_);
 str_ = strdup(s.str_);
 len_ = s.len_;
 }
 return *this;
 }
 ~String() { cout << "String dtor: "; print(); cout << endl; free(str_); } // dtor
 void print() const { cout << str_; }
};

#endif // __STRING_H
```

- Copy Constructor and Copy Assignment Operator added
- print() made a constant member function



# Program 15.05: Date Class: In header file with copy

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __DATE_H
#define __DATE_H
#include <iostream>
using namespace std;

char monthNames[] [4] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
char dayNames[] [10] = { "Monday", "Tuesday", "Wednesday", "Thursday",
 "Friday", "Saturday", "Sunday" };

class Date {
 enum Month { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
 enum Day { Mon, Tue, Wed, Thr, Fri, Sat, Sun };
 typedef unsigned int UINT;
 UINT date_; Month month_; UINT year_;

public:
 Date(UINT d, UINT m, UINT y) : date_(d), month_((Month)m), year_(y)
 { cout << "Date ctor: "; print(); cout << endl; }
 Date(const Date& d) : date_(d.date_), month_(d.month_), year_(d.year_)
 { cout << "Date ctor: "; print(); cout << endl; }
 Date& operator=(const Date& d) { date_ = d.date_; month_ = d.month_; year_ = d.year_;
 return *this;
 }
 ~Date() { cout << "Date dtor: "; print(); cout << endl; }
 void print() const { cout << date_ << "/" << monthNames[month_ - 1] << "/" << year_; }
 bool validDate() const { /* Check validity */ return true; } // Not Implemented (NI)
 Day day() const { /* Compute day from date using time.h */ return Mon; } // NI
};

#endif // __DATE_H
```

- Copy Constructor and Copy Assignment Operator added
- print(), validDate(), and day() made constant member functions



# Program 15.05: Name Class: In header file with copy

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __NAME_H
#define __NAME_H
#include <iostream>
using namespace std;

#include "String.h"

class Name {
 String firstName_, lastName_;
public:
 Name(const char* fn, const char* ln) : firstName_(fn), lastName_(ln)
 { cout << "Name ctor: "; print(); cout << endl; }
 Name(const Name& n) : firstName_(n.firstName_), lastName_(n.lastName_)
 { cout << "Name cctor: "; print(); cout << endl; }
 Name& operator=(const Name& n) {
 firstName_ = n.firstName_;
 lastName_ = n.lastName_;
 return *this;
 }
 ~Name() { cout << "Name dtor: "; print(); cout << endl; }
 void print() const
 { firstName_.print(); cout << " "; lastName_.print(); }
};

#endif // __NAME_H
```

- Copy Constructor and Copy Assignment Operator added
- print() made a constant member function



# Program 15.05: Address Class: In header file with copy

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __ADDRESS_H
#define __ADDRESS_H
#include <iostream>
using namespace std;

#include "String.h"

class Address {
 unsigned int houseNo_;
 String street_, city_, pin_;
public:
 Address(unsigned int hn, const char* sn, const char* cn, const char* pin) :
 houseNo_(hn), street_(sn), city_(cn), pin_(pin)
 { cout << "Address ctor: "; print(); cout << endl; }
 Address(const Address& a) :
 houseNo_(a.houseNo_), street_(a.street_), city_(a.city_), pin_(a.pin_)
 { cout << "Address cctor: "; print(); cout << endl; }
 Address& operator=(const Address& a) {
 houseNo_ = a.houseNo_; street_ = a.street_; city_ = a.city_; pin_ = a.pin_;
 return *this;
 }
 ~Address() { cout << "Address dtor: "; print(); cout << endl; }
 void print() const {
 cout << houseNo_ << " "; street_.print(); cout << " ";
 city_.print(); cout << " "; pin_.print();
 }
};

#endif // __ADDRESS_H
```

- Copy Constructor and Copy Assignment Operator added
- print() made a constant member function



# Program 15.05: Credit Card Class: In header file with edit options

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
#include <iostream>
using namespace std;
#include "Date.h"
#include "Name.h"
#include "Address.h"
class CreditCard { typedef unsigned int UINT; char *cardNumber_;
 Name holder_; Address addr_; Date issueDate_, expiryDate_; UINT cvv_;
public:
 CreditCard(const char* cNumber, const char* fn, const char* ln,
 unsigned int hn, const char* sn, const char* cn, const char* pin,
 UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear, UINT cvv) :
 holder_(fn, ln), addr_(hn, sn, cn, pin), issueDate_(1, issueMonth, issueYear),
 expiryDate_(1, expiryMonth, expiryYear), cvv_(cvv)
 { cardNumber_ = new char[strlen(cNumber) + 1]; strcpy(cardNumber_, cNumber);
 cout << "CC ctor: "; print(); cout << endl; }
 ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; }

 void setHolder(const Name& h) { holder_ = h; } // Change holder name
 void setAddress(const Address& a) { addr_ = a; } // Change address
 void setIssueDate(const Date& d) { issueDate_ = d; } // Change issue date
 void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
 void setCVV(UINT v) { cvv_ = v; } // Change cvv number
 void print() const { cout << cardNumber_ << " "; holder_.print(); cout << " "; addr_.print();
 cout << " "; issueDate_.print(); cout << " "; expiryDate_.print(); cout << " "; cout << cvv_;
 };
#endif // __CREDIT_CARD_H
```

- Set methods added
- print() made a constant member function



# Program 15.05: Credit Card Class Application

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

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

#include "CreditCard.h"

int main() {
 CreditCard cc("5321711934640027", "Sharlock", "Holmes",
 221, "Baker Street", "London", "NW1 6XE", 7, 2014, 6, 2016, 811);
 cout << endl; cc.print(); cout << endl << endl;

 cc.setHolder(Name("David", "Cameron"));
 cc.setAddress(Address(10, "Downing Street", "London", "SW1A 2AA"));
 cc.setIssueDate(Date(1, 7, 2017));
 cc.setExpiryDate(Date(1, 6, 2019));
 cc.setCVV(127);
 cout << endl; cc.print(); cout << endl << endl;

 return 0;
} // Construction of Data Members & Object

5321711934640027 Sharlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Jun/2016 811

// Construction & Destruction of temporary objects

5321711934640027 David Cameron 10 Downing Street London SW1A 2AA 1/Jul/2017 1/Jun/2019 127

// Destruction of Data Members & Object

- We could change address, issue date, expiry date, and cvv. This is fine
- We could change the name of the holder! This should not be allowed

```



# Program 15.06: Credit Card Class: Constant data members

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;

class CreditCard { typedef unsigned int UINT;
 char *cardNumber_;
 const Name holder_; // Holder name cannot be changed after construction
 Address addr_;
 Date issueDate_, expiryDate_; UINT cvv_;
public:
 CreditCard(...) : ... { ... }
 ~CreditCard() { ... }

 void setHolder(const Name& h) { holder_ = h; } // Change holder name
 // error C2678: binary '=' : no operator found which takes a left-hand operand
 // of type 'const Name' (or there is no acceptable conversion)

 void setAddress(const Address& a) { addr_ = a; } // Change address
 void setIssueDate(const Date& d) { issueDate_ = d; } // Change issue date
 void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
 void setCVV(UINT v) { cvv_ = v; } // Change cvv number

 void print() { ... }
};

#endif // __CREDIT_CARD_H
```

- We prefix Name holder\_ with const. Now the holder name cannot be changed after construction
- In setHolder(), we get a compilation error for holder\_ = h; in an attempt to change holder\_.
- With const prefix Name holder\_ becomes constant – unchangeable



# Program 15.06: Credit Card Class: Clean

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;

class CreditCard { typedef unsigned int UINT;
 char *cardNumber_;
 const Name holder_; // Holder name cannot be changed after construction
 Address addr_;
 Date issueDate_, expiryDate_; UINT cvv_;
public:
 CreditCard(...) : ... { ... }
 ~CreditCard() { ... }

 void setAddress(const Address& a) { addr_ = a; } // Change address
 void setIssueDate(const Date& d) { issueDate_ = d; } // Change issue date
 void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
 void setCVV(UINT v) { cvv_ = v; } // Change cvv number

 void print() { ... }
};

#endif // __CREDIT_CARD_H
```

- Method `setHolder()` removed



# Program 15.06: Credit Card Class Application: Revised

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

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

#include "CreditCard.h"

int main() {
 CreditCard cc("5321711934640027", "Sharlock", "Holmes",
 221, "Baker Street", "London", "NW1 6XE", 7, 2014, 6, 2016, 811);
 cout << endl; cc.print(); cout << endl << endl;

 // cc.setHolder(Name("David", "Cameron"));
 cc.setAddress(Address(10, "Downing Street", "London", "SW1A 2AA"));
 cc.setIssueDate(Date(1, 7, 2017));
 cc.setExpiryDate(Date(1, 6, 2019));
 cc.setCVV(127);
 cout << endl; cc.print(); cout << endl << endl;

 return 0;
}
// Construction of Data Members & Object

5321711934640027 Sharlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Jun/2016 811

// Construction & Destruction of temporary objects

5321711934640027 Sharlock Holmes 10 Downing Street London SW1A 2AA 1/Jul/2017 1/Jun/2019 127

// Destruction of Data Members & Object

- Now holder_ cannot be changed. So we are safe
- However, it is still possible to replace or edit the card number. This, too, should be disallowed

```



# Program 15.07: Credit Card Class: cardMember\_Issue

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;

class CreditCard { typedef unsigned int UINT;
 char *cardNumber_; // Card number is editable as well as replaceable
 const Name holder_; // Holder name cannot be changed after construction
 Address addr_;
 Date issueDate_, expiryDate_; UINT cvv_;
public:
 CreditCard(...) : ... { ... }
 ~CreditCard() { ... }

 void setAddress(const Address& a) { addr_ = a; } // Change address
 void setIssueDate(const Date& d) { issueDate_ = d; } // Change issue date
 void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
 void setCVV(UINT v) { cvv_ = v; } // Change cvv number

 void print() { ... }
};

#endif // __CREDIT_CARD_H
```

- It is still possible to replace or edit the card number
- To make the `cardNumber_` non-replaceable, we need to make this pointer constant
- Further, to make it non-editable we need to make `cardNumber_` point to a constant string
- Hence, we change `char *cardNumber_` to `const char * const cardNumber_`



# Program 15.07: Credit Card Class: cardMember\_Issue

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;
class CreditCard {
 typedef unsigned int UINT;
 const char * const cardNumber_; // Card number cannot be changed after construction
 const Name holder_; // Holder name cannot be changed after construction
 Address addr_; Date issueDate_, expiryDate_; UINT cvv_;
public:
 CreditCard(const char* cNumber, const char* fn, const char* ln,
 unsigned int hn, const char* sn, const char* cn, const char* pin,
 UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear, UINT cvv) :
 holder_(fn, ln), addr_(hn, sn, cn, pin), issueDate_(1, issueMonth, issueYear),
 expiryDate_(1, expiryMonth, expiryYear), cvv_(cvv)
 {
 cardNumber_ = new char[strlen(cNumber) + 1]; // ERROR: No assignment to const pointer
 strcpy(cardNumber_, cNumber); // ERROR: No copy to const C-string
 cout << "CC ctor: "; print(); cout << endl;
 }
 ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; }

 // Set methods and print method skipped ...
};

#endif // __CREDIT_CARD_H
```

- **cardNumber\_** is now a constant pointer to a constant string
- With this the allocation for the C-string fails in the body as constant pointer cannot be assigned
- Further, copy of C-string (`strcpy()`) fails as copy of constant C-string is not allowed
- We need to move these codes to the initialization list



# Program 15.07: Credit Card Class: cardMember\_ Issue Resolved

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#include <iostream>
using namespace std;
#include "String.h"
#include "Date.h"
#include "Name.h"
#include "Address.h"
class CreditCard {
 typedef unsigned int UINT;
 const char * const cardNumber_; // Card number cannot be changed after construction
 const Name holder_; // Holder name cannot be changed after construction
 Address addr_; Date issueDate_, expiryDate_; UINT cvv_;
public:
 CreditCard(const char* cNumber, const char* fn, const char* ln,
 unsigned int hn, const char* sn, const char* cn, const char* pin,
 UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear, UINT cvv) :
 cardNumber_(strncpy(new char[strlen(cNumber)+1], cNumber)),
 holder_(fn, ln), addr_(hn, sn, cn, pin), issueDate_(1, issueMonth, issueYear),
 expiryDate_(1, expiryMonth, expiryYear), cvv_(cvv)
 { cout << "CC ctor: "; print(); cout << endl; }
 ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; }
 void setAddress(const Address& a) { addr_ = a; } // Change address
 void setIssueDate(const Date& d) { issueDate_ = d; } // Change issue date
 void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
 void setCVV(UINT v) { cvv_ = v; } // Change cvv number
 void print() { cout << cardNumber_ << " "; holder_.print(); cout << " "; addr_.print();
 cout << " "; issueDate_.print(); cout << " "; expiryDate_.print(); cout << " "; cout << cvv_;
 };
};
```

- Note the initialization of `cardNumber_` in initialization list
- All constant data members must be initialized in initialization list



# mutable Data Members

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- While a *constant* data member is *not changeable* even in a *non-constant object*, a **mutable** data member is *changeable* in a *constant object*
- **mutable** is provided to model *Logical (Semantic) const-ness* against the default *Bit-wise (Syntactic) const-ness* of C++
- Note that:
  - **mutable** is applicable only to data members and not to variables
  - Reference data members cannot be declared **mutable**
  - Static data members cannot be declared **mutable**
  - *const* data members cannot be declared **mutable**
- If a data member is declared **mutable**, then it is legal to assign a value to it from a *const* member function
- Let us see an example



# Program 15.08: mutable Data Members

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

```
#include <iostream>
using namespace std;
class MyClass {
 int mem_;
 mutable int mutableMem_;
public:
 MyClass(int m, int mm) : mem_(m), mutableMem_(mm) {}
 int getMem() const { return mem_; }
 void setMem(int i) { mem_ = i; }
 int getMutableMem() const { return mutableMem_; }
 void setMutableMem(int i) const { mutableMem_ = i; } // Okay to change mutable
};
int main() {
 const MyClass myConstObj(1, 2);

 cout << myConstObj.getMem() << endl;
 //myConstObj.setMem(3); // Error to invoke

 cout << myConstObj.getMutableMem() << endl;
 myConstObj.setMutableMem(4);

 return 0;
}
```

- `setMutableMem()` is a constant member function so that constant `myConstObj` can invoke it
- `setMutableMem()` can still set `mutableMem_` because `mutableMem_` is `mutable`
- In contrast, `myConstObj` cannot invoke `setMem()` and hence `mem_` cannot be changed



# Logical vis-a-vis Bit-wise Const-ness

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- `const` in C++, models *bit-wise* constant. Once an object is declared `const`, no part (actually, *no bit*) of it can be changed after construction (and initialization)
- However, while programming we often need an object to be *logically* constant. That is, the concept represented by the object should be constant; but if its representation need more data members for computation and modeling, these have no reason to be constant.
- `mutable` allows such surrogate data members to be changeable in a (bit-wise) constant object to model logically const objects
- To use `mutable` we shall look for:
  - A logically constant concept
  - A need for data members outside the representation of the concept; but are needed for computation



# Program 15.09: When to use mutable Data Members?

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- Typically, when a class represents a constant concept, and
- It computes a value first time and caches the result for future use

```
// Source: http://www.highprogrammer.com/alan/rants/mutable.html

#include <iostream>
using namespace std;
class MathObject { // Constant concept of PI
 mutable bool piCached_; // Needed for computation
 mutable double pi_; // Needed for computation
public:
 MathObject() : piCached_(false) {} // Not available at construction
 double pi() const { // Can access PI only through this method
 if (!piCached_) { // An insanely slow way to calculate pi
 pi_ = 4;
 for (long step = 3; step < 1000000000; step += 4) {
 pi_ += ((-4.0 / (double)step) + (4.0 / ((double)step + 2)));
 }
 piCached_ = true; // Now computed and cached
 }
 return pi_;
 }
};

int main() {
 const MathObject mo;
 cout << mo.pi() << endl; // Access PI
 return 0;
}
```

- Here a MathObject is logically constant; but we use mutable members for computation



# Program 15.10: When *not* to use mutable Data Members?

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- **mutable** should be rarely used – only when it is really needed. A bad example follows:

| Improper Design ( <b>mutable</b> )                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | Proper Design ( <b>const</b> )                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>class Employee {<br/>    string _name;<br/>    string _id;<br/>    mutable double _salary;<br/><br/>public:<br/>    Employee(string name = "No Name",<br/>             string id = "000-00-0000",<br/>             double salary = 0)<br/>        : _name(name), _id(id)<br/>    { _salary = salary; }<br/>    string getName() const;<br/>    void setName(string name);<br/>    string getId() const;<br/>    void setId(string id);<br/>    double getSalary() const;<br/>    void setSalary(double salary);<br/>    void promote(double salary) const<br/>    {_salary = salary;}<br/>};<br/>---<br/>const Employee john("JOHN","007",5000.0);<br/>// ...<br/>john.promote(20000.0);</pre> | <pre>class Employee {<br/>    const string _name;<br/>    const string _id;<br/>    double _salary;<br/><br/>public:<br/>    Employee(string name = "No Name",<br/>             string id = "000-00-0000",<br/>             double salary = 0)<br/>        : _name(name), _id(id)<br/>    { _salary = salary; }<br/>    string getName() const;<br/><br/>    string getId() const;<br/><br/>    double getSalary() const;<br/>    void setSalary(double salary);<br/>    void promote(double salary)<br/>    {_salary = salary;}<br/>};<br/>---<br/>Employee john("JOHN","007",5000.0);<br/>// ...<br/>john.promote(20000.0);</pre> |

- **Employee** is not logically constant. If it is, then `_salary` should also be `const`
- Design on right makes that explicit



# Module Summary

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

- Studied const-ness in C++
- In C++, there are three forms of const-ness
  - Constant Objects:
    - No change is allowed after construction
    - Cannot invoke normal member functions
  - Constant Member Functions:
    - Can be invoked by constant (as well as non-constant) objects
    - Cannot make changes to the object
  - Constant Data Members:
    - No change is allowed after construction
    - Must be initialized in the initialization list
- Further, learnt how to model logical const-ness over bit-wise const-ness by proper use of `mutable` members



# Instructor and TAs

Module 15

Partha Pratim  
Das

Objectives &  
Outline

Constant  
Objects

Constant  
Member  
Functions

Constant Data  
Members

Credit Card  
Example

mutable  
Members

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

# Module 16: Programming in C++

## static Members

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

- Understand **static** data member and member function



# Module Outline

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member  
Print Task

static  
Member  
function  
Print Task

Singleton  
Class

Summary

- **static data member**
  - Print Task
- **static member function**
  - Print Task
- Singleton Class



# static Data Member

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

## A **static** data member

- is associated with class not with object
- is shared by all the objects of a class
- needs to be defined outside the class scope (in addition to the declaration within the class scope) to avoid linker error
- must be initialized in a source file
- is constructed before `main()` starts and destructed after `main()` ends
- can be private / public type
- can be accessed
  - with the class-name followed by the scope resolution operator (`::`)
  - as a member of any object of the class
- virtually eliminates any need for global variables in OOPs environment



# Program 16.01: static Data Member

## A Simple Case

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

### Non static Data Member

```
#include<iostream>
using namespace std;
class MyClass { int x; // Non-static
public:
 void get() { x = 15; }
 void print() {
 x = x + 10;
 cout << "x =" << x << endl ;
 }
};

int main() {
 MyClass obj1, obj2;
 obj1.get(); obj2.get();

 obj1.print(); obj2.print();
 return 0 ;
}

x = 25 , x = 25
```

- x is a non-static data member
- x cannot be shared between obj1 & obj2
- Non-static data members do not need separate definitions - instantiated with the object
- Non-static data members are initialized during object construction

### static Data Member

```
#include<iostream>
using namespace std;
class MyClass { static int x; // Declare static
public:
 void get() { x = 15; }
 void print() {
 x = x + 10;
 cout << "x =" << x << endl;
 }
};

int MyClass::x = 0; // Define static data member
int main() {
 MyClass obj1, obj2;
 obj1.get(); obj2.get();

 obj1.print(); obj2.print();
 return 0 ;
}

x = 25 , x = 35
```

- x is static data member
- x is shared by all MyClass objects including obj1 & obj2
- static data members must be defined in the global scope
- static data members are initialized during program start-up



# Program 16.02: static Data Member Print Task

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

```
#include <iostream>
using namespace std;
class PrintJobs { int nPages_; // # of pages in current job
public:
 static int nTrayPages_; // # of pages remaining in the tray
 static int nJobs_; // # of print jobs executing
 PrintJobs(int np): nPages_(np) {
 ++nJobs_;
 cout << "Printing " << np << " pages" << endl;
 nTrayPages_ = nTrayPages_ - np;
 }
 ~PrintJobs() { --nJobs_; }
};

int PrintJobs::nTrayPages_ = 500; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0; // Definition and initialization -- no job to start with
int main() {
 cout << "Jobs = " << PrintJobs::nJobs_ << endl;
 cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
 PrintJobs job1(10);
 cout << "Jobs = " << PrintJobs::nJobs_ << endl;
 cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
 {
 PrintJobs job1(30), job2(20);
 cout << "Jobs = " << PrintJobs::nJobs_ << endl;
 cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
 PrintJobs::nTrayPages_ += 100; // Load 100 more pages
 }
 cout << "Jobs = " << PrintJobs::nJobs_ << endl;
 cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
 return 0;
}
```

Output:  
Jobs = 0  
Pages= 500  
Printing 10 pages  
Jobs = 1  
Pages= 490  
Printing 30 pages  
Printing 20 pages  
Jobs = 3  
Pages= 440  
Jobs = 1  
Pages= 540



# static Member Function

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

## A **static** member function

- does not have this pointer – not associated with any object
- cannot access non-static data members
- cannot invoke non-static member functions
- can be accessed
  - with the class-name followed by the scope resolution operator (::)
  - as a member of any object of the class
- is needed to read / write static data members
  - Again, for encapsulation static data members should be private
  - get()-set() idiom is built for access (static member functions in public)
- may initialize static data members even before any object creation
- cannot co-exist with a non-static version of the same function
- cannot be declared as const



# Program 16.03: static Data & Member Function Print Task (safe)

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

```
#include <iostream>
using namespace std;
class PrintJobs { int nPages_; // # of pages in current job
 static int nTrayPages_; // # of pages remaining in the tray
 static int nJobs_; // # of print jobs executing
public: PrintJobs(int np) : nPages_(np) { ++nJobs_;}
 cout << "Printing " << np << " pages" << endl;
 nTrayPages_ = nTrayPages_ - np;
 }
~PrintJobs() { --nJobs_; }
static int getJobs() { return nJobs_; }
static int checkPages() { return nTrayPages_; }
static void loadPages(int np) { nTrayPages_ += np; }
};

int PrintJobs::nTrayPages_ = 500; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0; // Definition and initialization -- no job to start with
int main() {
 cout << "Jobs = " << PrintJobs::getJobs() << endl;
 cout << "Pages= " << PrintJobs::checkPages() << endl;
 PrintJobs job1(10);
 cout << "Jobs = " << PrintJobs::getJobs() << endl;
 cout << "Pages= " << PrintJobs::checkPages() << endl;
 { PrintJobs job1(30), job2(20);
 cout << "Jobs = " << PrintJobs::getJobs() << endl;
 cout << "Pages= " << PrintJobs::checkPages() << endl;
 PrintJobs::loadPages(100);
 }
 cout << "Jobs = " << PrintJobs::getJobs() << endl;
 cout << "Pages= " << PrintJobs::checkPages() << endl;
 return 0;
}
```

Output:

```
Jobs = 0
Pages= 500
Printing 10 pages
Jobs = 1
Pages= 490
Printing 30 pages
Printing 20 pages
Jobs = 3
Pages= 440
Jobs = 1
Pages= 540
```



# Singleton Class

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

- A class is called a Singleton if it can have *only* one instance
- Many classes are singleton:
  - President of India
  - Prime Minister of India
  - Director of IIT Kharagpur
  - ...
- How to implement a Singleton Class?
- How to restrict that user can create *only* one instance?



# Program 16.04: static Data & Member Function Singleton Printer

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function  
Print Task

Singleton  
Class

Summary

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

class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
 bool blackAndWhite_, bothSided_;

 Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
 { cout << "Printer constructed" << endl; } // Private -- Printer cannot be constructed!

 static Printer *myPrinter_; // Pointer to the Instance of the Singleton Printer

public:
 ~Printer() { cout << "Printer destructed" << endl; }

 static const Printer& printer(bool bw = false, bool bs = false) { // Access the Printer
 if (!myPrinter_) myPrinter_ = new Printer(bw, bs); // Constructed for first call
 return *myPrinter_; // Reused from next time
 }
 void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};

Printer *Printer::myPrinter_ = 0;
int main()
{
 Printer::printer().print(10);
 Printer::printer().print(20);

 delete &Printer::printer();
 return 0;
}
```

Output:

Printer constructed  
Printing 10 pages  
Printing 20 pages  
Printer destructed

*In the recorded video the destructor was directly called by Printer::printer().~Printer(); This is wrong and will leak memory. It is corrected here with delete &Printer::printer();*



# Program 16.05: Using function-local static Data Singleton Printer

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

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

class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
 bool blackAndWhite_, bothSided_;

 Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
 { cout << "Printer constructed" << endl; }

 ~Printer() { cout << "Printer destructed" << endl; }

public:

 static const Printer& printer(bool bw = false, bool bs = false) {
 static Printer myPrinter(bw, bs); // The Singleton -- constructed the first time

 return myPrinter;
 }

 void print(int nP) const {
 cout << "Printing " << nP << " pages" << endl;
 }
};

int main()
{
 Printer::printer().print(10);
 Printer::printer().print(20);

 return 0;
}
```

Output:  
Printer constructed  
Printing 10 pages  
Printing 20 pages  
Printer destructed

- Function local static object is used
- No memory management overhead – so destructor too get private
- This is called *Meyer's Singleton*



# Module Summary

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

- Introduced **static** data member
- Introduced **static** member function
- Exposed to use of **static** members
- Singleton Class discussed



# Instructor and TAs

Module 16

Partha Pratim  
Das

Objectives &  
Outline

static data  
member

Print Task

static  
Member  
function

Print Task

Singleton  
Class

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

# Module 17: Programming in C++

## friend Function and friend Class

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

- Understand **friend** function and class



# Module Outline

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

- **friend function**
  - Matrix-Vector Multiplication
  - Linked List
- **friend class**
  - Linked List
  - Iterator
- **friend-ly Notes**



# Program 17.01: friend function – Basic Notion

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

| Ordinary function                                                                                                                                                                                                                                                                                        | friend function                                                                                                                                                                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#include&lt;iostream&gt; using namespace std;  class MyClass { int data_; public:     MyClass(int i) : data_(i) {}  };  void display(const MyClass&amp; a) {     cout &lt;&lt; "data = " &lt;&lt; a.data_; // Error 1 }  int main(){     MyClass obj(10);      display(obj);      return 0; }</pre> | <pre>#include&lt;iostream&gt; using namespace std;  class MyClass { int data_; public:     MyClass(int i) : data_(i) {}      friend void display(const MyClass&amp; a); };  void display(const MyClass&amp; a) {     cout &lt;&lt; "data = " &lt;&lt; a.data_; // Okay }  int main(){     MyClass obj(10);      display(obj);      return 0; }</pre> |
| <ul style="list-style-type: none"> <li>● <code>display()</code> is a non-member function</li> <li>● Error 1: '<code>MyClass::data_</code>' : cannot access private member declared in class '<code>MyClass</code>'</li> </ul>                                                                            | <ul style="list-style-type: none"> <li>● <code>display()</code> is a non-member function; but friend to class <code>MyClass</code></li> <li>● Able to access <code>data_</code> even though it is private in class <code>MyClass</code></li> <li>● Output: <code>data = 10</code></li> </ul>                                                         |

*In the recorded video `void display(const MyClass& a);` is included in the class `MyClass` on left by mistake. This should be ignored. It is corrected here.*



# friend function

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class

Linked List  
Iterator

Notes

Summary

- A **friend** function of a class
  - has access to the private and protected members of the class (breaks the encapsulation)
  - must have its prototype included within the scope of the class prefixed with the keyword **friend**
  - does not have its name qualified with the class scope
  - is not called with an invoking object of the class
  - can be declared **friend** in more than one classes
- A **friend** function can be a
  - global function
  - a member function of a class
  - a function template



# Program 17.02: Multiply a Matrix with a Vector

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

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

class Matrix; // Forward declaration

class Vector { int e_[3]; int n_;
public:
 Vector(int n) : n_(n) {
 // Arbitrary initialization
 for (int i = 0; i < n_; ++i)
 e_[i] = i + 1;
 }
 void Clear() { // Set a zero vector
 for (int i = 0; i < n_; ++i)
 e_[i] = 0;
 }
 void Show() { //Show the vector
 for (int i = 0; i < n_; ++i)
 cout << e_[i] << " ";
 cout << endl << endl;
 }
 friend Vector Prod(Matrix *pM,
 Vector *pV);
};
```

```
class Matrix { int e_[3][3]; int m_, n_;
public:
 Matrix(int m, int n) : m_(m), n_(n) {
 // Arbitrary initialization
 for (int i = 0; i < m_; ++i)
 for (int j = 0; j < n_; ++j)
 e_[i][j] = i + j;
 }
 void Show() { //Show the matrix
 for (int i = 0; i < m_; ++i) {
 for (int j = 0; j < n_; ++j)
 cout << e_[i][j] << " ";
 cout << endl;
 }
 cout << endl;
 }
 friend Vector Prod(Matrix *pM,
 Vector *pV);
};

Vector Prod(Matrix *pM, Vector *pV) {
 Vector v(pM->m_); v.Clear();
 for (int i = 0; i < pM->m_; i++)
 for (int j = 0; j < pM->n_; j++)
 v.e_[i] += pM->e_[i][j] * pV->e_[j];
 return v;
}
```

- **Vector Prod(Matrix\*, Vector\*);** is a global function
- **Vector Prod(Matrix\*, Vector\*);** is friend of class **Vector** as well as class **Matrix**
- This function accesses the private data members of both these classes



# Program 17.02: Multiply a Matrix with a Vector

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

```
int main() {
 Matrix M(2, 3);
 Vector V(3);

 Vector PV = Prod(&M, &V);

 M.Show();
 V.Show();
 PV.Show();

 return 0;
}
```

Output:

0 1 2

1 2 3

1 2 3

8 14

- **Vector Prod(Matrix\*, Vector\*);** is a global function
- **Vector Prod(Matrix\*, Vector\*);** is friend of class **Vector** as well as class **Matrix**
- This function accesses the private data members of both these classes



# Program 17.03: Linked List

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class

Linked List  
Iterator

Notes

Summary

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

class Node; // Forward declaration

class List {
 Node *head; // Head of the list
 Node *tail; // Tail of the list
public:
 List(Node *h = 0):
 head(h),
 tail(h) {}
 void display();
 void append(Node *p);
};

class Node {
 int info; // Data of the node
 Node *next; // Ptr to next node
public:
 Node(int i): info(i), next(0) { }
 friend void List::display();
 friend void List::append(Node *);
};

void List::display() {
 Node *ptr = head;
 while (ptr) {
 cout << ptr->info << " ";
 ptr = ptr->next;
 }
}

void List::append(Node *p) {
 if (!head) head = tail = p;
 else {
 tail->next = p;
 tail = tail->next;
 }
}

int main() {
 List l; // Init null list
 Node n1(1), n2(2), n3(3); // Few nodes
 l.append(&n1); // Add nodes to list
 l.append(&n2);
 l.append(&n3);
 l.display(); // Show list
 return 0;
}
```

- List is built on Node. Hence List needs to know the internals of Node
- void List::append(Node \*); needs the internals of Node – hence friend member function is used
- void List::display(); needs the internals of Node – hence friend member function is used
- We can do better with friend classes



# friend class

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class

Linked List  
Iterator

Notes

Summary

- A **friend** class of a class

- has access to the private and protected members of the class (breaks the encapsulation)
- does not have its name qualified with the class scope (not a nested class)
- can be declared **friend** in more than one classes

- A **friend** class can be a

- class
- class template



# Program 17.04: Linked List

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class

Linked List  
Iterator

Notes

Summary

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

class Node; // Forward declaration

class List {
 Node *head; // Head of the list
 Node *tail; // Tail of the list
public:
 List(Node *h = 0):
 head(h),
 tail(h) {}
 void display();
 void append(Node *p);
};

class Node {
 int info; // Data of the node
 Node *next; // Ptr to next node
public:
 Node(int i): info(i), next(0) { }
 //friend void List::display();
 //friend void List::append(Node *);
 friend class List;
};
}

void List::display() {
 Node *ptr = head;
 while (ptr) {
 cout << ptr->info << " ";
 ptr = ptr->next;
 }
}

void List::append(Node *p) {
 if (!head) head = tail = p;
 else {
 tail->next = p;
 tail = tail->next;
 }
}

int main() {
 List l; // Init null list
 Node n1(1), n2(2), n3(3); // Few nodes
 l.append(&n1); // Add nodes to list
 l.append(&n2);
 l.append(&n3);

 l.display(); // Show list
 return 0;
}
```

- **List class** is now a **friend** of **Node class**. Hence it has full visibility into the internals of **Node**
- When multiple member functions need to be friends, it is better to use **friend class**



# Program 17.05: Linked List with Iterator

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

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

class Node; class List;
class Iterator {
 Node *node; // Current Node
 List *list; // Current List
public:
 Iterator() : node(0), list(0) {}
 void begin(List *); // Init
 bool end(); // Check end
 void next(); // Go to next
 int data(); // Get node data
};
class List { Node *head, *tail;
public:
 List(Node *h=0): head(h), tail(h) {}
 void append(Node *p);
 friend class Iterator;
};
class Node { int info; Node *next;
public:
 Node(int i) : info(i), next(0) {}
 friend class List;
 friend class Iterator;
};
```

```
void Iterator::begin(List *l) {
 list = l; node = l->head; // Set list & Init
}
bool Iterator::end() { return node == 0; }
void Iterator::next() { node = node->next; }
int Iterator::data() { return node->info; }

void List::append(Node *p) {
 if (!head)
 head = tail = p;
 else {
 tail->next = p;
 tail = tail->next;
 }
}
int main() {
 List l; Node n1(1), n2(2), n3(3);
 l.append(&n1); l.append(&n2); l.append(&n3);

 Iterator i;
 for (i.begin(&l); !i.end(); i.next()) {
 cout << i.data() << " ";
 }
 return 0;
}
```

- An **Iterator** now traverses over the elements of the **List**
- **void List::display()** is dropped from **List** and can be written in **main()**
- **List class** is a friend of **Node class**
- **Iterator class** is a friend of **List** and **Node class**



# friend-ly Notes

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

- **friend**-ship is neither commutative nor transitive
  - A is a friend of B does not imply that B is a friend of A
  - A is a friend of B and B is a friend of C does not imply that A is a friend of C
- Visibility and Encapsulation
  - **public**: a declaration that is accessible to all
  - **protected**: a declaration that is accessible only to the class itself and its subclasses
  - **private**: a declaration that is accessible only to the class itself
  - **friend**: a declaration that is accessible only to **friend**'s of a class.  
**friend**'s tend to break data hiding and should be used judiciously.  
Like:
    - A function needs to access the internals of two (or more) independent classes (Matrix-Vector Multiplication)
    - A class is built on top of another (List-Node Access, List Iterator)
    - Certain situations of operator overloading (like streaming operators)



# Module Summary

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

- Introduced the notion of **friend** function
- Introduced the notion of **friend** class
- Studied the use of **friend** function and **friend** class with examples
- **friend** introduces visibility hole by breaking encapsulation
  - should be used with care



# Instructor and TAs

Module 17

Partha Pratim  
Das

Objectives &  
Outline

friend  
function

Matrix-Vector  
Multiplication  
Linked List

friend class  
Linked List  
Iterator

Notes

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=  
Unary Operators

Summary

# Module 18: Programming in C++

Overloading Operator for User-Defined Types: Part 1

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=  
Unary Operators

Summary

- Understand how to overload operators for a user-defined type (class)
- Understand the aspects of overloading by global function and member function



# Module Outline

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

- Motivation
- Operator Function
- Using Global function
  - public data members
  - private data members
- Using Member function
  - operator+
  - operator=
  - Unary operators



# Motivation

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

- We have seen how **overloading operator+** a C-string wrapped in **struct** allows us a compact notation for concatenation of two strings (Module 09)
- We have seen how **overloading operator=** can define the deep / shallow copy for a UDT and / or help with user-defined copy semantics (Module 14)
- In general, operator overloading helps us **build complete algebra** for UDT's much in the same line as is available for built-in types:
  - **Complex type:** Add (+), Subtract (-), Multiply (\*), Divide (/), Conjugate (!), Compare (==, !=, ...), etc.
  - **Fraction type:** Add (+), Subtract (-), Multiply (\*), Divide (/), Normalize (unary \*), Compare (==, !=, ...), etc.
  - **Matrix type:** Add (+), Subtract (-), Multiply (\*), Divide (/), Invert (!), Compare (==), etc.
  - **Set type:** Union (+), Difference (-), Intersection (\*), Subset (<, <=), Superset (>, >=), Compare (==, !=), etc.
  - **Direct IO:** read (<<) and write (>>) for all types
- Advanced examples include:
  - **Smart Pointers:** De-reference (unary \*), Indirection (->), Copy (=), Compare (==, !=), etc.
  - **Function Objects or Functors:** Invocation ( () )



# Operator Functions in C++: RECAP (Module 9)

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

- Introduces a new keyword: `operator`
- Every operator is associated with an operator function that defines its behavior

| Operator Expression    | Operator Function                          |
|------------------------|--------------------------------------------|
| <code>a + b</code>     | <code>operator+(a, b)</code>               |
| <code>a = b</code>     | <code>operator=(a, b)</code>               |
| <code>c = a + b</code> | <code>operator=(c, operator+(a, b))</code> |

- Operator functions are implicit for predefined operators of built-in types and cannot be redefined
- An operator function may have a signature as:

```
MyType a, b; // An enum or struct
```

```
// Operator function
```

```
MyType operator+(const MyType&, const MyType&);
```

```
a + b // Calls operator+(a, b)
```

- C++ allows users to define an operator function and overload it



# Non-Member Operator Function

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

- A non-member operator function may be a

- Global Function
- friend Function

- **Binary Operator:**

```
MyType a, b; // An enum, struct or class
```

```
MyType operator+(const MyType&, const MyType&); // Global
friend MyType operator+(const MyType&, const MyType&); // Friend
```

- **Unary Operator:**

```
MyType operator++(const MyType&); // Global
```

```
friend MyType operator++(const MyType&); // Friend
```

- **Note:** The parameters may not be constant and may be passed by value.  
The return may also be by reference and may be constant

- **Examples:**

| Operator Expression | Operator Function               |
|---------------------|---------------------------------|
| a + b               | operator+(a, b)                 |
| a = b               | operator=(a, b)                 |
| ++a                 | operator++(a)                   |
| a++                 | operator++(a, int) Special Case |
| c = a + b           | operator=(c, operator+(a, b))   |



# Member Operator Function

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

## ● Binary Operator:

```
MyType a, b; // MYType is a Class
MyType operator+(const MyType&); // Operator function
```

- The left operand is the invoking object – right is taken as a parameter

## ● Unary Operator:

```
MyType operator-(); // Operator function for Unary minus
MyType operator++(); // For Pre-Incremente
MyType operator++(int); // For post-Incremente
```

- The only operand is the invoking object

- **Note:** The parameters may not be constant and may be passed by value.  
The return may also be by reference and may be constant

## ● Examples:

| Operator Expression | Operator Function                 |
|---------------------|-----------------------------------|
| a + b               | a.operator+(b)                    |
| a = b               | a.operator=(b)                    |
| ++a                 | a.operator++()                    |
| a++                 | a.operator++(int) // Special Case |
| c = a + b           | c.operator =(a.operator+(b))      |



# Operator Overloading – Summary of Rules: RECAP (Module 9)

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function  
  
public data  
members  
  
private data  
members

Using member  
function  
  
operator+  
operator-  
  
Unary Operators

Summary

- No new operator such as `**`, `<>`, or `&|` can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be changed
  - Preserves arity
  - Preserves precedence
  - Preserves associativity
- These operators can be overloaded:  
`[] + - * / % & | ~ ! = += -= *= /= %= = &= |=`  
`<< >> >>= <<= == != < > <= >= && || ++ -- , ->* -> ( ) [ ]`
- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), `sizeof`, and `?:` (ternary conditional) cannot be overloaded
- The overloads of operators `&&`, `||`, and `,` (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator `->` must either return a raw pointer or return an object (by reference or by value), for which operator `->` is in turn overloaded
- For a member operator function, invoking object is passed implicitly as the left operand but the right operand is passed explicitly
- For a non-member operator function (Global/friend) operands are always passed explicitly



# Program 18.01: Using Global Function – Unsafe (public Data members)

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator\*=  
Unary Operators

Summary

## Overloading + for complex addition

```
#include <iostream>
using namespace std;
struct complx { // public data member
 double re;
 double im;
} ;
complx operator+ (complx &a, complx &b) {
 complx r;
 r.re = a.re + b.re;
 r.im = a.im + b.im;
 return r;
}
int main(){
 complx d1 , d2 , d;
 d1.re = 10.5; d1.im = 12.25;
 d2.re = 20.5; d2.im = 30.25;
 d = d1 + d2;
 cout << "Real:" << d.re;
 cout << "Imaginary:" << d.im;
 return 0;
}
```

- **Output:** Real: 31, Imaginary: 42.5

- **operator+** is overloaded to perform addition of two complex numbers which are of **struct complx** type

## Overloading + for string cat

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
String operator+(const String& s1,
 const String& s2) {
 String s;
 s.str = (char *) malloc(strlen(s1.str) +
 strlen(s2.str) + 1);
 strcpy(s.str, s1.str);
 strcat(s.str, s2.str);
 return s;
}
int main() {
 String fName, lName, name;
 fName.str = strdup("Partha ");
 lName.str = strdup("Das");
 name = fName + lName; // Overload operator +
 cout << "First Name: " << fName.str << endl;
 cout << "Last Name: " << lName.str << endl;
 cout << "Full Name: " << name.str << endl;
 return 0;
}
```

- **Output:** First Name: Partha, Last Name: Das,  
Full name: Partha Das

- **operator+** is overloaded to perform concat of first name and last to form full name. The data type is **struct String**



# Program 18.02: Using Global Function – Safe (private Data members)

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

```
#include <iostream>
using namespace std;
class Complex { // Private data members
 double re, im;
public:
 Complex(double a=0.0, double b=0.0):
 re(a), im(b) {}
 ~Complex() {}
 void display();
 double real() { return re; }
 double img() { return im; }
 double set_real(double r) { re = r; }
 double set_img(double i) { im = i; }
} ;
void Complex::display() {
 cout << re;
 cout << " +j " << im << endl;
}

Complex operator+(Complex &t1, Complex &t2) {
 Complex sum;
 sum.set_real(t1.real() + t2.real());
 sum.set_img(t1.img() + t2.img());
 return sum;
}

int main() {
 Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
 cout << "1st complex No:";
 c1.display();
 cout << "2nd complex No:";
 c2.display();
 c3 = c1 + c2;
 cout << "Sum = ";
 c3.display();
 return 0;
}
```

## Output:

```
1st complex No: 4.5 +j 25.25
2nd complex No: 8.3 +j 10.25
Sum = 12.8 +j 35.5
```

- Accessing private data members inside operator functions is clumsy
- Critical data members need to be exposed (get/set) violating encapsulation
- Solution: Member operator function or friend operator function



# Program 18.03: Using Member Function

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

```
#include <iostream>
using namespace std;
class Complex { // Private data members
 double re, im;
public:
 Complex(double a=0.0, double b=0.0):
 re(a), im(b) {}
 ~Complex() {}
 void display();
 Complex operator+(const Complex &c) {
 Complex r;
 r.re = re + c.re;
 r.im = im + c.im;
 return r;
 }
};
```

```
void Complex::display(){
 cout << re;
 cout << " +j " << im << endl;
}
int main() {
 Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
 cout << "1st complex No:";
 c1.display();
 cout << "2nd complex No:";
 c2.display();
 c3 = c1 + c2;
 cout << "Sum = ";
 c3.display();
 return 0;
}
```

## Output:

```
1st complex No: 4.5 +j 25.25
2nd complex No: 8.3 +j 10.25
Sum = 12.8 +j 35.5
```

- Performing  $c1 + c2$  is equivalent to  $c1.operator+(c2)$
- $c1$  invokes the `operator+` function and  $c2$  is passed as an argument
- Similarly we can implement all binary operators (`%,-,*` etc..)
- **Note:** No need of two arguments in overloading



# Program 14.14: Overloading operator=

## RECAP (Module 14)

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=  
Unary Operators

Summary

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

class String { public: char *str_; size_t len_;
 String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // ctor
 String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // cctor
 ~String() { free(str_); } // dtor
 String& operator=(const String& s) {
 if (this != &s) {
 free(str_);
 str_ = strdup(s.str_);
 len_ = s.len_;
 }
 return *this;
 }
 void print() { cout << "(" << str_ << ":" << len_ << ")" << endl; }
};

int main() { String s1 = "Football", s2 = "Cricket";
 s1.print(); s2.print();
 s1 = s1; s1.print();
 return 0;
}

(Football: 8)
(Cricket: 7)
(Football: 8)
```

- Check for self-copy (`this != &s`)
- In case of self-copy, do nothing



# Notes on Overloading operator=

## RECAP (Module 14)

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=  
Unary Operators

Summary

- Overloaded operator= may choose between Deep and Shallow Copy for Pointer Members
  - Deep copy allocates new space for the contents and copies the pointed data
  - Shallow copy merely copies the pointer value – hence, the new copy and the original pointer continue to point to the same data
- If operator= is not overloaded by the user, compiler provides a free one.
- Free operator= can make only a shallow copy
- If the constructor uses operator new, operator= should be overloaded
- If there is a need to define a copy constructor then operator= must be overloaded and vice-versa



# Program 18.04: Overloading Unary Operators

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

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

class MyClass { int data;
public:
 MyClass(int d): data(d) { }

 MyClass& operator++() { // Pre-increment:
 ++data; // Operate and return the operated object
 return *this;
 }
 MyClass operator++(int) { // Post-Increment:
 MyClass t(data); // Return the (copy of) object; operate the object
 ++data;
 return t;
 }
 void disp() { cout << "Data = " << data << endl; }
};

int main() {
 MyClass obj1(8);
 obj1.disp();

 MyClass obj2 = obj1++;
 obj2.disp(); obj1.disp();

 obj2 = ++obj1;
 obj2.disp(); obj1.disp();

 return 0;
}
```

- **Output**

```
Data = 8
Data = 8
Data = 9
Data = 10
Data = 10
```

- The **pre-operator** should first perform the operation (increment / decrement / other) and then return the object. Hence its return type should be **MyClass&** and it should return **\*this**;

- The **post-operator** should perform the operation (increment / decrement / other) after it returns the original value. Hence it should copy the original object in a temporary **MyClass t**; and then **return t**. Its return type should be **MyClass**



# Program 18.05: Overloading Unary Operators: Pre-increment & Post Increment

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=

Unary Operators

Summary

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

class MyClass { int data;
public:
 MyClass(int d) : data(d) { }

 MyClass& operator++() { // Pre-Operator
 data *= 2;
 return *this;
 }
 MyClass operator++(int) { // Post-Operator
 MyClass t(data);
 data /= 3;
 return t;
 }
 void disp() { cout << "Data = " << data << endl; }
};

int main(){
 MyClass obj1(12);
 obj1.disp();

 MyClass obj2 = obj1++;
 obj2.disp(); obj1.disp();

 obj2 = ++obj1;
 obj2.disp(); obj1.disp();

 return 0;
}
```

- Output

Data = 12  
Data = 12  
Data = 4  
Data = 8  
Data = 8

- The pre-operator and the post-operator need not merely increment / decrement

- They may be used for any other computation as this example shows

- However, it is a good design practice to keep close to the native semantics of the operator



# Module Summary

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=  
Unary Operators

Summary

- Introduced operator overloading for user-defined types
- Illustrated methods of overloading operators using global functions and member functions
- Outlined semantics for overloading binary and unary operators



# Instructor and TAs

Module 18

Partha Pratim  
Das

Objectives &  
Outline

Motivation

Operator  
Function

Using global  
function

public data  
members  
private data  
members

Using member  
function

operator+  
operator=  
Unary Operators

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+  
-

Overloading  
IO Operators

Guidelines

Summary

# Module 19: Programming in C++

Overloading Operator for User-Defined Types: Part 2

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+  
-

Overloading  
IO Operators

Guidelines

Summary

- Understand how to overload operators for a user-defined type (class)
- Understand the aspects of overloading by friend function and its advantages



# Module Outline

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- Issues in Operator Overloading
- Extending operator+
- Overloading IO Operators
- Guidelines for Operator Overloading



# Operator Function for UDT RECAP (Module 18)

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- Operator Function options:

- Global Function
- Member Function
- friend Function

- **Binary Operator:**

```
MyType a, b; // An enum, struct or class
MyType operator+(const MyType&, const MyType&); // Global
MyType operator+(const MyType&); // Member
friend MyType operator+(const MyType&, const MyType&); // Friend
```

- **Unary Operator:**

```
MyType operator++(const MyType&); // Global
MyType operator++(); // Member
friend MyType operator++(const MyType&); // Friend
```

- **Examples:**

| Expression | Function        | Remarks         |
|------------|-----------------|-----------------|
| a + b      | operator+(a, b) | global / friend |
| ++a        | operator++(a)   | global / friend |
| a + b      | a.operator+(b)  | member          |
| ++a        | a.operator++()  | member          |



# Issue 1: Extending operator+

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- Consider a Complex class. We have learnt how to overload operator+ to add two Complex numbers:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
```

```
d3 = d1 + d2; // d3 = 4.1 +j 6.5
```

- Now we want to extend the operator so that a Complex number and a real number (no imaginary part) can be added together:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
```

```
d3 = d1 + 6.2; // d3 = 8.7 +j 3.2
```

```
d3 = 4.2 + d2; // d3 = 5.8 +j 3.3
```

- We show why global operator function is not good for this
- We show why member operator function cannot do this
- We show how friend function achieves this



# Issue 2: Overloading IO Operators: operator<<, operator>>

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+  
-

Overloading  
IO Operators

Guidelines

Summary

- Consider a Complex class. Suppose we want to overload the streaming operators for this class so that we can write the following code:

```
Complex d;

cin >> d;

cout << d;
```

- Let us note that these operators deal with stream types defined in `iostream`, `ostream`, and `istream`:
  - `cout` is an `ostream` object
  - `cin` is an `istream` object
- We show why global operator function is not good for this
- We show why member operator function cannot do this
- We show how friend function achieves this



# Program 19.01: Extending operator+ with Global Function

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

```
#include <iostream>
using namespace std;
class Complex { public: double re, im;
 explicit Complex(double r = 0, double i = 0): re(r), im(i) {}
 void disp() { cout << re << " +j " << im << endl; }
};

Complex operator+ (const Complex &a, const Complex &b) { // Overload 1
 return Complex(a.re + b.re, a.im + b.im);
}

Complex operator+ (const Complex &a, double d) { // Overload 2
 Complex b(d); return a + b; // Create temporary object and use Overload 1
}

Complex operator+ (double d, const Complex &b) { // Overload 3
 Complex a(d); return a + b; // Create temporary object and use Overload 1
}

int main(){
 Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

 d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.5
 d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2
 d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3
 return 0;
}
```

- Works fine with global functions - 3 separate overloading are provided
  - A bad solution as it breaks the encapsulation – as discussed in Module 18
  - Let us try to use member function
- 
- Note: A simpler solution uses Overload 1 and implicit casting (for this we need to remove explicit before constructor). But that too breaks encapsulation. We discuss this when we take up cast operators



# Program 19.02: Extending operator+ with Member Function

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

```
#include <iostream>
using namespace std;
class Complex { double re, im;
public:
 explicit Complex(double r = 0, double i = 0) : re(r), im(i) { }
 void disp() { cout << re << " +j " << im << endl; }
 Complex operator+ (const Complex &a) { // Overload 1
 return Complex(re + a.re, im + a.im);
 }
 Complex operator+ (double d) { // Overload 2
 Complex b(d); return *this + b; // Create temporary object and use Overload 1
 }
};
int main(){
 Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

 d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.5
 d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2

 //d3 = 4.2 + d2; // Overload 3 is not possible - needs an object of left
 //d3.disp();
 return 0;
}
```

- Overload 1 and 2 works
- Overload 3 cannot be done because the left operand is double – not an object
- Let us try to use friend function
- Note: This solution too avoids the feature of cast operators



# Program 19.03: Extending operator+ with friend Function

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

```
#include <iostream>
using namespace std;
class Complex { double re, im; public:
 explicit Complex(double r = 0, double i = 0) : re(r), im(i) { }
 void disp() { cout << re << " +j " << im << endl; }
 friend Complex operator+ (const Complex &a, const Complex &b) { // Overload 1
 return Complex(a.re + b.re, a.im + b.im);
 }
 friend Complex operator+ (const Complex &a, double d) { // Overload 2
 Complex b(d); return a + b; // Create temporary object and use Overload 1
 }
 friend Complex operator+ (double d, const Complex &b) { // Overload 3
 Complex a(d); return a + b; // Create temporary object and use Overload 1
 }
};

int main(){
 Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

 d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.5
 d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2
 d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3
 return 0;
}
```

- Works fine with friend functions - 3 separate overloading are provided
- Preserves the encapsulation too

• Note: A simpler solution uses only Overload 1 and implicit casting (for this we need to remove explicit before constructor) will be discussed when we take up cast operators



# Overloading IO Operators: operator<<, operator>>

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- Consider `operator<<` for `Complex` class. This operator should take an `ostream` object (stream to write to) and a `Complex` (object to write). Further it allows to chain the output. So for the following code

```
Complex d1, d2;

cout << d1 << d2; // (cout << d1) << d2;
```

the signature of `operator<<` may be one of:

```
// Global function
ostream& operator<< (ostream& os, const Complex &a);
```

```
// Member function in ostream
ostream& ostream::operator<< (const Complex &a);
```

```
// Member function in Complex
ostream& Complex::operator<< (ostream& os);
```

- Object to write is passed by constant reference
- Return by reference for `ostream` object is used so that chaining would work



# Program 19.04: Overloading IO Operators with Global Function

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

```
#include <iostream>
using namespace std;
class Complex {
public: double re, im;
 Complex(double r = 0, double i = 0): re(r), im(i) { }
};

ostream& operator<< (ostream& os, const Complex &a) {
 os << a.re << " +j " << a.im << endl;
 return os;
}

istream& operator>> (istream& is, Complex &a) {
 is >> a.re >> a.im;
 return is;
}

int main(){
 Complex d;

 cin >> d;

 cout << d;

 return 0;
}
```

- Works fine with global functions
- A bad solution as it breaks the encapsulation – as discussed in Module 18
- Let us try to use member function



# Overloading IO Operators with Member Function

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines  
Summary

- Case 1: `operator<<` is a member in `ostream` class:

```
ostream& ostream::operator<< (const Complex &a);
```

This is not possible as `ostream` is a class in C++ standard library and we are not allowed to edit it to include the above signature

- Case 2: `operator<<` is a member in `Complex` class:

```
ostream& Complex::operator<< (ostream& os);
```

In this case, the invocation of streaming will change to:

```
d << cout; // Left operand is the invoking object
```

This certainly spoils the natural syntax

- IO operators cannot be overloaded by member functions
- Let us try to use friend function



# Guidelines for Operator Overloading

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- Use **global function** when encapsulation is not a concern. For example, using `struct String { char* str; }` to wrap a C-string and overload `operator+` to concatenate strings and build a `String` algebra
- Use **member function** when the left operand is necessarily a class where the operator function is a member and multiple types of operands are not involved
- Use **friend function**, otherwise
- While overloading an operator, try to **preserve its natural semantics** for built-in types as much as possible. For example, `operator+` in a `Set` class should compute union and NOT intersection
- Usually stick to the **parameter passing** conventions (built-in types by value and UDT's by constant reference)
- Decide on the **return type** based on the natural semantics for built-in types. For example, as in pre-increment and post-increment operators
- Consider the **effect of casting** on operands
- Only overload the operators that you may need (**minimal design**)



# Module Summary

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+  
-

Overloading  
IO Operators

Guidelines

Summary

- Several issues operator overloading has been discussed
- Use of **friend** is illustrated in versatile forms of overloading with examples
- Discussed the overloading IO (streaming) operators
- Guidelines for operator overloading is summarized
- Use operator overloading to build algebra for:
  - Complex numbers
  - Fractions
  - Strings
  - Vector and Matrices
  - Sets
  - and so on ...



# Instructor and TAs

Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features  
Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

# Module 20: Programming in C++

## Namespace

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- Understand namespace as a free scoping mechanism to organize code better



# Module Outline

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- **namespace Fundamental**
- **namespace Scenarios**
- **namespace Features**
  - Nested namespace
  - using namespace
  - Global namespace
  - Standard Library std namespace
  - namespaces are open
- **namespace vis-a-vis class**
- **Lexical Scope**



# namespace Fundamental

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it
- It is used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries
- namespace provides a class-like modularization without class-like semantics
- Obliviates the use of File Level Scoping of C (file )static



# Program 20.01: namespace Fundamental

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

## ● Example:

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

namespace MyNameSpace {
 int myData; // Variable in namespace
 void myFunction() { cout << "MyNameSpace myFunction" << endl; } // Function in namespace
 class MyClass { int data; // Class in namespace
 public:
 MyClass(int d) : data(d) { }
 void display() { cout << "MyClass data = " << data << endl; }
 };
}
int main() {
 MyNameSpace::myData = 10; // Variable name qualified by namespace name
 cout << "MyNameSpace::myData = " << MyNameSpace::myData << endl;

 MyNameSpace::myFunction(); // Function name qualified by namespace name

 MyNameSpace::MyClass obj(25); // Class name qualified by namespace name
 obj.display();

 return 0;
}
```

- A name in a namespace is prefixed by the name of it
- Beyond scope resolution, all namespace items are treated as global



# Scenario 1: Redefining a Library Function (Program 20.02)

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- cstdlib has a function int abs(int n); that returns the absolute value of parameter n
- You need a special int abs(int n); function that returns the absolute value of parameter n if n is between -128 and 127. Otherwise, it returns 0
- Once you add your abs, you cannot use the abs from library! It is hidden and gone!
- namespace** comes to your rescue

Name-hiding: abs()

```
#include <iostream>
#include <cstdlib>

int abs(int n) {
 if (n < -128) return 0;
 if (n > 127) return 0;
 if (n < 0) return -n;
 return n;
}

int main() {
 std::cout << abs(-203) << " "
 << abs(-6) << " "
 << abs(77) << " "
 << abs(179) << std::endl;
 // Output: 0 6 77 0
 return 0;
}
```

namespace: abs()

```
#include <iostream>
#include <cstdlib>

namespace myNS {
 int abs(int n) {
 if (n < -128) return 0;
 if (n > 127) return 0;
 if (n < 0) return -n;
 return n;
 }
}

int main() {
 std::cout << myNS::abs(-203) << " "
 << myNS::abs(-6) << " "
 << myNS::abs(77) << " "
 << myNS::abs(179) << std::endl;
 // Output: 0 6 77 0
 std::cout << abs(-203) << " "
 << abs(-6) << " "
 << abs(77) << " "
 << abs(179) << std::endl;
 // Output: 203 6 77 179
 return 0;
}
```



# Scenario 2: Students' Record Application: The Setting (Program 20.03)

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- An organization is developing an application to process students records
- class St for Students and class StReg for list of Students are:

```
#include <iostream>
using namespace std;
class St { public: // A Student
 typedef enum GENDER { male = 0, female } ;
 St(char *n, GENDER g) : name(strcpy(new char[strlen(n) + 1], n)), gender(g) {}
 void setRoll(int r) { roll = r; } // Set roll while adding the student
 GENDER getGender() { return gender; } // Get the gender for processing
 friend ostream& operator<< (ostream& os, const St& s) { // Print a record
 cout << ((s.gender == St::male) ? "Male " : "Female ")
 << s.name << " " << s.roll << endl;
 return os;
 }
 private:
 char *name; GENDER gender; // name and gender provided for the student
 int roll; // roll is assigned by the system
 };
 class StReg { // Students' Register
 St **rec; // List of students
 int nStudents; // Number of student
 public:
 StReg(int size) : rec(new St*[size]), nStudents(0) {}
 void add(Students* s) { rec[nStudents] = s; s->setRoll(++nStudents); }
 Students *getStudent(int r) { return (r == nStudents + 1) ? 0 : rec[r - 1]; }
 };
}
```

- The classes are included in a header file Students.h



# Scenario 2: Students' Record Application: Team at Work (Program 20.03)

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- Two engineers – **Sabita** and **Niloy** – are assigned to develop processing applications for male and female students respectively. Both are given the **Students.h** file
- The lead **Purnima** of Sabita and Niloy has the responsibility to integrate what they produce and prepare a single application for both male and female students. The engineers produce:

## Processing for males by Sabita

```
////////// App1.cpp //////////
#include <iostream>
using namespace std;
#include "Students.h"
extern StReg *reg;
void ProcessStdudents() {
 cout << "MALE STUDENTS: " << endl;
 int r = 1; St *s;
 while (s = reg->getStudent(r++))
 if (s->getGender() ==
 St::male)
 cout << *s;
 cout << endl << endl;
 return;
}
////////// Main.cpp //////////
#include <iostream>
using namespace std;
#include "Students.h"
StReg *reg = new StReg(1000);
int main() {
 St s("Partha", St::male); reg->add(&s);
 ProcessStdudents();
 return 0;
}
```

## Processing for females by Niloy

```
////////// App2.cpp //////////
#include <iostream>
using namespace std;
#include "Students.h"
extern StReg *reg;
void ProcessStdudents() {
 cout << "FEMALE STUDENTS: " << endl;
 int r = 1; St *s;
 while (s = reg->getStudent(r++))
 if (s->getGender() ==
 St::female)
 cout << *s;
 cout << endl << endl;
 return;
}
////////// Main.cpp //////////
#include <iostream>
using namespace std;
#include "Students.h"
StReg *reg = new StReg(1000);
int main() {
 St s("Ramala", St::female); reg->add(&s);
 ProcessStdudents();
 return 0;
}
```



# Scenario 2: Students' Record Application: The Integration Nightmare (Program 20.03)

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace

using namespace  
Global  
namespace

std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- To integrate, Purnima prepares the following main() in her Main.cpp where she intends to call the processing functions for males (as prepared by Sabita) and for females (as prepared by Niloy) one after the other:

```
#include <iostream>
using namespace std;
#include "Students.h"

void ProcessStdudents(); // Function from App1.cpp by Sabita
void ProcessStdudents(); // Function from App2.cpp by Niloy

StReg *reg = new StReg(1000);

int main() {
 St s1("Ramala", St::female); reg->add(&s1);
 St s2("Partha", St::male); reg->add(&s2);

 ProcessStdudents(); // Function from App1.cpp by Sabita
 ProcessStdudents(); // Function from App2.cpp by Niloy

 return 0;
}
```

- But the integration failed due to name clashes**
- Both use the same signature void ProcessStdudents(); for their respective processing function.**  
**Actually, they have several functions, classes, and variables in their respective development with the same name and with same / different purposes**
- How does Purnima perform the integration without major changes in the codes? – namespace**



# Scenario 2: Students' Record Application: Wrap in Namespace (Program 20.03)

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- Introduce two namespaces – App1 for **Sabita** and App2 for **Niloy**
- Wrap the respective codes:

### Processing for males by Sabita

```
////////// App1.cpp //////////
#include <iostream>
using namespace std;

#include "Students.h"

extern StReg *reg;

namespace App1 {
 void ProcessStudents() {
 cout << "MALE STUDENTS: " << endl;
 int r = 1;
 St *s;

 while (s = reg->getStudent(r++))
 if (s->getGender() == St::male)
 cout << *s;

 cout << endl << endl;
 return;
 }
};
```

### Processing for females by Niloy

```
////////// App2.cpp //////////
#include <iostream>
using namespace std;

#include "Students.h"

extern StReg *reg;

namespace App2 {
 void ProcessStudents() {
 cout << "FEMALE STUDENTS: " << endl;
 int r = 1;
 St *s;

 while (s = reg->getStudent(r++))
 if (s->getGender() == St::female)
 cout << *s;

 cout << endl << endl;
 return;
 }
};
```



# Scenario 2: Students' Record Application: A Good Night's Sleep (Program 20.03)

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- Now the integration gets smooth:

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

#include "Students.h"

namespace App1 { void ProcessStudents(); } // App1.cpp by Sabita

namespace App2 { void ProcessStudents(); } // App2.cpp by Niloy

StReg *reg = new StReg(1000);

int main() {
 St s1("Ramala", St::female); reg->add(&s1);
 St s2("Partha", St::male); reg->add(&s2);

 App1::ProcessStudents(); // App1.cpp by Sabita
 App2::ProcessStudents(); // App2.cpp by Niloy

 return 0;
}
```

- Clashing names are made distinguishable by distinct namespace names



# Program 20.04: Nested namespace

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- A namespace may be nested in another namespace

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

int data = 0; // Global name ::

namespace name1 {
 int data = 1; // In namespace name1
 namespace name2 {
 int data = 2; // In nested namespace name1::name2
 }
}

int main() {
 cout << data << endl; // 0
 cout << name1::data << endl; // 1
 cout << name1::name2::data << endl; // 2

 return 0;
}
```



# Program 20.05: Using using namespace and using for shortcut

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace

using namespace

Global  
namespace

std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- Using using namespace we can avoid lengthy prefixes

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

namespace name1 {
 int v11 = 1;
 int v12 = 2;
}

namespace name2 {
 int v21 = 3;
 int v22 = 4;
}

using namespace name1; // All symbols of namespace name1 will be available
using name2::v21; // Only v21 symbol of namespace name2 will be available

int main() {
 cout << v11 << endl; // name1::v11
 cout << name1::v12 << endl; // name1::v12
 cout << v21 << endl; // name2::v21
 cout << name2::v21 << endl; // name2::v21
 cout << v22 << endl; // Treated as undefined

 return 0;
}
```



# Program 20.06: Global namespace

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace

std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- using or using namespace hides some of the names

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

int data = 0; // Global Data

namespace name1 {
 int data = 1; // namespace Data
}

int main() {
 using name1::data;

 cout << data << endl; // 1 // name1::data -- Hides global data
 cout << name1::data << endl; // 1
 cout << ::data << endl; // 0 // ::data -- global data

 return 0;
}
```

- Items in Global namespace may be accessed by scope resolution operator (::)



# Program 20.07: std Namespace

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- Entire C++ Standard Library is put in its own namespace, called std

| Without using using std                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | With using using std                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#include &lt;iostream&gt;  int main(){     int num;     std::cout &lt;&lt; "Enter a value: " ;     std::cin &gt;&gt; num;     std::cout &lt;&lt; "value is: " ;     std::cout &lt;&lt; num ;     return 0; }</pre> <ul style="list-style-type: none"><li>Here, cout, cin are explicitly qualified by their namespace. So, to write to standard output, we specify std::cout; to read from standard input, we use std::cin</li><li>It is useful if a few library is to be used; no need to add entire std library to the global namespace</li></ul> | <pre>#include &lt;iostream&gt; using namespace std;  int main(){     int num;     cout &lt;&lt; "Enter a value: " ;     cin &gt;&gt; num;     cout &lt;&lt; "value is: " ;     cout &lt;&lt; num ;     return 0; }</pre> <ul style="list-style-type: none"><li>By the statement using namespace std; std namespace is brought into the current namespace, which gives us direct access to the names of the functions and classes defined within the library without having to qualify each one with std::</li><li>When several libraries are to be used it is a convenient method</li></ul> |



# Program 20.08: namespaces are Open

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features  
Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- namespaces are open: New Declarations can be added

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

```
namespace open
{ int x = 30; }
```

```
namespace open
{ int y = 40; }
```

```
int main()
{
 using namespace open;
 x = y = 20;
 cout << x << " " << y ;
 return 0 ;
}
```

Output: 20 20



# namespace vis-a-vis class

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

| namespace                                                                                                                                                                                                                                                                                                                      | class                                                                                                                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Every namespace is not a class</li><li>• A namespace can be reopened and more declaration added to it</li><li>• No instance of a namespace can be created</li><li>• using-declarations can be used to short-cut namespace qualification</li><li>• A namespace may be unnamed</li></ul> | <ul style="list-style-type: none"><li>• Every class defines a namespace</li><li>• A class cannot be reopened</li><li>• A class has multiple instances</li><li>• No using-like declaration for a class</li><li>• An unnamed class is not allowed</li></ul> |



# Lexical Scope

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- The scope of a name binding – an association of a name to an entity, such as a variable – is the part of a computer program where the binding is valid: where the name can be used to refer to the entity
- C++ supports a variety of scopes:
  - **Expression Scope** – restricted to one expression, mostly used by compiler
  - **Block Scope** – create local context
  - **Function Scope** – create local context associated with a function
  - **Class Scope** – context for data members and member functions
  - **Namespace Scope** – grouping of symbols for code organization
  - **File Scope** – limit symbols to a single file
  - **Global Scope** – outer-most, singleton scope containing the whole program



# Lexical Scope

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- Scopes may be named or Unnamed
  - Named Scope – Option to refer to the scope from outside
    - Class Scope – class name
    - Namespace Scope – namespace name or unnamed
    - Global Scope – "::"
  - Unnamed Scope
    - Expression Scope
    - Block Scope
    - Function Scope
    - File Scope
- Scopes may or may not be nested
  - Scopes that may be nested
    - Block Scope
    - Class Scope
    - Namespace Scope
  - Scopes that cannot be nested
    - Expression Scope
    - Function Scope – may contain Class Scopes
    - File Scope – will contain several other scopes
    - Global Scope – will contain several other scopes



# Module Summary

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

- Understood `namespace` as a scoping tool in C++
- Analyzed typical scenarios that `namespace` helps address
- Studied several features of `namespace`
- Understood how `namespace` is placed in respect of different lexical scopes of C++



# Instructor and TAs

Module 20

Partha Pratim  
Das

Objectives &  
Outline

namespace  
Fundamental

namespace  
Scenarios

namespace  
Features

Nested  
namespace  
using namespace  
Global  
namespace  
std namespace  
namespaces are  
Open

namespace  
vis-a-vis class

Lexical Scope

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++  
Semantics

Summary

# Module 21: Programming in C++

## Inheritance: Part 1 (Inheritance Semantics)

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++  
Semantics

Summary

- Revisit ISA Relationship in OOAD and understand how hierarchy can be created in C++ with Inheritance



# Module Outline

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++  
Semantics

Summary

- ISA Relationship
- Inheritance in C++
  - Semantics
  - Data Members and Object Layout
  - Member Functions
    - Overriding
    - Overloading
  - protected Access
  - Constructor & Destructor
  - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (private)
  - Implemented-As Semantics



# ISA Relationship

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++  
Semantics

Summary

- We often find one object is a *specialization / generalization* of another
- OOAD models this using **ISA** relationship
- C++ models **ISA** relationship by *Inheritance* of classes



# ISA Relationship

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++

Semantics

Summary

- Rose ISA Flower

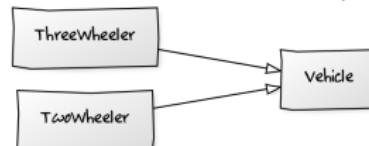
- Rose has the properties of Flower – like fragrance, having petals etc.
- Rose has some additional properties – like rosy fragrance
- Rose is a *specialization* of Flower
- Flower is a *generalization* of Rose

- Red Rose ISA Rose

- Red Rose has the properties of Rose – like rosy fragrance etc.
- Red Rose has some additional properties – like it is red
- Red Rose is a *specialization* of Rose
- Rose is a *generalization* of Red Rose



- TwoWheeler ISA Vehicle; ThreeWheeler ISA Vehicle



- Manager ISA Employee





# Inheritance in C++: Hierarchy

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++

Semantics

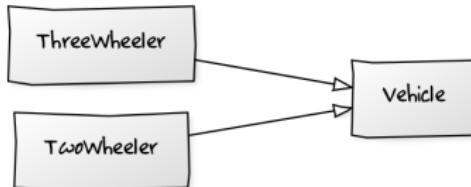
Summary

- Manager ISA Employee [Single Inheritance]



```
class Employee; // Base Class = Employee
class Manager: public Employee; // Derived Class = Manager
```

- TwoWheeler ISA Vehicle; ThreeWheeler ISA Vehicle [Hybrid Inheritance]



```
class Vehicle; // Base Class = Employee -- Root
class TwoWheeler: public Vehicle; // Derived Class = TwoWheeler
class ThreeWheeler: public Vehicle; // Derived Class = ThreeWheeler
```

- Red Rose ISA Rose ISA Flower [Multi-Level Inheritance]



```
class Flower; // Base Class = Flower -- Root
class Rose: public Flower; // Derived Class = Rose; Base Class = Rose
class RedRose: public Rose; // Derived Class = RedRose;
```



# Inheritance in C++: Phones

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++

Semantics

Summary

- Landline Phone
  - Call: By dial / keyboard
  - Answer
- Mobile Phone
  - Call: By keyboard – shows number
    - By Number
    - By Name
  - Answer
  - Redial
  - Set Ring Tone
  - Add Contact
    - Number
    - Name
- Smart Phone
  - Call: By touchscreen – shows number & photo
    - By Number
    - By Name
  - Answer
  - Redial
  - Set Ring Tone
  - Add Contact
    - Number
    - Name
    - Photo

- There exists a substantial overlap between the functionality of the phones
- A mobile phone is more capable than a land line phone and can perform (almost) all its functions
- A smart phone is more capable than a mobile phone and can perform (almost) all its functions
- These phones belong to a Specialization / Generalization hierarchy



# Inheritance in C++: Semantics

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++  
Semantics

Summary

- Derived **ISA** Base



```
class Base; // Base Class = Base
class Derived: public Base; // Derived Class = Derived
```

- Use keyword **public** after class name to denote inheritance
- Name of the Base class follow the keyword

**"Public inheritance means "is-a." Everything that applies to base classes must also apply to derived classes, because every derived class object is a base class object"**  
– Scott Meyers in Item 32, Effective C++ (3rd. Edition)



# Inheritance in C++: Semantics

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++  
Semantics

Summary

- Derived ISA Base
- Data Members
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the *same signature*
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
- Access Specification
  - Derived class *cannot access private* members of Base class
  - Derived class *can access protected* members of Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class



# Module Summary

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++  
Semantics

Summary

- Revisited Hierarchy or ISA Relationship in OOAD
- Introduced the Semantics of Inheritance in C++



# Instructor and TAs

Module 21

Partha Pratim  
Das

Objectives &  
Outline

ISA  
Relationship

Inheritance in  
C++  
Semantics

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

## Module 22: Programming in C++

Inheritance: Part 2 (Data Member & Member Function - Override)

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

- Understand how inheritance impacts data members and member functions
- Introduce overriding of member function and its interactions with overloading



# Module Outline

Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

- ISA Relationship
- Inheritance in C++
  - Semantics
  - Data Members and Object Layout
  - Member Functions
    - Overriding
    - Overloading
  - protected Access
  - Constructor & Destructor
  - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (private)
  - Implemented-As Semantics



# Inheritance in C++: Semantics

Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

- Derived ISA Base
- Data Members
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the *same signature*
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
- Access Specification
  - Derived class *cannot access private* members of Base class
  - Derived class *can access protected* members of Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class



# Inheritance in C++: Data Members and Object Layout

Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

- Derived **ISA** Base
- Data Members
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- Object Layout
  - Derived class *layout* contains an instance of the Base class
  - Further, Derived class *layout* will have data members of its own
  - C++ does not guarantee the *relative position* of the Base class instance and Derived class members



# Inheritance in C++: Data Members and Object Layout

Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

```
class B { // Base Class
 int data1B_;
public:
 int data2B_;
 // ...
};

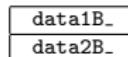
class D: public B { // Derived Class
 // Inherits B::data1B_
 // Inherits B::data2B_
 int infoD_; // Adds D::infoD_
public:
 / ...
};

B b;

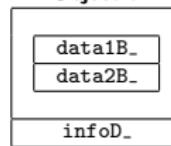
D d;
```

## Object Layout

Object b



Object d



d cannot access data1B\_ even though data\_ is a part of it!  
d can access data2B\_



# Inheritance in C++: Member Functions – Overrides and Overloads

Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

- Derived **ISA** Base
- Member Functions
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the *same signature*
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
  - Derived class *may add* new member functions
- Static Member Functions
  - Derived class *does not inherit* the static member functions of Base class
- Friend Functions
  - Derived class *does not inherit* the friend functions of Base class



# Inheritance in C++: Member Functions – Overrides and Overloads

## Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

### Inheritance

```
class B { // Base Class
public:
 void f(int i);
 void g(int i);
};

class D: public B { // Derived Class
public:
 // Inherits B::f(int)
 // Inherits B::g(int)
};

B b;
D d;

b.f(1); // Calls B::f(int)
b.g(2); // Calls B::g(int)

d.f(3); // Calls B::f(int)
d.g(4); // Calls B::g(int)
```

### Override & Overload

```
class B { // Base Class
public:
 void f(int);
 void g(int i);
};

class D: public B { // Derived Class
public:
 // Inherits B::f(int)
 void f(int); // Overrides B::f(int)
 void f(string&); // Overloads B::f(int)
 // Inherits B::g(int)
 void h(int i); // Adds D::h(int)
};

B b;
D d;

b.f(1); // Calls B::f(int)
b.g(2); // Calls B::g(int)

d.f(3); // Calls D::f(int)
d.g(4); // Calls B::g(int)

d.f("red"); // Calls D::f(string&)
d.h(5); // Calls D::h(int)
```

- D::f(int) overrides B::f(int)
- D::f(string) overloads B::f(int)



# Module Summary

Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

- Discussed the effect of inheritance on Data Members and Object Layout
- Discussed the effect of inheritance on Member Functions with special reference to Overriding and Overloading



# Instructor and TAs

Module 22

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

Data Members  
Overrides and  
Overloads

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Summary

# Module 23: Programming in C++

Inheritance: Part 3 (Constructor & Destructor - Object Lifetime)

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

- Understand protected access specifier
- Understand the construction and destruction process on an object hierarchy
- Revisit Object Lifetime for a hierarchy



# Module Outline

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

- ISA Relationship
- Inheritance in C++
  - Semantics
  - Data Members and Object Layout
  - Member Functions
    - Overriding
    - Overloading
  - [protected Access](#)
  - [Constructor & Destructor](#)
  - [Object Lifetime](#)
- Example – Phone Hierarchy
- Inheritance in C++ ([private](#))
  - Implemented-As Semantics



# Inheritance in C++: Semantics

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

- Derived ISA Base
- Data Members
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the *same signature*
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
- Access Specification
  - Derived class *cannot access private* members of Base class
  - Derived class *can access protected* members of Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class



# Inheritance in C++: Access Members of Base: protected Access

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

- Derived **ISA** Base
- Access Specification
  - Derived class *cannot access private* members of Base class
  - Derived class *can access public* members of Base class
- **protected** Access Specification
  - A new **protected** access specification is introduced for Base class
  - Derived class *can access protected* members of Base class
  - No other class or global function *can access protected* members of Base class
  - A **protected** member in Base class is like **public** in Derived class
  - A **protected** member in Base class is like **private** in other classes or global functions



# Inheritance in C++: protected Access

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

| private Access                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | protected Access                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>class B { private: // Inaccessible to child            // Inaccessible to others     int data_; public:     // ...     void Print() { cout &lt;&lt; "B Object: ";                   cout&lt;&lt;data_&lt;&lt;endl;     } };  class D: public B { int info_; public:     // ...     void Print() { cout &lt;&lt; "D Object: ";                   cout&lt;&lt;data_&lt;&lt;", ";                   cout&lt;&lt;info_&lt;&lt;endl;     } };  B b(0); D d(1, 2);  b.data_ = 5; // Inaccessible to all  b.Print(); d.Print();</pre> | <pre>class B { protected: // Accessible to child            // Inaccessible to others     int data_; public:     // ...     void Print() { cout &lt;&lt; "B Object: ";                   cout&lt;&lt;data_&lt;&lt;endl;     } };  class D: public B { int info_; public:     // ...     void Print() { cout &lt;&lt; "D Object: ";                   cout&lt;&lt;data_&lt;&lt;", ";                   cout&lt;&lt;info_&lt;&lt;endl;     } };  B b(0); D d(1, 2);  b.data_ = 5; // Inaccessible to others  b.Print(); d.Print();</pre> |
| <ul style="list-style-type: none"> <li>• <b>D::Print()</b> cannot access <b>B::data_</b> as it is <b>private</b></li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                         | <ul style="list-style-type: none"> <li>• <b>D::Print()</b> can access <b>B::data_</b> as it is <b>protected</b></li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                             |



# Inheritance in C++: Streaming

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

## Streaming in B

```
class B {
protected: int data_;
public:
 friend ostream& operator<<(ostream& os,
 const B& b) {
 os << b.data_ << endl;
 return os;
 }
};
class D: public B { int info_;
public:
 //friend ostream& operator<<(ostream& os,
 // const D& d) {
 // os << d.data_ << endl;
 // os << d.info_ << endl;
 // return os;
 //}
};

B b(0);
D d(1, 2);

cout << b; cout << d;
```

B Object: 0  
B Object: 1

- d printed as a B object; info\_ missing

## Streaming in B & D

```
class B {
protected: int data_;
public:
 friend ostream& operator<<(ostream& os,
 const B& b) {
 os << b.data_ << endl;
 return os;
 }
};
class D: public B { int info_;
public:
 friend ostream& operator<<(ostream& os,
 const D& d) {
 os << d.data_ << endl;
 os << d.info_ << endl;
 return os;
 }
};

B b(0);
D d(1, 2);

cout << b; cout << d;
```

B Object: 0  
D Object: 1 2

- d printed as a D object as expected



# Inheritance in C++: Constructor & Destructor

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

- Derived **ISA** Base

- Constructor-Destructor

- Derived class *inherits* the Constructors and Destructor of Base class (*but in a different semantics*)
- Derived class *cannot override* or *overload* a Constructor or the Destructor of Base class

- Construction-Destruction

- A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
- The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class



# Inheritance in C++: Constructor & Destructor

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access  
Constructor &  
Destructor

Object Lifetime

Summary

```
class B { protected: int data_;
public:
 B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }

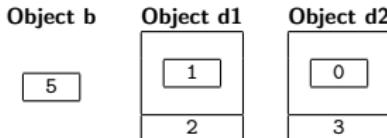
 ~B() { cout << "B::~B(): " << data_ << endl; }
 // ...
};
class D: public B { int info_;
public:
 D(int d, int i) : B(d), info_(i) // ctor-1: Explicit construction of Base
 { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }

 D(int i) : info_(i) // ctor-2: Default construction of Base
 { cout << "D::D(int): " << data_ << ", " << info_ << endl; }

 ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
 // ...
};

B b(5);
D d1(1, 2); // ctor-1: Explicit construction of Base
D d2(3); // ctor-2: Default construction of Base
```

## Object Layout





# Inheritance in C++: Object Lifetime

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Summary

```
class B { protected: int data_;
public:
 B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
 ~B() { cout << "B::~B(): " << data_ << endl; }
 // ...
};

class D: public B { int info_;
public:
 D(int d, int i) : B(d), info_(i) // Explicit construction of Base
 { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
 D(int i) : info_(i) // Default construction of Base
 { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
 ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
 // ...
};

B b(0);
D d1(1, 2);
D d2(3);
```

## Construction O/P

B::B(int): 0 // Obj. b  
B::B(int): 1 // Obj. d1  
D::D(int, int): 1, 2 // Obj. d1  
B::B(int): 0 // Obj. d2  
D::D(int): 0, 3 // Obj. d2

## Destruction O/P

D::~D(): 0, 3 // Obj. d2  
B::~B(): 0 // Obj. d2  
D::~D(): 1, 2 // Obj. d1  
B::~B(): 1 // Obj. d1  
B::~B(): 0 // Obj. b

- First construct base class object, then derived class object
- First destruct derived class object, then base class object



# Module Summary

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

- Understood the need and use of protected Access specifier
- Discussed the Construction and Destruction process of class hierarchy and related Object Lifetime



# Instructor and TAs

Module 23

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

protected  
Access

Constructor &  
Destructor

Object Lifetime

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

# Module 24: Programming in C++

## Inheritance: Part 4 (Example – Phone Hierarchy)

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

- Model a hierarchy of phones using inheritance



# Module Outline

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

- ISA Relationship
- Inheritance in C++
  - Semantics
  - Data Members and Object Layout
  - Member Functions
    - Overriding
    - Overloading
  - protected Access
  - Constructor & Destructor
  - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (private)
  - Implemented-As Semantics



# Phone Hierarchy

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

- Let us model a hierarchy of phones comprising:
  - Land line Phone
  - Mobile Phone
  - Smart Phone
- We model Helper classes
- We model each phone separately
- We model the phone hierarchy



# Helper Classes

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

| Class                    | Description                                          |
|--------------------------|------------------------------------------------------|
| <b>class PhoneNumber</b> | 12-digit phone number                                |
| <b>class Name</b>        | Subscriber Name (as string)                          |
| <b>class Photo</b>       | Image & Subscriber Name as alt text                  |
| <b>class RingTone</b>    | Audio & ring tone name                               |
| <b>class Contact</b>     | PhoneNumber, Name, and Photo (optional) of a contact |
| <b>class AddressBook</b> | List of contacts                                     |



# Land line Phone Model

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

- Landline Phone
  - Call: By dial / keyboard
  - Answer

```
class LandlinePhone {
 PhoneNumber number_;
 Name subscriber_;
 RingTone rTone_;

public:
 LandlinePhone(const char *num,
 const char *subs);

 void Call(const PhoneNumber *p);

 void Answer();

 friend ostream& operator<<(ostream& os,
 const LandlinePhone& p);
};
```



# Mobile Phone Model

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

## ● Mobile Phone

- Call: By keyboard – shows number
  - By Number
  - By Name
- Answer
- Redial
- Set Ring Tone
- Add Contact
  - Number
  - Name

```
class MobilePhone {
 PhoneNumber number_;
 Name subscriber_;
 RingTone rTone_;
 AddressBook aBook_;
 PhoneNumber *lastDial_;
 void SetLastDialed(const PhoneNumber& p);
 void ShowNumber();

public:
 MobilePhone(const char *num,
 const char *subs);

 void Call(PhoneNumber *p);
 void Call(const Name& n);

 void Answer();

 void ReDial();
 void SetRingTone(RingTone::RINGTONE r);
 void AddContact(const char *num = 0,
 const char *subs = 0);

 friend ostream& operator<<(ostream& os,
 const MobilePhone& p);
};
```



# Smart Phone Model

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

## ● Smart Phone

- Call: By touchscreen – shows number & photo
  - By Number
  - By Name
- Answer
- Redial
- Set Ring Tone
- Add Contact
  - Number
  - Name
  - Photo

```
class SmartPhone {
 PhoneNumber number_;
 Name subscriber_;
 RingTone rTone_;
 AddressBook aBook_;
 PhoneNumber *lastDial_;
 void SetLastDialed(const PhoneNumber& p);
 void ShowNumber();
 unsigned int size_;
 void DisplayPhoto();

public:
 SmartPhone(const char *num,
 const char *subs);

 void Call(PhoneNumber *p);
 void Call(const Name& n);

 void Answer();

 void ReDial();
 void SetRingTone(RingTone::RINGTONE r);
 void AddContact(const char *num = 0,
 const char *subs = 0);

 friend ostream& operator<<(ostream& os,
 const MobilePhone& p);
};
```



# Comparison of Phones

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

- Landline Phone
  - Call: By dial / keyboard
  - Answer
- Mobile Phone
  - Call: By keyboard – shows number
    - By Number
    - By Name
  - Answer
  - Redial
  - Set Ring Tone
  - Add Contact
    - Number
    - Name
- Smart Phone
  - Call: By touchscreen – shows number & photo
    - By Number
    - By Name
  - Answer
  - Redial
  - Set Ring Tone
  - Add Contact
    - Number
    - Name
    - Photo

- There exists a substantial overlap between the functionality of the phones
- A mobile phone is more capable than a land line phone and can perform (almost) all its functions
- A smart phone is more capable than a mobile phone and can perform (almost) all its functions
- These phones belong to a Specialization / Generalization hierarchy



# Hierarchy of Phones

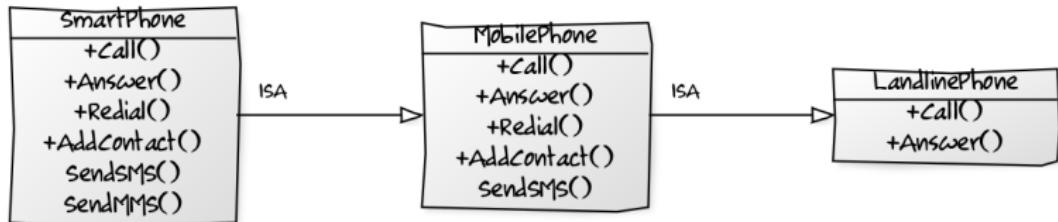
Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary



- **MobilePhone ISA LandlinePhone**
  - **LandlinePhone** is *generalization*
  - **MobilePhone** is *specialization*
  - **MobilePhone** inherits the properties of **LandlinePhone**
- **SmartPhone ISA MobilePhone**
  - **MobilePhone** is *generalization*
  - **SmartPhone** is *specialization*
  - **SmartPhone** inherits the properties of **MobilePhone**
- **ISA** is *transitive*



# Compare LandlinePhone and MobilePhone

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

```
class LandlinePhone {
 PhoneNumber number_;
 Name subscriber_;
 RingTone rTone_;

public:
 LandlinePhone(const char *num,
 const char *subs);
 void Call(const PhoneNumber *p);

 void Answer();

 friend ostream& operator<<(ostream& os,
 const LandlinePhone& p);
};

class MobilePhone {
 PhoneNumber number_;
 Name subscriber_;
 RingTone rTone_;
 AddressBook aBook_;
 PhoneNumber *lastDial_;
 void SetLastDialed(const PhoneNumber& p);
 void ShowNumber();

public:
 MobilePhone(const char *num,
 const char *subs);
 void Call(PhoneNumber *p);
 void Call(const Name& n);
 void ReDial();
 void Answer();
 void SetRingTone(RingTone::RINGTONE r);
 void AddContact(const char *num = 0,
 const char *subs = 0);

 friend ostream& operator<<(ostream& os,
 const MobilePhone& p);
};
```



# Hierarchy of Phones

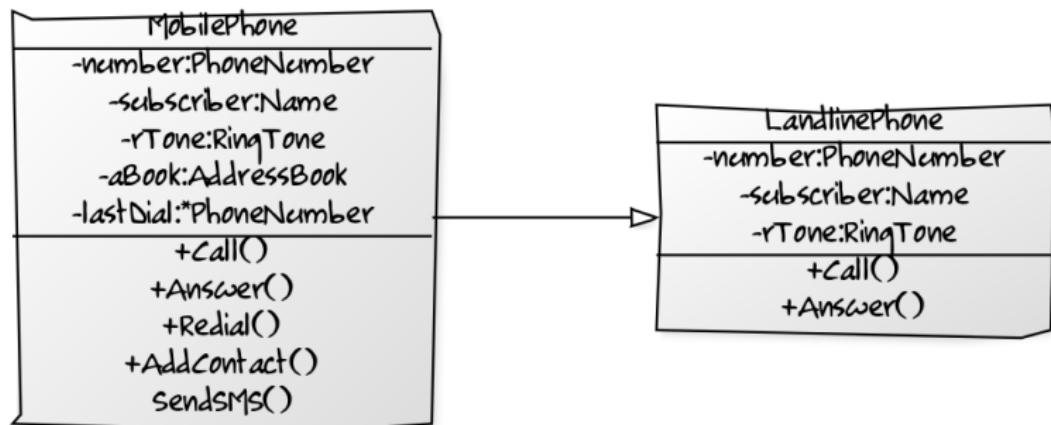
Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary





# MobilePhone ISA LandlinePhone

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

## Base Class

```
class LandlinePhone {
protected:
 PhoneNumber number_;
 Name subscriber_;
 RingTone rTone_;

public:
 LandlinePhone(const char *num,
 const char *subs) :
 number_(num), subscriber_(subs),
 rTone_() {}

 void Call(const PhoneNumber *p);

 void Answer();

 friend ostream& operator<<(ostream& os,
 const LandlinePhone& p);
};
```

## Derived Class

```
class MobilePhone : public LandlinePhone {
protected:
 //PhoneNumber number_;
 //Name subscriber_;
 //RingTone rTone_;
 AddressBook aBook_;
 PhoneNumber *lastDial_;
 void SetLastDialed(const PhoneNumber& p);
 void ShowNumber();

public:
 MobilePhone(const char *num,
 const char *subs) :
 LandlinePhone(num, subs), // Base ctor
 lastDial_(0) {}

 void Call(const PhoneNumber *p); // Override
 void Call(const Name& n); // Overload
 //void Answer();
 void ReDial();
 void SetRingTone(RingTone::RINGTONES r);
 void AddContact(const char *num = 0,
 const char *subs = 0);

 friend ostream& operator<< (ostream& os,
 const MobilePhone& p);
};
```



# SmartPhone ISA MobilePhone

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

| <b>Base Class</b>                          | <b>Derived Class</b>                         |
|--------------------------------------------|----------------------------------------------|
| class MobilePhone : public LandlinePhone { | class SmartPhone : public MobilePhone {      |
| protected:                                 | protected:                                   |
| //PhoneNumber number_;                     | //PhoneNumber number_;                       |
| //Name subscriber_;                        | //Name subscriber_;                          |
| //RingTone rTone_;                         | //RingTone rTone_;                           |
| AddressBook aBook_;                        | //AddressBook aBook_;                        |
| PhoneNumber *lastDial_;                    | //PhoneNumber *lastDial_;                    |
| void SetLastDialed(const PhoneNumber& p);  | //void SetLastDialed(const PhoneNumber& p);  |
| void ShowNumber();                         | //void ShowNumber();                         |
|                                            | unsigned int size_;                          |
|                                            | void DisplayPhoto()                          |
| public:                                    | public:                                      |
| MobilePhone(const char *num,               | SmartPhone(const char *num,                  |
| const char *subs) :                        | const char *subs) :                          |
| LandlinePhone(num, subs), // Base ctor     | MobilePhone(num, subs), // Base ctor         |
| lastDial_(0) {}                            | lastDial_(0) {}                              |
|                                            | void Call(const PhoneNumber *p); // Override |
| void Call(const Name& n); // Overload      | void Call(const Name& n); // Override        |
| //void Answer();                           | //void Answer();                             |
| void ReDial();                             | // void ReDial(); // Override                |
| void SetRingTone(RingTone::RINGTONE r);    | //void SetRingTone(RingTone::RINGTONE r);    |
| void AddContact(const char *num = 0,       | //void AddContact(const char *num = 0,       |
| const char *subs = 0);                     | //const char *subs = 0);                     |
|                                            | friend ostream& operator<< (ostream& os,     |
| friend ostream& operator<< (ostream& os,   | const SmartPhone& p);                        |
| const MobilePhone& p);                     |                                              |
| }                                          | };                                           |



# Phone Hierarchy

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

```
class Phone {
public:
 virtual void Call(const PhoneNumber *p)
 = 0;
 virtual void Answer() = 0;
 virtual void ReDial() = 0;
};
class LandlinePhone: public Phone {
 void ReDial()
 { cout << "Not implemented" << endl; }

protected:
 PhoneNumber number_;
 Name subscriber_;
 RingTone rTone_;
public:
 LandlinePhone(const char *num,
 const char *subs) :
 number_(num), subscriber_(subs),
 rTone_() {}
 void Call(const PhoneNumber *p);
 void Answer();
 friend ostream& operator<<(ostream& os,
 const LandlinePhone& p);
};

class MobilePhone : public LandlinePhone {
protected:
 AddressBook aBook_;
 PhoneNumber *lastDial_;
 void SetLastDialed(const PhoneNumber& p);
 void ShowNumber();
public:
 MobilePhone(const char *num,
 const char *subs) :
 LandlinePhone(num, subs), // Base ctor
 lastDial_(0) {}
 void Call(const PhoneNumber *p); // Override
 void Call(const Name& n); // Overload
 void ReDial();
 friend ostream& operator<<(ostream& os,
 const MobilePhone& p);
};
class SmartPhone : public MobilePhone {
protected: unsigned int size_;
 void DisplayPhoto()
public:
 SmartPhone(const char *num,
 const char *subs) :
 MobilePhone(num, subs), // Base ctor
 lastDial_(0) {}
 void Call(const PhoneNumber *p); // Override
 void Call(const Name& n); // Override
 void ReDial(); // Override
 friend ostream& operator<<(ostream& os,
 const SmartPhone& p);
};
```



# Hierarchy of Phones

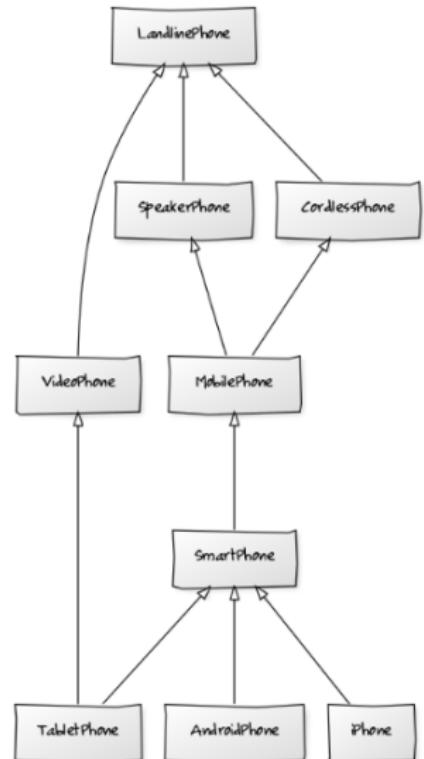
Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary





# Module Summary

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

- Using the Phone Hierarchy as an example analyzed the design process with inheritance



# Instructor and TAs

Module 24

Partha Pratim  
Das

Objectives &  
Outline

Example –  
Phone  
Hierarchy

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

# Module 25: Programming in C++

Inheritance: Part 5 (private & protected Inheritance)

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

- Explore restricted forms of inheritance (**private** and **protected**) in C++ and their semantic implications



# Module Outline

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

- ISA Relationship
- Inheritance in C++
  - Semantics
  - Data Members and Object Layout
  - Member Functions
    - Overriding
    - Overloading
  - protected Access
  - Constructor & Destructor
  - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (private)
  - Implemented-As Semantics



# Inheritance in C++: Semantics

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

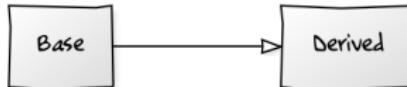
protected  
Inheritance

Visibility

Use &  
Examples

Summary

- Derived **ISA** Base



```
class Base; // Base Class = Base
class Derived: public Base; // Derived Class = Derived
```

- Use keyword **public** after class name to denote inheritance
- Name of the Base class follow the keyword



# Inheritance Exercise: What is the output?

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

```
class B {
public:
 B() { cout << "B "; }
 ~B() { cout << "~B "; } };

class C {
public:
 C() { cout << "C "; }
 ~C() { cout << "~C "; } };

class D : public B {
 C data_;
public:
 D() { cout << "D " << endl; }
 ~D() { cout << "~D "; }
};

int main() {
 D d;

 return 0;
}
```



# Inheritance Exercise: What is the output?

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

```
class B {
public:
 B() { cout << "B "; }
 ~B() { cout << "~B "; } };

class C {
public:
 C() { cout << "C "; }
 ~C() { cout << "~C "; } };

class D : public B {
 C data_;
public:
 D() { cout << "D " << endl; }
 ~D() { cout << "~D "; }
};

int main() {
 D d;

 return 0;
}
```

**Output:**

B C D  
~D ~C ~B



# private Inheritance

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

## ● private Inheritance

- Definition

```
class Base;
class Derived: private Base;
```

- Use keyword **private** after class name
- Name of the Base class follow the keyword
- **private** inheritance **does not** mean generalization / specialization

- Private inheritance means nothing during software design, only during software implementation
- Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions
  - Scott Meyers in Item 32, Effective C++ (3rd. Edition)



# private Inheritance

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

## public Inheritance

```
class Person {...};

class Student:
 public Person {...};

 // anyone can eat
 void eat(const Person& p);

 // only students study
 void study(const Student& s);

Person p; // p is a Person

Student s; // s is a Student

eat(p); // fine, p is a Person

eat(s); // fine, s is a Student,
 // and a Student is-a Person

study(s); // fine

study(p); // error! p isn't a Student
```

Compilers converts a derived class object (Student) into a base class object (Person) if the inheritance relationship is public

## private Inheritance

```
class Person { ... };

class Student: // inheritance is now private
 private Person { ... };

 // anyone can eat
 void eat(const Person& p);

 // only students study
 void study(const Student& s);

Person p; // p is a Person

Student s; // s is a Student

eat(p); // fine, p is a Person

eat(s); // error! a Student isn't a Person
```

Compilers will not convert a derived class object (Student) into a base class object (Person) if the inheritance relationship is private



# protected Inheritance

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

## ● protected Inheritance

- Definition

```
class Base;
class Derived: protected Base;
```

- Use keyword **protected** after class name
- Name of the Base class follow the keyword
- **protected** inheritance **does not** mean generalization / specialization

● Private inheritance means something entirely different (from public inheritance), and protected inheritance is something whose meaning eludes me to this day

– Scott Meyers in Item 32, Effective C++ (3rd. Edition)



# Visibility across Access and Inheritance

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

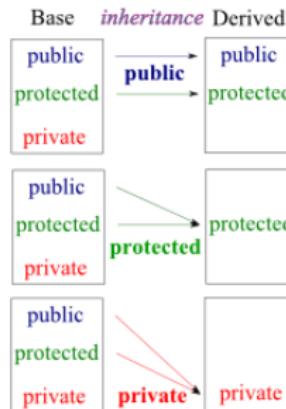
Visibility

Use &  
Examples

Summary

## Visibility Matrix

|            |           | Inheritance |           |         |
|------------|-----------|-------------|-----------|---------|
|            |           | public      | protected | private |
| Visibility | public    | public      | protected | private |
|            | protected | protected   | protected | private |
|            | private   | private     | private   | private |





# Inheritance Exercise: What is the output?

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

```
class B {
protected:
 B() { cout << "B "; }
 ~B() { cout << "~B "; }
};
class C : public B {
protected:
 C() { cout << "C "; }
 ~C() { cout << "~C "; }
};
class D : private C {
 C data_;
public:
 D() { cout << "D " << endl; }
 ~D() { cout << "~D "; }
};

int main() {
 D d;

 return 0;
}
```



# Inheritance Exercise: What is the output?

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

```
class B {
protected:
 B() { cout << "B "; }
 ~B() { cout << "~B "; }
};
class C : public B {
protected:
 C() { cout << "C "; }
 ~C() { cout << "~C "; }
};
class D : private C {
 C data_;
public:
 D() { cout << "D " << endl; }
 ~D() { cout << "~D "; }
};

int main() {
 D d;

 return 0;
}
```

**Output:**

B C B C D  
~D ~C ~B ~C ~B



# Inheritance Exercise: Access Rights

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

## Inaccessible Members

```
class A {
private: int x;
protected: int y;
public: int z;
};

class B : public A {
private: int u;
protected: int v;
public: int w; void f() { x; }
};

class C: protected A {
private: int u;
protected: int v;
public: int w; void f() { x; }
};

class D: private A {
private: int u;
protected: int v;
public: int w; void f() { x; }
};

class E : public B {
public: void f() { x; u; }
};

class F : public C {
public: void f() { x; u; }
};

class G : public D {
public: void f() { x; y; z; u; }
};
```

## Accessible Members

```
void f(A& a,
 B& b, C& c, D& d,
 E& e, F& f, G& g) {
 a.z;

 b.z;
 b.w;

 c.w;

 d.w;

 e.z;
 e.w;

 f.w;

 g.w;
}
```



# Car HAS-A Engine: Composition OR private Inheritance?

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

## Simple Composition

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

class Engine {
public:
 Engine(int numCylinders) { }
 // Starts this Engine
 void start() { }
};

class Car {
public:
 // Initializes this Car with 8 cylinders
 Car() : e_(8) { }

 // Start this Car by starting its Engine
 void start() { e_.start(); }

private:
 Engine e_; // Car has-a Engine
};

int main() {
 Car c;

 c.start();

 return 0;
}
```

## private Inheritance

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

class Engine {
public:
 Engine(int numCylinders) { }
 // Starts this Engine
 void start() { }
};

class Car : private Engine { // Car has-a Engine
public:
 // Initializes this Car with 8 cylinders
 Car() : Engine(8) { }

 // Start this Car by starting its Engine
 using Engine::start;
};

int main() {
 Car c;

 c.start();

 return 0;
}
```



# private Inheritance

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

- Use composition when you can, private inheritance when you have to

- Private inheritance means nothing during software design, only during software implementation
- Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions
  - Scott Meyers in Item 32, Effective C++ (3rd. Edition)



# Module Summary

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

- Introduced restricted forms of inheritance and **protected specifier**
- Discussed how **private inheritance** is used for *Implemented-As Semantics*



# Instructor and TAs

Module 25

Partha Pratim  
Das

Objectives &  
Outline

Inheritance in  
C++

private  
Inheritance

protected  
Inheritance

Visibility

Use &  
Examples

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

# Module 26: Programming in C++

## Dynamic Binding (Polymorphism): Part 1

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

- Understand Casting in a class hierarchy
- Understand Static and Dynamic Binding



# Module Outline

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

- Casting
  - Upcast & Downcast
  - Static and Dynamic Binding



# Casting: Basic Rules

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

- Casting is performed when a value (variable) of one type is used in place of some other type

```
int i = 3;
double d = 2.5;

double result = d / i; // i is cast to double and used
```

- Casting can be implicit or explicit

```
int i = 3;
double d = 2.5;

double *p = &d;

d = i; // implicit

i = d; // implicit -- // warning C4244: '=' : conversion from 'double' to 'int',
 // possible loss of data
i = (int)d; // explicit

i = p; // error C2440: '=' : cannot convert from 'double *' to 'int'
i = (int)p; // explicit
```



# Casting: Basic Rules

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

- (Implicit) Casting between unrelated classes is not permitted

```
class A { int i; };
class B { double d; };

A a;
B b;

A *p = &a;
B *q = &b;

a = b; // error C2679: binary '=' : no operator found
 // which takes a right-hand operand of type 'main::B'

a = (A)b; // error C2440: 'type cast' : cannot convert from 'main::B' to 'main::A'

b = a; // error C2679: binary '=' : no operator found
 // which takes a right-hand operand of type 'main::A'

b = (B)a; // error C2440: 'type cast' : cannot convert from 'main::A' to 'main::B'

p = q; // error C2440: '=' : cannot convert from 'main::B *' to 'main::A *'

q = p; // error C2440: '=' : cannot convert from 'main::A *' to 'main::B *'

p = (A*)&b; // Forced -- Okay
q = (B*)&a; // Forced -- Okay
```



# Casting: Basic Rules

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

- Forced Casting between unrelated classes is dangerous

```
class A { public: int i; };
class B { public: double d; };

A a;
B b;

a.i = 5;
b.d = 7.2;

A *p = &a;
B *q = &b;

cout << p->i << endl; // prints 5
cout << q->d << endl; // prints 7.2

p = (A*)&b;
q = (B*)&a;

cout << p->i << endl; // prints -858993459 ----- GARBAGE
cout << q->d << endl; // prints -9.25596e+061 ----- GARBAGE
```



# Casting on a Hierarchy

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

- Casting on a hierarchy is permitted in a limited sense

```
class A {};
class B : public A {};

A *pa = 0;
B *pb = 0;
void *pv = 0;

pa = pb; // okay ----- // UPCAST

pb = pa; // error C2440: '=' : cannot convert from 'A *' to 'B *' // DOWNCASE

pv = pa; // okay ----- // Lose the type
pv = pb; // okay ----- // Lose the type

pa = pv; // error C2440: '=' : cannot convert from 'void *' to 'A *'
pb = pv; // error C2440: '=' : cannot convert from 'void *' to 'B *'
```



# Casting on a Hierarchy

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

## Up-Casting is safe

```
class A { public: int dataA_; };
class B : public A { public: int dataB_; };

A a;
B b;

a.dataA_ = 2;
b.dataA_ = 3;
b.dataB_ = 5;

A *pa = &a;
B *pb = &b;

cout << pa->dataA_ << endl; // prints 2
cout << pb->dataA_ << " " << pb->dataB_ << endl; // prints 3 5

pa = &b;

cout << pa->dataA_ << endl; // prints 3
// cout << pa->dataB_ << endl; // error C2039: 'dataB_' : is not a member of 'A'
```



# Static and Dynamic Binding

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

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

class B {
public:
 void f() { cout << "B::f()" << endl; }
 virtual void g() { cout << "B::g()" << endl; }
};

class D: public B {
public:
 void f() { cout << "D::f()" << endl; }
 virtual void g() { cout << "D::g()" << endl; }
};

int main() {
 B b;
 D d;

 B *pb = &b;
 B *pd = &d; // UPCAST

 B &rb = b;
 B &rd = d; // UPCAST

 b.f(); // B::f()
 b.g(); // B::g()
 d.f(); // D::f()
 d.g(); // D::g()

 pb->f(); // B::f() -- Static Binding
 pb->g(); // B::g() -- Dynamic Binding
 pd->f(); // B::f() -- Static Binding
 pd->g(); // D::g() -- Dynamic Binding

 rb.f(); // B::f() -- Static Binding
 rb.g(); // B::g() -- Dynamic Binding
 rd.f(); // B::f() -- Static Binding
 rd.g(); // D::g() -- Dynamic Binding

 return 0;
}
```



# Module Summary

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

- Introduced casting and discussed the notions of upcast and downcast
- Introduced Static and Dynamic Binding



# Instructor and TAs

Module 26

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Static and  
Dynamic  
Binding

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, TA                    | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, TA                 | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA             | himadribhuyan@gmail.com   | 9438911655 |



Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types  
Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

# Module 27: Programming in C++

## Dynamic Binding (Polymorphism): Part 2

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types  
Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

- Understand Static and Dynamic Binding
- Understand Polymorphic Type



# Module Outline

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types  
Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

- Binding
  - Types
  - Static Binding
  - Dynamic Binding
- Polymorphic Type



# Type of an Object

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types

Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

- The static type of the object is the type declared for the object while writing the code
- Compiler sees static type
- The dynamic type of the object is determined by the type of the object to which it currently refers
- Compiler does not see dynamic type

```
class A {};
class B : public A {};

int main() {
 A *p;
 p = new B; // Static type of p = A
 // Dynamic type of p = B
}
```



# Static and Dynamic Binding

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding

Types

Static Binding

Dynamic

Binding

Polymorphic  
Type

Summary

- **Static binding (early binding)**: When a function invocation binds to the function definition based on the static type of objects
  - This is done at compile-time
  - Normal function calls, overloaded function calls, and overloaded operators are examples of static binding
- **Dynamic binding (late binding)**: When a function invocation binds to the function definition based on the dynamic type of objects
  - This is done at run-time
  - Function pointers, Virtual functions are examples of late binding



# Static Binding

## Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding

Types

Static Binding

Dynamic  
Binding

Polymorphic  
Type

Summary

|  | Inherited Method                                                                                                                                                                                                                                                                              | Overridden Method                                                                                                                                                                                                                                                                                                                                                    |
|--|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <pre>#include&lt;iostream&gt; using namespace std; class B { public:     void f() {}  };  class D : public B { public:     void g() {} // new function };  int main() {     B b;     D d;      b.f(); // B::f()     d.f(); // B::f() ----- Inherited     d.g(); // D::g() ----- Added }</pre> | <pre>#include&lt;iostream&gt; using namespace std; class B { public:     void f() {}  };  class D : public B { public:     void f() {}  };  int main() {     B b;     D d;      b.f(); // B::f()     d.f(); // D::f() ----- Overridden             // masks the base class function }</pre>                                                                          |
|  | <ul style="list-style-type: none"><li>Object d of derived class inherits the base class function f() and has its own function g()</li><li>Function calls are resolved at compile time based on static type</li></ul>                                                                          | <p>If a member function of a base class is redefined in a derived class with the same signature then it masks the base class method</p> <ul style="list-style-type: none"><li>The derived class method f() is linked to the object d. As f() is redefined in the derived class, the base class version cannot be called with the object of a derived class</li></ul> |



# Member Functions – Overrides and Overloads: RECAP (Module 22)

## Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types  
Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

### Inheritance

```
class B { // Base Class
public:
 void f(int i);
 void g(int i);
};

class D: public B { // Derived Class
public:
 // Inherits B::f(int)
 // Inherits B::g(int)

};

B b;
D d;

b.f(1); // Calls B::f(int)
b.g(2); // Calls B::g(int)

d.f(3); // Calls B::f(int)
d.g(4); // Calls B::g(int)
```

### Override & Overload

```
class B { // Base Class
public:
 void f(int);
 void g(int i);
};

class D: public B { // Derived Class
public:
 // Inherits B::f(int)
 void f(int); // Overrides B::f(int)
 void f(string&); // Overloads B::f(int)
 // Inherits B::g(int)
 void h(int i); // Adds D::h(int)

};

B b;
D d;

b.f(1); // Calls B::f(int)
b.g(2); // Calls B::g(int)

d.f(3); // Calls D::f(int)
d.g(4); // Calls B::g(int)

d.f("red"); // Calls D::f(string&)
d.h(5); // Calls D::h(int)
```

- D::f(int) overrides B::f(int)
- D::f(string) overloads B::f(int)



# using Construct – Avoid Method Hiding

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding

Types

Static Binding

Dynamic

Binding

Polymorphic  
Type

Summary

```
#include<iostream>
using namespace std;
class A { public:
 void f() {}}
;

class B : public A {
 // To overload, rather than hide the base class function f()
 // is introduced into the scope of B with a using declaration
 using A::f;
 void f(int) { }
};

int main() {
 B b; // function calls resolved at compile time
 b.f(3); // B::f(int)
 b.f(); // A::f()
}
```

- Object b of derived class linked to with inherited base class function f() and the overloaded version defined by the derived class f(int), based on the input parameters – function calls resolved at compile time



# Dynamic Binding

## Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding

Types

Static Binding

Dynamic

Binding

Polymorphic  
Type

Summary

### Non-Virtual Method

```
#include<iostream>
using namespace std;
class B { public:
 void f() { }
};
class D : public B { public:
 void f() { }
};
int main() {
 B b;
 D d;

 B *p;

 p = &b; p->f(); // B::f()
 p = &d; p->f(); // B::f()
}
```

### Virtual Method

```
#include<iostream>
using namespace std;
class B { public:
 virtual void f() { }
};
class D : public B { public:
 virtual void f() { }
};
int main() {
 B b;
 D d;

 B *p;

 p = &b; p->f(); // B::f()
 p = &d; p->f(); // D::f()
}
```

- `p->f()` always binds to `B::f()`
- Binding is decided by the type of pointer
- **Static Binding**

- `p->f()` binds to `B::f()` for a `B` object, and to `D::f()` for a `D` object
- Binding is decided by the type of object
- **Dynamic Binding**



# Static and Dynamic Binding: RECAP (Module 26)

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types  
Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

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

class B {
public:
 void f() { cout << "B::f()" << endl; }
 virtual void g() { cout << "B::g()" << endl; }
};

class D: public B {
public:
 void f() { cout << "D::f()" << endl; }
 virtual void g() { cout << "D::g()" << endl; }
};

int main() {
 B b;
 D d;

 B *pb = &b;
 B *pd = &d; // UPCAST

 B &rb = b;
 B &rd = d; // UPCAST

 b.f(); // B::f()
 b.g(); // B::g()
 d.f(); // D::f()
 d.g(); // D::g()

 pb->f(); // B::f() -- Static Binding
 pb->g(); // B::g() -- Dynamic Binding
 pd->f(); // B::f() -- Static Binding
 pd->g(); // D::g() -- Dynamic Binding

 rb.f(); // B::f() -- Static Binding
 rb.g(); // B::g() -- Dynamic Binding
 rd.f(); // B::f() -- Static Binding
 rd.g(); // D::g() -- Dynamic Binding

 return 0;
}
```



# Polymorphic Type: Virtual Functions

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types  
Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

- Dynamic binding is possible only for pointer and reference data types and for member functions that are declared as virtual in the base class.
- These are called **Virtual Functions**
- If a member function is declared as virtual, it can be overridden in the derived class
- If a member function is not virtual and it is re-defined in the derived class then the latter definition hides the former one
- Any class containing a virtual member function – by definition or by inheritance – is called a **Polymorphic Type**
- A hierarchy may be polymorphic or non-polymorphic
- A non-polymorphic hierarchy has little value



# Polymorphism Rule

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types  
Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

```
#include <iostream>
using namespace std;
class A { public:
 void f() { cout << "A::f()" << endl; } // Non-Virtual
 virtual void g() { cout << "A::g()" << endl; } // Virtual
 void h() { cout << "A::h()" << endl; } // Non-Virtual
};
class B : public A { public:
 void f() { cout << "B::f()" << endl; } // Non-Virtual
 void g() { cout << "B::g()" << endl; } // Virtual
 virtual void h() { cout << "B::h()" << endl; } // Virtual
};
class C : public B { public:
 void f() { cout << "C::f()" << endl; } // Non-Virtual
 void g() { cout << "C::g()" << endl; } // Virtual
 void h() { cout << "C::h()" << endl; } // Virtual
};
```

|                                                                                                                                                             |                                                      |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|
| <pre>int main() { B *q = new C; A *p = q;     p-&gt;f();     p-&gt;g();     p-&gt;h();      q-&gt;f();     q-&gt;g();     q-&gt;h();      return 0; }</pre> | <pre>A::f() C::g() A::h() B::f() C::g() C::h()</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|



# Module Summary

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding

Types

Static Binding

Dynamic

Binding

Polymorphic  
Type

Summary

- Static and Dynamic Binding are discussed in depth
- Polymorphic type introduced



# Instructor and TAs

Module 27

Partha Pratim  
Das

Objectives &  
Outline

Binding  
Types  
Static Binding  
Dynamic  
Binding

Polymorphic  
Type

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, TA                    | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, TA                 | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA             | himadribhuyan@gmail.com   | 9438911655 |



Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

# Module 28: Programming in C++

## Dynamic Binding (Polymorphism): Part 3

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

- Understand why destructor must be virtual in a class hierarchy
- Learn to work with class hierarchy



# Module Outline

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

- Virtual Destructor
- Pure Virtual Function
- Abstract Base Class



# Virtual Destructor

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

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

class B {
 int data_;
public:
 B(int d) :data_(d) { cout << "B()" << endl; }
 ~B() { cout << "~B()" << endl; }
 virtual void Print() { cout << data_; }
};

class D: public B {
 int *ptr_;
public:
 D(int d1, int d2) :B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
 ~D() { cout << "~D()" << endl; delete ptr_; }
 void Print() { B::Print(); cout << " " << *ptr_; }
};

int main() {
 B *p = new B(2);
 B *q = new D(3, 5);

 p->Print(); cout << endl;
 q->Print(); cout << endl;

 delete p;
 delete q;

 return 0;
}
```

Output:

B()  
B()  
D()  
2  
3 5  
~B()  
~B()

Destructor of d (type D) not called!



# Virtual Destructor

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

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

class B {
 int data_;
public:
 B(int d) :data_(d) { cout << "B()" << endl; }
 virtual ~B() { cout << "~B()" << endl; } // Destructor made virtual
 virtual void Print() { cout << data_; }
};

class D: public B {
 int *ptr_;
public:
 D(int d1, int d2) :B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
 ~D() { cout << "~D()" << endl; delete ptr_; }
 void Print() { B::Print(); cout << " " << *ptr_; }
};

int main() {
 B *p = new B(2);
 B *q = new D(3, 5);

 p->Print(); cout << endl;
 q->Print(); cout << endl;

 delete p;
 delete q;

 return 0;
}
```

Output:

B()  
B()  
D()  
2  
3 5  
~B()  
~D()  
~B()

Destructor of d (type D) is called!



# Virtual Destructor

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

- If the destructor is not virtual in a polymorphic hierarchy, it leads to **Slicing**
- **Destructor must be declared virtual in the base class**



# Hierarchy of Shapes

Module 28

Partha Pratim  
Das

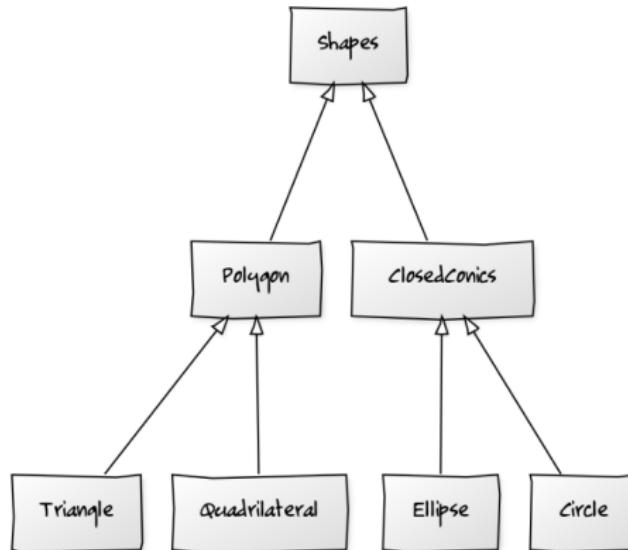
Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary



- We want to have a polymorphic `draw()` function for the hierarchy
- `draw()` will be overridden in every class based on the drawing algorithms
- What is the `draw()` function for the root `Shapes` class?



# Pure Virtual Function

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

- For the polymorphic hierarchy of Shapes, we need draw() to be a virtual function
- draw() must be a member of Shapes class for polymorphic dispatch to work
- But we cannot define the body of draw() function for the root Shapes class as we do not have an algorithm to draw an arbitrary shape. In fact, we cannot even have a representation for shapes in general!
- Pure Virtual Function** solves the problem
- A **Pure Virtual Function** has a signature but no body!



# Abstract Base Class

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

- A class containing at least one **Pure Virtual Function** is called an **Abstract Base Class**
- **Pure Virtual Functions** may be inherited or defined in the class
- No instance can be created for an **Abstract Base Class**
- Naturally it does not have a constructor or the destructor
- An **Abstract Base Class**, however, may have other virtual (non-pure) and non-virtual member functions as well as data members
- Data members in an **Abstract Base Class** should be protected. Of course, private and public data are also allowed
- Member functions in an **Abstract Base Class** should be public. Of course, private and protected methods are also allowed
- A **Concrete Class** must override and implement all **Pure Virtual Functions** so that it can be instantiated



# Shape Hierarchy

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

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

class Shapes { public: // Abstract Base Class
 virtual void draw() = 0; // Pure Virtual Function
};

class Polygon: public Shapes { // Concrete Class
 void draw() { cout << "Polygon: Draw by Triangulation" << endl; }
};

class ClosedConics : public Shapes { // Abstract Base Class
 // draw() inherited - Pure Virtual
};

class Triangle : public Polygon { public: // Concrete Class
 void draw() { cout << "Triangle: Draw by Lines" << endl; }
};

class Quadrilateral : public Polygon { public: // Concrete Class
 void draw() { cout << "Quadrilateral: Draw by Lines" << endl; }
};

class Circle : public ClosedConics { public: // Concrete Class
 void draw() { cout << "Circle: Draw by Bresenham Algorithm" << endl; }
};

class Ellipse : public ClosedConics { public: // Concrete Class
 void draw() { cout << "Ellipse: Draw by ..." << endl; }
};

int main() {
 Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };

 for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i) arr[i]->draw();
 // ...
 return 0;
}
```



# Shape Hierarchy

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

```
int main() {
 Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };

 for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i) arr[i]->draw();
 // ...
 return 0;
}

Output:

Triangle: Draw by Lines
Quadrilateral: Draw by Lines
Circle: Draw by Bresenham Algorithm
Ellipse: Draw by ...
```

- Instances for class `Shapes` and class `ClosedConics` cannot be created



# Shape Hierarchy: A Pure Virtual Function may have a body!

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

```
#include <iostream>
using namespace std;
class Shapes { public:
 virtual void draw() = 0 // Pure Virtual Function
 { cout << "Shapes: Init Brush" << endl; }
};

class Polygon: public Shapes { // Concrete Class
 void draw() { Shapes::draw(); cout << "Polygon: Draw by Triangulation" << endl; }
};

class ClosedConics : public Shapes { // Abstract Base Class
 // draw() inherited - Pure Virtual
};

class Triangle : public Polygon { public: // Concrete Class
 void draw() { Shapes::draw(); cout << "Triangle: Draw by Lines" << endl; }
};

class Quadrilateral : public Polygon { public: // Concrete Class
 void draw() { Shapes::draw(); cout << "Quadrilateral: Draw by Lines" << endl; }
};

class Circle : public ClosedConics { public: // Concrete Class
 void draw() { Shapes::draw(); cout << "Circle: Draw by Bresenham Algorithm" << endl; }
};

class Ellipse : public ClosedConics { public: // Concrete Class
 void draw() { Shapes::draw(); cout << "Ellipse: Draw by ..." << endl; }
};

int main() {
 Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };

 for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i) arr[i]->draw();
 // ...
 return 0;
}
```



# Shape Hierarchy

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

```
int main() {
 Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };

 for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i) arr[i]->draw();
 // ...
 return 0;
}

Output:

Shapes: Init Brush
Triangle: Draw by Lines
Shapes: Init Brush
Quadrilateral: Draw by Lines
Shapes: Init Brush
Circle: Draw by Bresenham Algorithm
Shapes: Init Brush
Ellipse: Draw by ...
```

- Instances for class `Shapes` and class `ClosedConics` cannot be created



# Module Summary

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

- Discussed why destructors must be virtual
- Introduced Pure Virtual Functions
- Introduced Abstract Base Class



# Instructor and TAs

Module 28

Partha Pratim  
Das

Objectives &  
Outline

Virtual  
Destructor

Pure Virtual  
Function

Abstract Base  
Class

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, TA                    | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, TA                 | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA             | himadribhuyan@gmail.com   | 9438911655 |



Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

# Module 29: Programming in C++

## Dynamic Binding (Polymorphism): Part 4

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

- Understand design with class hierarchy



# Module Outline

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

- Binding Exercise
- Staff Salary Processing
  - C Solution
  - C++ Solution
    - Non-Polymorphic Hierarchy
    - Polymorphic Hierarchy
    - Polymorphic Hierarchy (Flexible)



# Binding: Exercise

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

```
// Class Definitions
class A { public:
 virtual void f(int) { }
 virtual void g(double) { }
 int h(A *) { }
};

class B: public A { public:
 void f(int) { }
 virtual int h(B *) { }
};

class C: public B { public:
 void g(double) { }
 int h(B *) { }
};

// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

| Invocation  | Initialization |          |          |
|-------------|----------------|----------|----------|
|             | pA = &a;       | pA = &b; | pA = &c; |
| pA->f(2);   | A::f           | B::f     | C::f     |
| pA->g(3.2); | A::g           | A::g     | C::g     |
| pA->h(&a);  | A::h           | A::h     | A::h     |
| pA->h(&b);  | A::h           | A::h     | A::h     |



# Binding: Exercise

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

```
// Class Definitions
class A { public:
 virtual void f(int) { }
 virtual void g(double) { }
 int h(A *) { }
};

class B: public A { public:
 void f(int) { }
 virtual int h(B *) { }
};

class C: public B { public:
 void g(double) { }
 int h(B *) { }
};
```

```
// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

| Invocation  | Initialization |                              |          |
|-------------|----------------|------------------------------|----------|
|             | pB = &a;       | pB = &b;                     | pB = &c; |
| pB->f(2);   | Error          | B::f                         | B::f     |
| pB->g(3.2); | Downcast       | A::g                         | C::g     |
| pB->h(&a);  | (A *) to       | No conversion (A *) to (B *) |          |
| pB->h(&b);  | (B *)          | B::h                         | C::h     |



# Staff Salary Processing: Problem Statement

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing

C Solution

Summary

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where **Engineers** and **Managers** work. Every **Engineer** reports to some **Manager**. Every **Manager** can also work like an **Engineer**
- The logic for processing salary for **Engineers** and **Managers** are different as they have different salary heads
- In future, it may add **Directors** to the team. Then every **Manager** will report to some **Director**. Every **Director** could also work like a **Manager**
- The logic for processing salary for **Directors** will also be distinct
- Further, in future it may open other divisions, like Sales division, and expand the workforce
- **Make a suitable extensible design**



# C Solution: Function Switch Engineer + Manager

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

- How to represent **Engineers** and **Managers**?
  - Collection of **structs**
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of **union**
- How to model variations in salary processing algorithms?
  - **struct-specific functions**
- How to invoke the correct algorithm for a correct employee type?
  - **Function Switch**
  - **Function Pointers**



# C Solution: Function Switch Engineer + Manager

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

```
#include <stdio.h>
#include <string.h>

typedef enum E_TYPE { Er, Mgr } E_TYPE;

typedef struct Engineer { char *name_; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
 e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(Engineer *e) {
 printf("%s: Process Salary for Engineer\n", e->name_);
}

typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
 m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(Manager *m) {
 printf("%s: Process Salary for Manager\n", m->name_);
}

typedef struct Staff { E_TYPE type_;
 union { Engineer *pE; Manager *pM; };
} Staff;
```



# C Solution: Function Switch Engineer + Manager

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

```
int main() {
 Staff allStaff[10];
 allStaff[0].type_ = Er;
 allStaff[0].pE = InitEngineer("Rohit");
 allStaff[1].type_ = Mgr;
 allStaff[1].pM = InitManager("Kamala");
 allStaff[2].type_ = Mgr;
 allStaff[2].pM = InitManager("Rajib");
 allStaff[3].type_ = Er;
 allStaff[3].pE = InitEngineer("Kavita");
 allStaff[4].type_ = Er;
 allStaff[4].pE = InitEngineer("Shambhu");

 for (int i = 0; i < 5; ++i) {
 E_TYPE t = allStaff[i].type_;
 if (t == Er) ProcessSalaryEngineer(allStaff[i].pE);
 else if (t == Mgr) ProcessSalaryManager(allStaff[i].pM);
 else printf("Invalid Staff Type\n");
 }
 return 0;
}

```

Output:

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
```



# C Solution: Function Switch Engineer + Manager + Director

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

- How to represent Engineers, Managers, and Directors?
  - Collection of structs
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of union
- How to model variations in salary processing algorithms?
  - struct-specific functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers



# C Solution: Function Switch Engineer + Manager + Director

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

```
#include <stdio.h>
#include <string.h>

typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE;

typedef struct Engineer { char *name_; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
 e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(Engineer *e) {
 printf("%s: Process Salary for Engineer\n", e->name_);
}

typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
 m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(Manager *m) {
 printf("%s: Process Salary for Manager\n", m->name_);
}

typedef struct Director { char *name_; Manager *reports_[10]; } Director;
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
 d->name_ = strdup(name); return d;
}
void ProcessSalaryDirector(Director *d) {
 printf("%s: Process Salary for Director\n", d->name_);
}

typedef struct Staff { E_TYPE type_;
 union { Engineer *pE; Manager *pM; Director *pD; };
} Staff;
```



# C Solution: Function Switch Engineer + Manager + Director

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing

C Solution

Summary

```
int main() { Staff allStaff[10];
 allStaff[0].type_ = Er;
 allStaff[0].pE = InitEngineer("Rohit");
 allStaff[1].type_ = Mgr;
 allStaff[1].pM = InitManager("Kamala");
 allStaff[2].type_ = Mgr;
 allStaff[2].pM = InitManager("Rajib");
 allStaff[3].type_ = Er;
 allStaff[3].pE = InitEngineer("Kavita");
 allStaff[4].type_ = Er;
 allStaff[4].pE = InitEngineer("Shambhu");
 allStaff[5].type_ = Dir;
 allStaff[5].pD = InitDirector("Ranjana");

 for (int i = 0; i < 6; ++i) {
 E_TYPE t = allStaff[i].type_;
 if (t == Er) ProcessSalaryEngineer(allStaff[i].pE);
 else if (t == Mgr) ProcessSalaryManager(allStaff[i].pM);
 else if (t == Dir) ProcessSalaryDirector(allStaff[i].pD);
 else printf("Invalid Staff Type\n");
 }
 return 0;
}

Output:
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
NPTEL MOOCs Programming in C++
```



# C Solution: Advantages and Disadvantages

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

## Advantages:

- Solution exists!
- Code is well structured – has patterns

## Disadvantages:

- Employee data has scope for better organization
  - No encapsulation for data
  - Duplication of fields across types of employees – possible to mix up types for them (say, `char *` and `string`)
  - Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
- Types of objects are managed explicitly by `E_Type`:
  - Difficult to extend the design – addition of a new type needs to:
    - Add new type code to enum `E_Type`
    - Add a new pointer field in struct `Staff` for the new type
    - Add a new case (`if-else`) based on the new type
  - Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.)

## Recommendation:

- Use classes for encapsulation on a hierarchy



# Module Summary

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

- Practiced exercise with binding – various mixed cases
- Started designing for a staff salary problem and worked out C solutions



# Instructor and TAs

Module 29

Partha Pratim  
Das

Objectives &  
Outline

Binding:  
Exercise

Staff Salary  
Processing  
C Solution

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, TA                    | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, TA                 | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA             | himadribhuyan@gmail.com   | 9438911655 |



Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

# Module 30: Programming in C++

## Dynamic Binding (Polymorphism): Part 5

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

- Understand design with class hierarchy



# Module Outline

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary

## ● Staff Salary Processing

- C Solution
- C++ Solution
  - Non-Polymorphic Hierarchy
  - Polymorphic Hierarchy
  - Polymorphic Hierarchy (Flexible)



# Staff Salary Processing: Problem Statement: RECAP (Module 29)

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where **Engineers** and **Managers** work. Every **Engineer** reports to some **Manager**. Every **Manager** can also work like an **Engineer**
- The logic for processing salary for **Engineers** and **Managers** are different as they have different salary heads
- In future, it may add **Directors** to the team. Then every **Manager** will report to some **Director**. Every **Director** could also work like a **Manager**
- The logic for processing salary for **Directors** will also be distinct
- Further, in future it may open other divisions, like Sales division, and expand the workforce
- **Make a suitable extensible design**



# C Solution: Engineer + Manager: RECAP (Module 29)

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

- How to represent **Engineers** and **Managers**?
  - **struct**
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of union
- How to model variations in salary processing algorithms?
  - struct-specific functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers



# C Solution: Advantages and Disadvantages

## RECAP (Module 29)

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

### Advantages:

- Solution exists!
- Code is well structured – has patterns

### Disadvantages:

- Employee data has scope for better organization
  - No encapsulation for data
  - Duplication of fields across types of employees – possible to mix up types for them (say, `char *` and `string`)
  - Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
- Types of objects are managed explicitly by `E_Type`:
  - Difficult to extend the design – addition of a new type needs to:
    - Add new type code to enum `E_Type`
    - Add a new pointer field in struct `Staff` for the new type
    - Add a new case (`if-else`) based on the new type
  - Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.)

### Recommendation:

- Use classes for encapsulation on a hierarchy



# C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager

Module 30

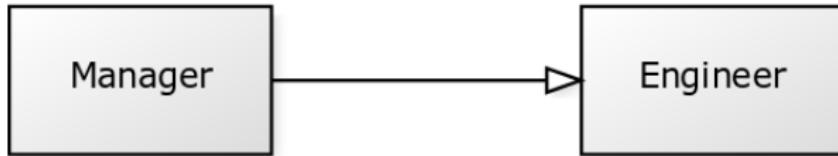
Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary



- How to represent Engineers and Managers?
  - Non-Polymorphic class hierarchy
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - array of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers



# C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

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

typedef enum E_TYPE { Er, Mgr };
class Engineer { protected: string name_; E_TYPE type_;
public: Engineer(const string& name, E_TYPE e = Er) : name_(name), type_(e) {}
 E_TYPE GetType() { return type_; }
 void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer { Engineer *reports_[10];
public: Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
 void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

int main() { Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
 Manager m1("Kamala"), m2("Rajib");
 Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3 };

 for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
 E_TYPE t = staff[i]->GetType();
 if (t == Er) staff[i]->ProcessSalary();
 else if (t == Mgr) ((Manager *)staff[i])->ProcessSalary();
 else cout << "Invalid Staff Type" << endl;
 }
 return 0;
}
```



# C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary

```
Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
Manager m1("Kamala"), m2("Rajib");
Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3 };
```

Output:

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
```



# C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary



- How to represent Engineers, Managers, and Directors?
  - Non-Polymorphic class hierarchy
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - array of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers



# C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary

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

typedef enum E_TYPE { Er, Mgr, Dir };
class Engineer { protected: string name_; E_TYPE type_;
public: Engineer(const string& name, E_TYPE e = Er) : name_(name), type_(e) {}
 E_TYPE GetType() { return type_; }
 void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer { Engineer *reports_[10];
public: Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
 void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager { Manager *reports_[10];
public: Director(const string& name) : Manager(name, Dir) {}
 void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};

int main() { Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
 Manager m1("Kamala"), m2("Rajib"); Director d("Ranjana");
 Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

 for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
 E_TYPE t = staff[i]->GetType();
 if (t == Er) staff[i]->ProcessSalary();
 else if (t == Mgr) ((Manager *)staff[i])->ProcessSalary();
 else if (t == Dir) ((Director *)staff[i])->ProcessSalary();
 else cout << "Invalid Staff Type" << endl;
 }
 return 0;
}
```



# C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary

```
Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
Manager m1("Kamala"), m2("Rajib"); Director d("Ranjana");
Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };
```

Output:

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
```



# C++ Solution: Non-Polymorphic Hierarchy: Advantages and Disadvantages

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

## Advantages:

- Data is encapsulated
- Hierarchy factors common data members
- Constructor / Destructor to manage lifetime
- struct-specific functions made member function (overridden)
- E-Type subsumed in class – no need for union
- Code reuse evidenced

## Disadvantages:

- Types of objects are managed explicitly by E-Type:
  - Difficult to extend the design – addition of a new type needs to:
    - Add new type code to enum E-Type
    - Application code need to have a new case (if-else) based on the new type
  - Error prone because the application programmer has to cast to right type to call ProcessSalary

## Recommendation:

- Use a polymorphic hierarchy with dynamic dispatch



# C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary



- How to represent Engineers, Managers, and Directors?
  - Polymorphic class hierarchy
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - array of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Virtual Functions



# C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

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

class Engineer { protected: string name_;
public: Engineer(const string& name) : name_(name) {}
 virtual void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer { Engineer *reports_[10];
public: Manager(const string& name) : Engineer(name) {}
 void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager { Manager *reports_[10];
public: Director(const string& name) : Manager(name) {}
 void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};

int main() { Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
 Manager m1("Kamala"), m2("Rajib"); Director d("Ranjana");
 Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

 for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) staff[i]->ProcessSalary();

 return 0;
}
```



# C++ Solution: Polymorphic Hierarchy Engineer + Manager + Director

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary

```
Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
Manager m1("Kamala"), m2("Rajib"); Director d("Ranjana");
Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };
```

Output:

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
```



# C++ Solution: Polymorphic Hierarchy: Advantages and Disadvantages

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary

## Advantages:

- Data is fully encapsulated
- Polymorphic Hierarchy removes the need for explicit E-Type
- Application code is independent of types in the system (virtual functions manage types through polymorphic dispatch)
- High Code reuse – code is short and simple

## Disadvantages:

- Difficult to add an employee type that is not a part of this hierarchy (for example, employees of *Sales Division*)

## Recommendation:

- Use an abstract base class for employees



# C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary



- How to represent Engineers, Managers, Directors, etc.?
  - Polymorphic class hierarchy with an Abstract Base Employee
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - array of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Virtual Functions (Pure in Employee)



# C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

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

class Employee { protected: string name_;
public: virtual void ProcessSalary() = 0;
};

class Engineer: public Employee { public: Engineer(const string& name) { name_ = name; }
void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer { Manager *reports_[10];
public: Manager(const string& name) : Engineer(name) {}
void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager { Manager *reports_[10];
public: Director(const string& name) : Manager(name) {}
void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};

class SalesExecutive : public Employee { public:
SalesExecutive(const string& name) { name_ = name; }
void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};

int main() {
 Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
 Manager m1("Kamala"), m2("Rajib"); SalesExecutive s1("Hari"), s2("Bishnu");
 Director d("Ranjana");
 Employee *staff[] = { &e1, &m1, &m2, &e2, &s1, &e3, &d, &s2 };

 for (int i = 0; i < sizeof(staff) / sizeof(Employee*); ++i) staff[i]->ProcessSalary();
 return 0;
}
```



# C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

```
Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
Manager m1("Kamala"), m2("Rajib"); SalesExecutive s1("Hari"), s2("Bishnu");
Director d("Ranjana");
Employee *staff[] = { &e1, &m1, &m2, &e2, &s1, &e3, &d, &s2 };
```

Output:

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Hari: Process Salary for Sales Executive
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
Bishnu: Process Salary for Sales Executive
```



# C++ Solution: Polymorphic Hierarchy (Flexible): Advantages and Disadvantages

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary

## Advantages:

- Data is fully encapsulated
- Flexible Polymorphic Hierarchy makes addition of any class possible on the hierarchy
- Application code is independent of types in the system (virtual functions manage types through polymorphic dispatch)
- Maximum Code reuse – code is short and simple

## Disadvantages:

- None in particular

## Recommendation:

- Enjoy the solution



# Module Summary

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution  
Non-  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
Polymorphic  
Hierarchy  
(Flexible)

Summary

- Completed design for a staff salary problem using hierarchy and worked out extensible C++ solution



# Instructor and TAs

Module 30

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Non-  
Polymorphic  
Hierarchy

Polymorphic  
Hierarchy

Polymorphic  
Hierarchy  
(Flexible)

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

# Module 31: Programming in C++

## Virtual Function Table

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

- Understand Virtual Function Table for dynamic binding (polymorphic dispatch)



# Module Outline

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

- Staff Salary Processing: RECAP
  - C Solution using Function Pointers
  - C++ Solution using Polymorphic Hierarchy
  - Comparison of C and C++ Solutions
- Virtual Function Table for Polymorphic Dispatch



# Staff Salary Processing: Problem Statement: RECAP (Module 29)

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where **Engineers** and **Managers** work. Every **Engineer** reports to some **Manager**. Every **Manager** can also work like an **Engineer**
- The logic for processing salary for **Engineers** and **Managers** are different as they have different salary heads
- In future, it may add **Directors** to the team. Then every **Manager** will report to some **Director**. Every **Director** could also work like a **Manager**
- The logic for processing salary for **Directors** will also be distinct
- Further, in future it may open other divisions, like Sales division, and expand the workforce
- **Make a suitable extensible design**



# C Solution: Function Pointers

## Engineer + Manager: RECAP (Module 29)

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

- How to represent Engineers, Managers, and Directors?
  - struct
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of union
- How to model variations in salary processing algorithms?
  - struct-specific functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers



# C Solution: Function Pointers

## Engineer + Manager + Director

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

```
#include <stdio.h>
#include <string.h>

typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE;
typedef void (*psFuncPtr)(void *);

typedef struct Engineer { char *name_; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
 e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(void *v) { Engineer *e = (Engineer *)v;
 printf("%s: Process Salary for Engineer\n", e->name_);
}

typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
 m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(void *v) { Manager *m = (Manager *)v;
 printf("%s: Process Salary for Manager\n", m->name_);
}

typedef struct Director { char *name_; Manager *reports_[10]; } Director;
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
 d->name_ = strdup(name); return d;
}
void ProcessSalaryDirector(void *v) { Director *d = (Director *)v;
 printf("%s: Process Salary for Director\n", d->name_);
}
```



# C Solution: Function Pointers

## Engineer + Manager + Director

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

```
typedef struct Staff {
 E_TYPE type_;
 void *p;
} Staff;
int main() {
 psFuncPtr psArray[] = { ProcessSalaryEngineer,
 ProcessSalaryManager,
 ProcessSalaryDirector };

 Staff staff[] = { { Er, InitEngineer("Rohit") },
 { Mgr, InitEngineer("Kamala") },
 { Mgr, InitEngineer("Rajib") },
 { Er, InitEngineer("Kavita") },
 { Er, InitEngineer("Shambhu") },
 { Dir, InitEngineer("Ranjana") } };

 for (int i = 0; i < sizeof(staff) / sizeof(Staff); ++i)
 psArray[staff[i].type_](staff[i].p);

 return 0;
}

Output:
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
```



# C++ Solution: Polymorphic Hierarchy: RECAP

## Engineer + Manager + Director: (Module 30)

Module 31

Partha Pratim  
Das

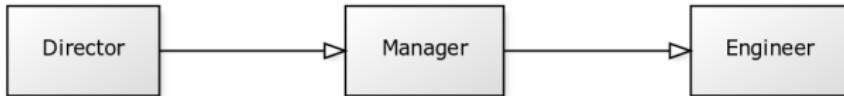
Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary



- How to represent Engineers, Managers, and Directors?
  - Polymorphic class hierarchy
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - array of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Virtual Functions



# C++ Solution: Polymorphic Hierarchy: RECAP

## Engineer + Manager + Director: (Module 30)

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

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

class Engineer {
protected:
 string name_;
public:
 Engineer(const string& name) : name_(name) {}
 virtual void ProcessSalary()
 { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer {
 Engineer *reports_[10];
public:
 Manager(const string& name) : Engineer(name) {}
 void ProcessSalary()
 { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager {
 Manager *reports_[10];
public:
 Director(const string& name) : Manager(name) {}
 void ProcessSalary()
 { cout << name_ << ": Process Salary for Director" << endl; }
};
```



# C++ Solution: Polymorphic Hierarchy: RECAP

## Engineer + Manager + Director: (Module 30)

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

```
int main() {
 Engineer e1("Rohit");
 Engineer e2("Kavita");
 Engineer e3("Shambhu");
 Manager m1("Kamala");
 Manager m2("Rajib");
 Director d("Ranjana");

 Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

 for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) staff[i]->ProcessSalary();

 return 0;
}
```

Output:

Rohit: Process Salary for Engineer  
Kamala: Process Salary for Manager  
Rajib: Process Salary for Manager  
Kavita: Process Salary for Engineer  
Shambhu: Process Salary for Engineer  
Ranjana: Process Salary for Director



# C and C++ Solutions: A Comparison

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

## C Solution

- How to represent Engineers, Managers, and Directors?
  - structs
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - array of union wrappers
- How to model variations in salary processing algorithms?
  - functions for structs
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - Function pointers

## C++ Solution

- How to represent Engineers, Managers, and Directors?
  - Polymorphic hierarchy
- How to initialize objects?
  - Ctor / Dtor
- How to have a collection of mixed objects?
  - array of base class pointers
- How to model variations in salary processing algorithms?
  - class member functions
- How to invoke the correct algorithm for a correct employee type?
  - Virtual Functions



# C and C++ Solutions: A Comparison

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

## C Solution (Function Pointer)

```
#include <stdio.h>
#include <string.h>
typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE;
typedef void (*psFuncPtr)(void *);
typedef struct { E_TYPE type_; void *p; } Staff;

typedef struct { char *name_ ; } Engineer;
Engineer *InitEngineer(const char *name);
void ProcessSalaryEngineer(void *v);
typedef struct { char *name_ ; } Manager;
Manager *InitManager(const char *name);
void ProcessSalaryManager(void *v);
typedef struct { char *name_ ; } Director;
Director *InitDirector(const char *name);
void ProcessSalaryDirector(void *v);
int main() { psFuncPtr psArray[] = {
 ProcessSalaryEngineer,
 ProcessSalaryManager,
 ProcessSalaryDirector };

 Staff staff[] = {
 { Er, InitEngineer("Rohit") },
 { Mgr, InitEngineer("Kamala") },
 { Dir, InitEngineer("Ranjana") } };

 for (int i = 0; i <
 sizeof(staff)/sizeof(Staff); ++i)
 psArray[staff[i].type_](staff[i].p);
 return 0;
}
```

## C++ Solution (Virtual Function)

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

class Engineer { protected: string name_;
public: Engineer(const string& name);
 virtual void ProcessSalary(); };
class Manager : public Engineer {
public: Manager(const string& name);
 void ProcessSalary(); };
class Director : public Manager {
public: Director(const string& name);
 void ProcessSalary(); };
int main() {

 Engineer e1("Rohit");
 Manager m1("Kamala");
 Director d("Ranjana");
 Engineer *staff[] = { &e1, &m1, &d };

 for(int i = 0; i <
 sizeof(staff)/sizeof(Engineer*); ++i)
 staff[i]->ProcessSalary();
 return 0;
}
```



# Virtual Function Pointer Table

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

```
class B {
 int i;
public:
 B(int i_): i(i_) {}
 void f(int); // B::f(B*const, int)
 virtual void g(int); // B::g(B*const, int)
}

B b(100);

B *p = &b;
```

b Object Layout

| Object | VFT |
|--------|-----|
| vft    | → 0 |
| B::i   | 100 |

**Source Expression**  
b.f(15);

p->f(25);

b.g(35);

p->g(45);

**Compiled Expression**  
B::f(&b, 15);

B::f(p, 25);

B::g(&b, 35);

p->vft[0](p, 45);

```
class D: public B {
 int j;
public:
 D(int i_, int j_): B(i_), j(j_) {}
 void f(int); // D::f(D*const, int)
 void g(int); // D::g(D*const, int)
}

D d(200, 500);

B *p = &d;
```

d Object Layout

| Object | VFT |
|--------|-----|
| vft    | → 0 |
| B::i   | 200 |
| D::j   | 500 |

**Source Expression**  
d.f(15);

p->f(25);

d.g(35);

p->g(45);

**Compiled Expression**  
D::f(&d, 15);

D::f(p, 25);

D::g(&d, 35);

p->vft[0](p, 45);



# Virtual Function Pointer Table

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

- Whenever a class defines a virtual function a hidden member variable is added to the class which points to an array of pointers to (virtual) functions called the **Virtual Function Table (VFT)**
- VFT pointers are used at run-time to invoke the appropriate function implementations, because at compile time it may not yet be known if the base function is to be called or a derived one implemented by a class that inherits from the base class
- VFT is class-specific – all instances of the class has the same VFT
- VFT carries the **Run-Time Type Information (RTTI)** of objects



# Virtual Function Pointer Table

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

```
class A { public:
 virtual void f(int) { }
 virtual void g(double) { }
 int h(A *) { }
};

class B: public A { public:
 void f(int) { }
 virtual int h(B *) { }
};

class C: public B { public:
 void g(double) { }
 int h(B *) { }
};

A a; B b; C c;

A *pA; B *pB;
```

## Source Expression

```
pA->f(2);
pA->g(3.2);
pA->h(&a);
pA->h(&b);
```

```
pB->f(2);
pB->g(3.2);
pB->h(&a);
pB->h(&b);
```

## Compiled Expression

```
pA->vft[0](pA, 2);
pA->vft[1](pA, 3.2);
A::h(pA, &a);
A::h(pA, &b);
```

```
pB->vft[0](pB, 2);
pB->vft[1](pB, 3.2);
pB->vft[2](pB, &a);
pB->vft[2](pB, &b);
```

## a Object Layout

| Object | VFT                                             |
|--------|-------------------------------------------------|
| vft    | →                                               |
|        | 0 A::f(A*const, int)<br>1 A::g(A*const, double) |

## b Object Layout

| Object | VFT                                             |
|--------|-------------------------------------------------|
| vft    | →                                               |
|        | 0 B::f(B*const, int)<br>1 A::g(A*const, double) |
|        | 2 B::h(B*const, B*)                             |

## c Object Layout

| Object | VFT                                             |
|--------|-------------------------------------------------|
| vft    | →                                               |
|        | 0 B::f(B*const, int)<br>1 C::g(C*const, double) |
|        | 2 C::h(C*const, B*)                             |



# Module Summary

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing  
C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on virtual function tables



# Instructor and TAs

Module 31

Partha Pratim  
Das

Objectives &  
Outline

Staff Salary  
Processing

C Solution  
C++ Solution

Virtual  
Function  
Pointer Table

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators  
`const_cast`  
Summary

# Module 32: Programming in C++

## Type Casting & Cast Operators: Part 1

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators  
`const_cast`

Summary

- Understand casting in C and C++



# Module Outline

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators  
`const_cast`

Summary

- Casting: C-Style: RECAP
  - Upcast & Downcast
- Cast Operators in C++
  - `const_cast` Operator
  - `static_cast` Operator
  - `reinterpret_cast` Operator
  - `dynamic_cast` Operator
- `typeid` Operator



# Type Casting

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

- Why casting?
  - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
  - The standard C++ conversions and user-defined conversions
- Explicit conversions
  - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
  - Allocates temporary storage
  - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }
```

```
// compiler generates:
double f (int i, int j) {
 double temp_i = i, temp_j = j; // Conversion in temporary
 return temp_i / temp_j;
}
```



# Casting: C-Style: RECAP (Module 26)

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

- Casting is performed when a value (variable) of one type is used in place of some other type

```
int i = 3;
double d = 2.5;

double result = d / i; // i is cast to double and used
```

- Casting can be implicit or explicit

```
int i = 3;
double d = 2.5;

double *p = &d;

d = i; // implicit

i = d; // implicit -- // warning C4244: '=' : conversion from 'double' to 'int',
 // possible loss of data
i = (int)d; // explicit

i = p; // error C2440: '=' : cannot convert from 'double *' to 'int'
i = (int)p; // explicit
```



# Casting: C-Style: RECAP (Module 26)

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

- (Implicit) Casting between unrelated classes is not permitted

```
class A { int i; };
class B { double d; };

A a;
B b;

A *p = &a;
B *q = &b;

a = b; // error C2679: binary '=' : no operator found
 // which takes a right-hand operand of type 'main::B'

a = (A)b; // error C2440: 'type cast' : cannot convert from 'main::B' to 'main::A'

b = a; // error C2679: binary '=' : no operator found
 // which takes a right-hand operand of type 'main::A'

b = (B)a; // error C2440: 'type cast' : cannot convert from 'main::A' to 'main::B'

p = q; // error C2440: '=' : cannot convert from 'main::B *' to 'main::A *'

q = p; // error C2440: '=' : cannot convert from 'main::A *' to 'main::B *'

p = (A*)&b; // Forced -- Okay
q = (B*)&a; // Forced -- Okay
```



# Casting: C-Style: RECAP (Module 26)

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

- Forced Casting between unrelated classes is dangerous

```
class A { public: int i; };
class B { public: double d; };

A a;
B b;

a.i = 5;
b.d = 7.2;

A *p = &a;
B *q = &b;

cout << p->i << endl; // prints 5
cout << q->d << endl; // prints 7.2

p = (A*)&b;
q = (B*)&a;

cout << p->i << endl; // prints -858993459 ----- GARBAGE
cout << q->d << endl; // prints -9.25596e+061 ----- GARBAGE
```



# Casting on a Hierarchy: C-Style: RECAP (Module 26)

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

- Casting on a hierarchy is permitted in a limited sense

```
class A {};
class B : public A {};

A *pa = 0;
B *pb = 0;
void *pv = 0;

pa = pb; // okay ----- // UPCAST

pb = pa; // error C2440: '=' : cannot convert from 'A *' to 'B *' // DOWNCASE

pv = pa; // okay ----- // Lose the type
pv = pb; // okay ----- // Lose the type

pa = pv; // error C2440: '=' : cannot convert from 'void *' to 'A *'
pb = pv; // error C2440: '=' : cannot convert from 'void *' to 'B *'
```



# Casting on a Hierarchy: C-Style: RECAP (Module 26)

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

## Up-Casting is safe

```
class A { public: int dataA_; };
class B : public A { public: int dataB_; };

A a;
B b;

a.dataA_ = 2;
b.dataA_ = 3;
b.dataB_ = 5;

A *pa = &a;
B *pb = &b;

cout << pa->dataA_ << endl; // prints 2
cout << pb->dataA_ << " " << pb->dataB_ << endl; // prints 3 5

pa = &b;

cout << pa->dataA_ << endl; // prints 3
// cout << pa->dataB_ << endl; // error C2039: 'dataB_' : is not a member of 'A'
```



# Casting in C and C++

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Cast  
Operators

const\_cast

Summary

## ● Casting in C

- Implicit cast
- Explicit C-Style cast
- Loses type information in several contexts
- Lacks clarity of semantics

## ● Casting in C++

- Performs fresh inference of types without change of value
- Performs fresh inference of types with change of value
  - Using implicit computation
  - Using explicit (user-defined) computation
- Preserves type information in all contexts
- Provides clear semantics through cast operators:
  - const\_cast
  - static\_cast
  - reinterpret\_cast
  - dynamic\_cast
- Cast operators can be grep-ed in source
- C-Style cast must be avoided in C++



# Cast Operators

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators

const\_cast

Summary

- A **cast operator** take an expression of **source type** (implicit from the expression) and convert it to an expression of **target type** (explicit in the operator) following the **semantics of the operator**
- Use of cast operators increases robustness by generating errors in **static** or **dynamic** time



# Cast Operators

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators

`const_cast`

Summary

- **`const_cast` operator:** `const_cast<type>(expr)`
  - Explicitly overrides `const` and/or `volatile` in a cast
  - Usually does not perform computation or change value
- **`static_cast` operator:** `static_cast<type>(expr)`
  - Performs a non-polymorphic cast
  - Usually performs computation to change value – implicit or user-defined
- **`reinterpret_cast` operator:** `reinterpret_cast<type>(expr)`
  - Casts between unrelated pointer types or pointer and integer
  - Does not perform computation yet reinterprets value
- **`dynamic_cast` operator:** `dynamic_cast<type>(expr)`
  - Performs a run-time cast that verifies the validity of the cast
  - Performs pre-defined computation, sets null or throws exception



# const\_cast Operator

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators  
`const_cast`

Summary

- `const_cast` converts between types with different cv-qualification
- Only `const_cast` may be used to cast away (remove) const-ness or volatility
- Usually does not perform computation or change value



# const\_cast Operator

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Cast  
Operators

const\_cast

Summary

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

class A { int i_;
public: A(int i) : i_(i) {}
 int get() const { return i_; }
 void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
 const char * c = "sample text";
// print(c); // error: 'void print(char *)': cannot convert argument 1
// // from 'const char *' to 'char *'

 print(const_cast<char *>(c));

 const A a(1);
 a.get();

// a.set(5); // error: 'void A::set(int)': cannot convert
// // 'this' pointer from 'const A' to 'A &'

 const_cast<A&>(a).set(5);

// const_cast<A>(a).set(5); // error: 'const_cast': cannot convert
// // from 'const A' to 'A'
 return 0;
}
```



# const\_cast Operator vis-a-vis C-Style Cast

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting

Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

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

class A { int i_;
public: A(int i) : i_(i) {}
 int get() const { return i_; }
 void set(int j) { i_ = j; }
};
void print(char * str) { cout << str; }

int main() {
 const char * c = "sample text";

// print(const_cast<char *>(c));
 print((char *)(c)); // C-Style Cast

 const A a(1);

// const_cast<A&>(a).set(5);
 ((A&)a).set(5); // C-Style Cast

// const_cast<A>(a).set(5); // error: 'const_cast': cannot convert
 // from 'const A' to 'A'
 ((A)a).set(5); // C-Style Cast

 return 0;
}
```



# const\_cast Operator

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

```
#include <iostream>
using namespace std;
struct type { type() :i(3) {}
 void m1(int v) const {
 //this->i = v; // error C3490: 'i' cannot be modified because
 // it is being accessed through a const object
 const_cast<type*>(this)->i = v; // OK as long as the type object isn't const
 }
 int i;
};
int main() {
 int i = 3; // i is not declared const
 const int& cref_i = i;
 const_cast<int&>(cref_i) = 4; // OK: modifies i
 cout << "i = " << i << '\n';

 type t; // note, if this is const type t;, then t.m1(4); is undefined behavior
 t.m1(4);
 cout << "type::i = " << t.i << '\n';

 const int j = 3; // j is declared const
 int* pj = const_cast<int*>(&j);
 *pj = 4; // undefined behavior! Value of j and *pj may differ
 cout << j << " " << *pj << endl;

 void (type::*mfp)(int) const = &type::m1; // pointer to member function
 //const_cast<void(type::*)(int)>(mfp); // error C2440: 'const_cast' : cannot convert
 // from 'void (_thiscall type::*)(int) const' to 'void (_thiscall type::*)(int)'
 // const_cast does not work on function pointers
 return 0;
}
```

Output:

i = 4

type::i = 4

3 4



# Module Summary

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

- Understood casting in C and C++
- Explained cast operators in C++ and discussed the evils of C-style casting
- Studied `const_cast` with examples



# Instructor and TAs

Module 32

Partha Pratim  
Das

Objectives &  
Outline

Casting  
Upcast &  
Downcast

Cast  
Operators  
const\_cast

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, TA                    | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, TA                 | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA             | himadribhuyan@gmail.com   | 9438911655 |



Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`static_cast`  
`reinterpret_cast`

Summary

# Module 33: Programming in C++

## Type Casting & Cast Operators: Part 2

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`static_cast`  
`reinterpret_cast`

Summary

- Understand casting in C and C++



# Module Outline

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`static_cast`  
`reinterpret_cast`

Summary

- Casting: C-Style: RECAP
  - Upcast & Downcast
- Cast Operators in C++
  - `const_cast` Operator
  - `static_cast` Operator
  - `reinterpret_cast` Operator
  - `dynamic_cast` Operator
- `typeid` Operator



# Casting in C and C++

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`static_cast`  
`reinterpret_cast`

Summary

## ● Casting in C

- Implicit cast
- Explicit C-Style cast
- Loses type information in several contexts
- Lacks clarity of semantics

## ● Casting in C++

- Performs fresh inference of types without change of value
- Performs fresh inference of types with change of value
  - Using implicit computation
  - Using explicit (user-defined) computation
- Preserves type information in all contexts
- Provides clear semantics through cast operators:
  - `const_cast`
  - `static_cast`
  - `reinterpret_cast`
  - `dynamic_cast`
- Cast operators can be grep-ed in source
- C-Style cast must be avoided in C++



# static\_cast Operator

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`static_cast`  
`reinterpret_cast`

Summary

- `static_cast` performs all conversions allowed implicitly (not only those with pointers to classes), and also the opposite of these. It can:
  - Convert from `void*` to any pointer type
  - Convert integers, floating-point values and `enum` types to `enum` types
- `static_cast` can perform conversions between pointers to related classes:
  - Not only up-casts, but also down-casts
  - No checks are performed during run-time to guarantee that the object being converted is in fact a full object of the destination type
- Additionally, `static_cast` can also perform the following:
  - Explicitly call a single-argument constructor or a conversion operator
    - The User-Defined Cast
  - Convert to rvalue references
  - Convert `enum` class values into integers or floating-point values
  - Convert any type to `void`, evaluating and discarding the value



# static\_cast Operator: Built-in Types

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
~~static\_cast~~  
~~reinterpret\_cast~~

Summary

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

// Built-in Types
int main() {
 int i = 2;
 double d = 3.7;
 double *pd = &d;

 i = d; // implicit -- warning
 i = static_cast<int>(d); // static_cast -- okay
 i = (int)d; // C-style -- okay

 d = i; // implicit -- okay
 d = static_cast<double>(i); // static_cast -- okay
 d = (double)i; // C-style -- okay

 i = pd; // implicit -- error
 i = static_cast<int>(pd); // static_cast -- error
 i = (int)pd; // C-style -- okay: RISKY: Should use reinterpret_cast

 return 0;
}
```



# static\_cast Operator: Class Hierarchy

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

static\_cast  
reinterpret\_cast

Summary

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

// Class Hierarchy
class A {};
class B: public A {};

int main() {
 A a;
 B b;

 // UPCAST
 A *p = &b; // implicit -- okay
 p = static_cast<A*>(&b); // static_cast -- okay
 p = (A*)&b; // C-style -- okay

 // DOWNCAST
 q = &a; // implicit -- error
 q = static_cast<B*>(&a); // static_cast -- okay: RISKY: Should use dynamic_cast
 q = (B*)&a; // C-style -- okay

 return 0;
}
```



## static\_cast Operator: Pitfall

Module 33

## Cast Operators

```
class Window { public: virtual void onResize(); ... }
class SpecialWindow: public Window { // derived class
public:
 virtual void onResize() { // derived onResize impl;
 static_cast<Window>(*this).onResize(); // cast *this to Window,
 // then call its onResize;
 // this doesn't work!

 ... // do SpecialWindow-specific stuff
 }
 ...
};
```

Slices the object, creates a temporary and calls the method!

```
class SpecialWindow: public Window { // derived class
public:
 virtual void onResize() { // derived onResize impl;
 Window::onResize(); // Direct call works

 ... // do SpecialWindow-specific stuff
 }
 ...
};
```



# static\_cast Operator: Unrelated Classes

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`static_cast`  
`reinterpret_cast`

Summary

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

// Un-related Types
class B;
class A {
public:
};

class B { };

int main() {
 A a;
 B b;
 int i = 5;

 // B ==> A
 a = b; // error
 a = static_cast<A>(b); // error
 a = (A)b; // error

 // int ==> A
 a = i; // error
 a = static_cast<A>(i); // error
 a = (A)i; // error

 return 0;
}
```

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

// Un-related Types
class B;
class A {
public:
 A(int i = 0) { cout << "A::A(i)\n"; }
 A(const B&) { cout << "A::A(B&)\n"; }
};

class B { };

int main() {
 A a;
 B b;
 int i = 5;

 // B ==> A
 a = b; // Uses A::A(B&)
 a = static_cast<A>(b); // Uses A::A(B&)
 a = (A)b; // Uses A::A(B&)

 // int ==> A
 a = i; // Uses A::A(int)
 a = static_cast<A>(i); // Uses A::A(int)
 a = (A)i; // Uses A::A(int)

 return 0;
}
```



# static\_cast Operator: Unrelated Classes

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`static_cast`  
`reinterpret_cast`

Summary

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

// Un-related Types
class B;
class A { int i_; public:
 A(int i = 0) : i_(i)
 { cout << "A::A(i)\n"; }
 operator int()
 { cout << "A::operator int()\n"; return i_; }
};

class B { public:
 operator A()
 { cout << "B::operator A()\n"; return A(); }
};

int main() {
 A a; B b; int i = 5;

 // B ==> A
 a = b; // error
 a = static_cast<A>(b); // error
 a = (A)b; // error

 // A ==> int
 i = a; // error
 i = static_cast<int>(a); // error
 i = (int)a; // error

 return 0;
}

#include <iostream>
using namespace std;

// Un-related Types
class B;
class A { int i_; public:
 A(int i = 0) : i_(i)
 { cout << "A::A(i)\n"; }
 operator int()
 { cout << "A::operator int()\n"; return i_; }
};

class B { public:
 operator A()
 { cout << "B::operator A()\n"; return A(); }
};

int main() {
 A a; B b; int i = 5;

 // B ==> A
 a = b; // B::operator A()
 a = static_cast<A>(b); // B::operator A()
 a = (A)b; // B::operator A()

 // A ==> int
 i = a; // A::operator int()
 i = static_cast<int>(a); // A::operator int()
 i = (int)a; // A::operator int()

 return 0;
}
```



# reinterpret\_cast Operator

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`static_cast`  
`reinterpret_cast`

Summary

- `reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes
- The operation result is a simple binary copy of the value from one pointer to the other
- All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked
- It can also cast pointers to or from integer types
- The format in which this integer value represents a pointer is platform-specific
- The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as `intptr_t`), is guaranteed to be able to be cast back to a valid pointer
- The conversions that can be performed by `reinterpret_cast` but not by `static_cast` are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable



# reinterpret\_cast Operator

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

static\_cast  
reinterpret\_cast

Summary

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

class A {};
class B {};

int main() {
 int i = 2;
 double d = 3.7;
 double *pd = &d;

 i = pd; // implicit -- error
 i = reinterpret_cast<int>(pd); // reinterpret_cast -- okay
 i = (int)pd; // C-style -- okay
 cout << pd << " " << i << endl;

 A *pA;
 B *pB;

 pA = pB; // implicit -- error
 pA = reinterpret_cast<A*>(pB); // reinterpret_cast -- okay
 pA = (A*)pB; // C-style -- okay

 return 0;
}
```



# Module Summary

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

static\_cast  
reinterpret\_cast

Summary

- Studied `static_cast`, and `reinterpret_cast` with examples



# Instructor and TAs

Module 33

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

static\_cast  
reinterpret\_cast

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, TA                    | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, TA                 | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA             | himadribhuyan@gmail.com   | 9438911655 |



Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

`dynamic_cast`

`typeid`  
Operator

Summary

# Module 34: Programming in C++

## Type Casting & Cast Operators: Part 3

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

`dynamic_cast`

`typeid`  
Operator

Summary

- Understand casting in C and C++



# Module Outline

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

`dynamic_cast`

`typeid`  
Operator

Summary

- Casting: C-Style: RECAP
  - Upcast & Downcast
- Cast Operators in C++
  - `const_cast` Operator
  - `static_cast` Operator
  - `reinterpret_cast` Operator
  - `dynamic_cast` Operator
- `typeid` Operator



# Casting in C and C++

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

dynamic\_cast

typeid  
Operator

Summary

## ● Casting in C

- Implicit cast
- Explicit C-Style cast
- Loses type information in several contexts
- Lacks clarity of semantics

## ● Casting in C++

- Performs fresh inference of types without change of value
- Performs fresh inference of types with change of value
  - Using implicit computation
  - Using explicit (user-defined) computation
- Preserves type information in all contexts
- Provides clear semantics through cast operators:
  - const\_cast
  - static\_cast
  - reinterpret\_cast
  - dynamic\_cast
- Cast operators can be grep-ed in source
- C-Style cast must be avoided in C++



# dynamic\_cast Operator

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

dynamic\_cast

typeid  
Operator

Summary

- `dynamic_cast` can only be used with pointers and references to classes (or with `void*`)
- Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type
- This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an implicit conversion
- But `dynamic_cast` can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if-and-only-if the pointed object is a valid complete object of the target type
- If the pointed object is not a valid complete object of the target type, `dynamic_cast` returns a null pointer
- If `dynamic_cast` is used to convert to a reference type and the conversion is not possible, an exception of type `bad_cast` is thrown instead
- `dynamic_cast` can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a `void*` pointer



# dynamic\_cast Operator: Pointers

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

dynamic\_cast

typeid  
Operator

Summary

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

class A { public: virtual ~A() {} };
class B: public A {};
class C { public: virtual ~C() {} };

int main() {
 A a; B b; C c; A *pA; B *pB; C *pC; void *pV;

 pB = &b; pA = dynamic_cast<A*>(pB);
 cout << pB << " casts to " << pA << ": Up-cast: Valid" << endl;

 pA = &b; pB = dynamic_cast<B*>(pA);
 cout << pA << " casts to " << pB << ": Down-cast: Valid" << endl;

 pA = &a; pB = dynamic_cast<B*>(pA);
 cout << pA << " casts to " << pB << ": Down-cast: Invalid" << endl;

 pA = (A*)&c; pC = dynamic_cast<C*>(pA);
 cout << pA << " casts to " << pC << ": Unrelated-cast: Invalid" << endl;

 pA = 0; pC = dynamic_cast<C*>(pA);
 cout << pA << " casts to " << pC << ": Unrelated-cast: Valid for null" << endl;

 pA = &a; pV = dynamic_cast<void*>(pA);
 cout << pA << " casts to " << pV << ": Cast-to-void: Valid" << endl;

 //pA = dynamic_cast<A*>(pV); // error: 'void *': invalid expression type for dynamic_cast

 return 0;
}
```

Output:

00EFFCA8 casts to 00EFFCA8: Up-cast: Valid  
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid  
00EFFCB4 casts to 00000000: Down-cast: Invalid  
00EFFC9C casts to 00000000: Unrelated-cast: Invalid  
00000000 casts to 00000000: Unrelated: Valid for null  
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid



# dynamic\_cast Operator: References

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
dynamic\_cast

typeid  
Operator

Summary

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

class A { public: virtual ~A() {} };
class B: public A {};
class C { public: virtual ~C() {} };

int main() {
 A a; B b; C c;
 try {
 B &rB1 = b;
 A &rA2 = dynamic_cast<A&>(rB1);
 cout << "Up-cast: Valid" << endl;

 A &rA3 = b;
 B &rB4 = dynamic_cast<B&>(rA3);
 cout << "Down-cast: Valid" << endl;

 try {
 A &rA5 = a;
 B &rB6 = dynamic_cast<B&>(rA5);
 } catch (bad_cast e) { cout << "Down-cast: Invalid: " << e.what() << endl; }

 try {
 A &rA7 = (A&)c;
 C &rC8 = dynamic_cast<C&>(rA7);
 } catch (bad_cast e) { cout << "Unrelated-cast: Invalid: " << e.what() << endl; }
 } catch (bad_cast e) { cout << "Bad-cast: " << e.what() << endl; }
 return 0;
}
```

Output:

Up-cast: Valid

Down-cast: Valid

Down-cast: Invalid: Bad dynamic\_cast!

Unrelated-cast: Invalid: Bad dynamic\_cast!



# typeid Operator

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`dynamic_cast`

`typeid`  
Operator

Summary

- `typeid` operator is used where the **dynamic type** of a **polymorphic object** must be known and for static type identification
- `typeid` operator can be applied on a type or an expression
- `typeid` operator returns `const std::type_info`. The major members are:
  - `operator==`, `operator!=`: checks whether the objects refer to the same type
  - `name`: implementation-defined name of the type
- `typeid` operator works for polymorphic type only (as it uses RTTI – virtual function table)
- If the polymorphic object is bad, the `typeid` throws `bad_typeid` exception



# Using typeid Operator: Polymorphic Hierarchy

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
dynamic\_cast

typeid  
Operator

Summary

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

// Polymorphic Hierarchy
class A { public: virtual ~A() {} };
class B : public A {};

int main() {
 A a;
 cout << typeid(a).name() << ":" << typeid(&a).name() << endl; // Static
 A *p = &a;
 cout << typeid(p).name() << ":" << typeid(*p).name() << endl; // Dynamic

 B b;
 cout << typeid(b).name() << ":" << typeid(&b).name() << endl; // Static
 p = &b;
 cout << typeid(p).name() << ":" << typeid(*p).name() << endl; // Dynamic

 A &r1 = a; A &r2 = b;
 cout << typeid(r1).name() << ":" << typeid(r2).name() << endl;

 return 0;
}

class A: class A *
class A *: class A
class B: class B *
class A *: class B
class A: class B
```



# Using typeid Operator: Staff Salary Application: Polymorphic Hierarchy

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`dynamic_cast`

typeid  
Operator

Summary

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

class Engineer { protected: string name_;
public: Engineer(const string& name) : name_(name) {}
 virtual void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer { Engineer *reports_[10];
public: Manager(const string& name) : Engineer(name) {}
 void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager { Manager *reports_[10];
public: Director(const string& name) : Manager(name) {}
 void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};

int main() {
 Engineer e("Rohit"); Manager m("Kamala"); Director d("Ranjana");
 Engineer *staff[] = { &e, &m, &d };
 for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
 cout << typeid(staff[i]).name() << ":" << typeid(*staff[i]).name() << endl;
 }

 return 0;
}

class Engineer *: class Engineer
class Engineer *: class Manager
class Engineer *: class Director
```



# Using typeid Operator: Non-Polymorphic Hierarchy

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
dynamic\_cast

typeid  
Operator

Summary

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

// Non-Polymorphic Hierarchy
class X {};
class Y : public X {};

int main() {
 X x;
 cout << typeid(x).name() << ":" << typeid(&x).name() << endl; // Static
 X *q = &x;
 cout << typeid(q).name() << ":" << typeid(*q).name() << endl; // Dynamic

 Y y;
 cout << typeid(y).name() << ":" << typeid(&y).name() << endl; // Static
 q = &y;
 cout << typeid(q).name() << ":" << typeid(*q).name() << endl; // Dynamic -- FAILS

 X &r1 = x; X &r2 = y;
 cout << typeid(r1).name() << ":" << typeid(r2).name() << endl;

 return 0;
}

class X: class X *
class X *: class X
class Y: class Y *
class X *: class X
class X: class X
```



# Using typeid Operator: bad\_typeid Exception

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
dynamic\_cast

typeid  
Operator

Summary

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

class A { public: virtual ~A() {} };
class B : public A {};

int main() {
 A *pA = new A;
 try {
 cout << typeid(pA).name() << endl;
 cout << typeid(*pA).name() << endl;
 } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

 delete pA;
 try {
 cout << typeid(pA).name() << endl;
 cout << typeid(*pA).name() << endl;
 } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

 pA = 0;
 try {
 cout << typeid(pA).name() << endl;
 cout << typeid(*pA).name() << endl;
 }
 catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

 return 0;
}
```

Output:

```
class A *
class A
class A *
caught Access violation - no RTTI data!
class A *
caught Attempted a typeid of NULL pointer!
```



# Module Summary

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators

dynamic\_cast

typeid  
Operator

Summary

- Understood casting at run-time
- Studied dynamic\_cast with examples
- Understood RTTI and typeid operator



# Instructor and TAs

Module 34

Partha Pratim  
Das

Objectives &  
Outline

Cast  
Operators  
`dynamic_cast`

`typeid`  
Operator

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, TA                    | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, TA                 | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA             | himadribhuyan@gmail.com   | 9438911655 |



Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
`protected`  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise  
Design Choice  
Summary

# Module 35: Programming in C++

## Multiple Inheritance

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
`protected`  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

- Understand Multiple Inheritance in C++



# Module Outline

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members

Overrides and  
Overloads

protected

Access

Constructor &  
Destructor

Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

- Multiple Inheritance in C++
  - Semantics
  - Data Members and Object Layout
  - Member Functions
  - protected Access
  - Constructor & Destructor
  - Object Lifetime
- Diamond Problem
  - Exercise
- Design Choice



# Multiple Inheritance in C++: Hierarchy

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

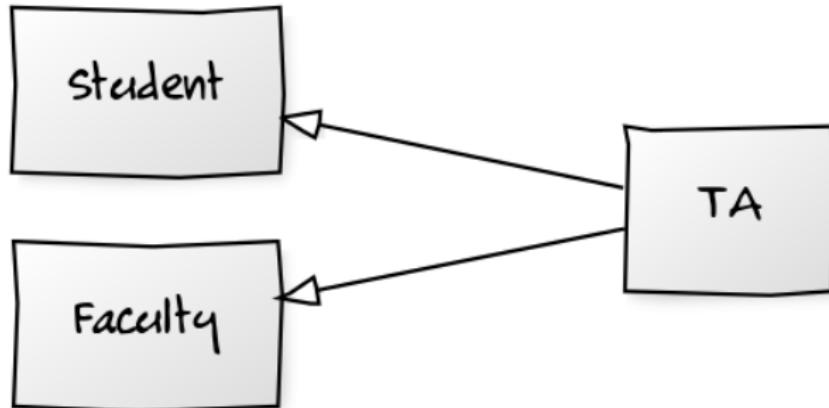
Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

- TA ISA Student; TA ISA Faculty



```
class Student; // Base Class = Student
class Faculty; // Base Class = Faculty
class TA: public Student, public Faculty; // Derived Class = TA
```

- TA inherits properties and operations of both Student as well as Faculty



# Multiple Inheritance in C++: Hierarchy

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads

protected  
Access

Constructor &  
Destructor

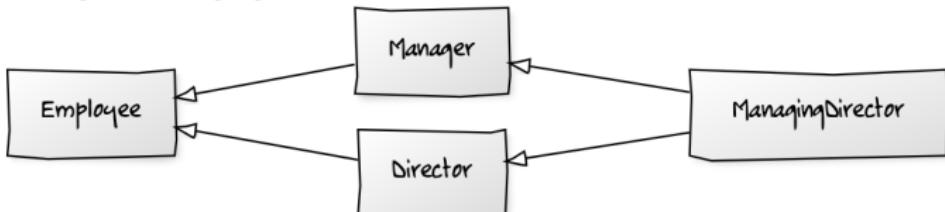
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

- Manager ISA Employee, Director ISA Employee, ManagingDirector ISA Manager, ManagingDirector ISA Director



```
class Employee; // Base Class = Employee -- Root
class Manager: public Employee; // Derived Class = Manager
class Director: public Employee; // Derived Class = Director
class ManagingDirector: public Manager, public Director;
// Derived Class = ManagingDirector
```

- Manager inherits properties and operations of Employee
- Director inherits properties and operations of Employee
- ManagingDirector inherits properties and operations of both Manager as well as Director
- ManagingDirector, by transitivity, inherits properties and operations of Employee
- Multiple inheritance hierarchy usually has a common base class
- This is known as the Diamond Hierarchy



# Multiple Inheritance in C++: Semantics

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++  
Semantics

Data Members  
Overrides and  
Overloads

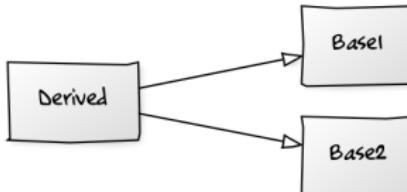
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

- Derived **ISA** Base1, Derived **ISA** Base2



```
class Base1; // Base Class = Base1
class Base2; // Base Class = Base2
class Derived: public Base1, public Base2;
 // Derived Class = Derived
```

- Use keyword **public** after class name to denote inheritance
- Name of the Base class follow the keyword
- There may be more than two base classes
- **public** and **private** inheritance may be mixed



# Multiple Inheritance in C++: Semantics

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++  
Semantics

Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice  
Summary

- Derived ISA Base1, Base2
- Data Members
  - Derived class *inherits all* data members of *all* Base classes
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits all* member functions of *all* Base classes
  - Derived class may *override* a member function of *any* Base class by *redefining* it with the *same signature*
  - Derived class may *overload* a member function of *any* Base class by *redefining* it with the *same name*; but *different signature*
- Access Specification
  - Derived class *cannot access private* members of *any* Base class
  - Derived class *can access protected* members of *any* Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call *all constructors* of the Base classes to construct the Base class instances of the Derived class – Base class *constructors* are called in *listing order*
  - The *destructor* of the Derived class *must* call the *destructors* of the Base classes to destroy the Base class instances of the Derived class



# Multiple Inheritance in C++: Data Members and Object Layout

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

- Derived **ISA** Base1, Base2
- Data Members
  - Derived class *inherits all* data members of *all* Base classes
  - Derived class may *add* data members of its own
- Object Layout
  - Derived class *layout* contains instances of *each* Base class
  - Further, Derived class *layout* will have data members of its own
  - C++ does not guarantee the *relative position* of the Base class instances and Derived class members



# Multiple Inheritance in C++: Data Members and Object Layout

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++  
Semantics  
Data Members

Overrides and  
Overloads

protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

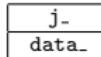
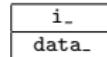
Design Choice

Summary

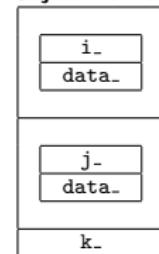
```
class Base1 { protected:
 int i_;
 int data_;
public: // ...
};
class Base2 { protected:
 int j_;
 int data_;
public: // ...
};
class Derived : public Base1, public Base2 {
 int k_;
public: // ...
};
```

## Object Layout

Object Base1      Object Base2



Object Derived



Object Derived has two data\_ member!

Ambiguity to be resolved with base class name: `Base1::data_ & Base2::data_`



# Multiple Inheritance in C++: Member Functions – Overrides and Overloads

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads

protected  
Access

Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice  
Summary

- Derived **ISA** Base1, Base2
- Member Functions
  - Derived class *inherits all* member functions of *all* Base classes
  - Derived class may *override* a member function of *any* Base class by *redefining* it with the *same signature*
  - Derived class may *overload* a member function of *any* Base class by *redefining* it with the *same name*; but *different signature*
- Static Member Functions
  - Derived class *does not inherit* the static member functions of *any* Base class
- Friend Functions
  - Derived class *does not inherit* the friend functions of *any* Base class



# Multiple Inheritance in C++: Member Functions – Overrides and Overloads

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads

protected  
Access  
Constructor &  
Destructor

Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

```
class Base1 { protected:
 int i_;
 int data_;
public: Base1(int a, int b) : i_(a), data_(b);
 void f(int) { cout << "Base1::f(int) "; }
 void g() { cout << "Base1::g() "; }
};
class Base2 { protected:
 int j_;
 int data_;
public: Base2(int a, int b) : j_(a), data_(b);
 void h(int) { cout << "Base2::h(int) "; }
};
class Derived : public Base1, public Base2 {
 int k_;
public: Derived(int x, int y, int u, int v, int z);
 void f(int) { cout << "Derived::f(int) "; } // -- Overridden Base1::f(int)
 // -- Inherited Base1::g()
 void h(string) { cout << "Derived::h(string) "; } // -- Overloaded Base2:: h(int)
 void e(char) { cout << "Derived::e(char) "; } // -- Added Derived::e(char)
};

Derived c(1, 2, 3, 4, 5);

c.f(5); // Derived::f(int) -- Overridden Base1::f(int)
c.g(); // Base1::g() -- Inherited Base1::g()
c.h("ppd"); // Derived::h(string) -- Overloaded Base2:: h(int)
c.e('a'); // Derived::e(char) -- Added Derived::e(char)
```



# Inheritance in C++:

## Member Functions – using for Name Resolution

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads

protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

| Ambiguous Calls                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Unambiguous Calls                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>class Base1 { public:     Base1(int a, int b) : i_(a), data_(b);     void f(int) { cout &lt;&lt; "Base1::f(int) " ; }     void g() { cout &lt;&lt; "Base1::g() " ; } };  class Base2 { public:     Base2(int a, int b) : j_(a), data_(b);     void f(int) { cout &lt;&lt; "Base2::f(int) " ; }     void g(int) { cout &lt;&lt; "Base2::g(int) " ; } };  class Derived : public Base1, public Base2 { public: Derived(int x, int y, int u,                int v, int z); };  Derived c(1, 2, 3, 4, 5);  c.f(5); // Base1::f(int) or Base2::f(int)? c.g(5); // Base1::g() or Base2::g(int)? c.f(3); // Base1::f(int) or Base2::f(int)? c.g(); // Base1::g() or Base2::g(int)?</pre> | <pre>class Base1 { public:     Base1(int a, int b) : i_(a), data_(b);     void f(int) { cout &lt;&lt; "Base1::f(int) " ; }     void g() { cout &lt;&lt; "Base1::g() " ; } };  class Base2 { public:     Base2(int a, int b) : j_(a), data_(b);     void f(int) { cout &lt;&lt; "Base2::f(int) " ; }     void g(int) { cout &lt;&lt; "Base2::g(int) " ; } };  class Derived : public Base1, public Base2 { public: Derived(int x, int y, int u,                int v, int z);     using Base1::f; // Hides Base2::f     using Base2::g; // Hides Base1::g };  Derived c(1, 2, 3, 4, 5);  c.f(5); // Base1::f(int) c.g(5); // Base2::g(int) c.Base2::f(3); // Base2::f(int) c.Base1::g(); // Base1::g()</pre> |

- Overload resolution does not work between `Base1::g(int)` and `Base2::g(int)`
- `using` hides other candidates
- Explicit use of base class name can resolve (weak solution)



# Multiple Inheritance in C++: Access Members of Base: protected Access

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads

**protected**  
Access

Constructor &  
Destructor

Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

## ● Access Specification

- Derived class *cannot access private* members of *any* Base class
- Derived class *can access protected* members of *any* Base class



# Multiple Inheritance in C++: Constructor & Destructor

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

## ● Constructor-Destructor

- Derived class *inherits all Constructors and Destructor of Base classes (but in a different semantics)*
- Derived class *cannot override or overload a Constructor or the Destructor of any Base class*

## ● Construction-Destruction

- A *constructor* of the Derived class *must first call all constructors* of the Base classes to construct the Base class instances of the Derived class
- Base class *constructors* are called in *listing order*
- The *destructor* of the Derived class *must call the destructors* of the Base classes to destruct the Base class instances of the Derived class



# Multiple Inheritance in C++: Constructor & Destructor

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access

Constructor &  
Destructor  
Object Lifetime

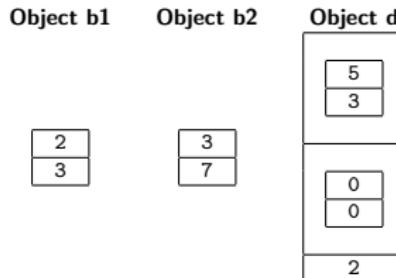
Diamond  
Problem  
Exercise

Design Choice

Summary

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b) : i_(a), data_(b) { cout << "Base1::Base1() "; }
 ~Base1() { cout << "Base1::~Base1() "; }
};
class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0) : j_(a), data_(b) { cout << "Base2::Base2() "; }
 ~Base2() { cout << "Base2::~Base2() "; }
};
class Derived : public Base1, public Base2 { int k_;
public: Derived(int x, int y, int z) :
 Base1(x, y), k_(z) { cout << "Derived::Derived() "; }
 // Base1::Base1 explicit, Base2::Base2 default
 ~Derived() { cout << "Derived::~Derived() "; }
};
Base1 b1(2, 3);
Base2 b2(3, 7);
Derived d(5, 3, 2);
```

## Object Layout





# Multiple Inheritance in C++: Object Lifetime

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads

protected  
Access

Constructor &  
Destructor

Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b) : i_(a), data_(b) { cout << "Base1::Base1() ";}
 ~Base1() { cout << "Base1::~Base1() ";}
};

class Base2 { protected: int j_; int data_;
public:
 Base2(int a = 0, int b = 0) : j_(a), data_(b) { cout << "Base2::Base2() ";}
 ~Base2() { cout << "Base2::~Base2() ";}
};

class Derived : public Base1, public Base2 { int k_;
public:
 Derived(int x, int y, int z) :
 Base1(x, y), k_(z) { cout << "Derived::Derived() ";}
 // Base1::Base1 explicit, Base2::Base2 default
 ~Derived() { cout << "Derived::~Derived() ";}
};

Derived d(5, 3, 2);
```

## Construction O/P

Base1::Base1(): 5, 3 // Obj. d.Base1  
Base2::Base2(): 0, 0 // Obj. d.Base2  
Derived::Derived(): 2 // Obj. d

## Destruction O/P

Derived::~Derived(): 2 // Obj. d  
Base2::~Base2(): 0, 0 // Obj. d.Base2  
Base1::~Base1(): 3, 5 // Obj. d.Base1

- First construct base class objects, then derived class object
- First destruct derived class object, then base class objects



# Multiple Inheritance in C++: Diamond Problem

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

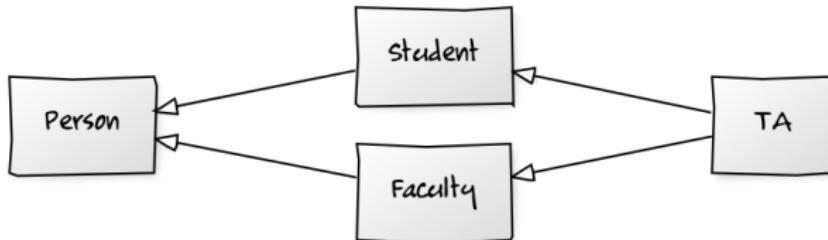
Diamond  
Problem

Exercise

Design Choice

Summary

- Student ISA Person
- Faculty ISA Person
- TA ISA Student; TA ISA Faculty



```
class Person; // Base Class = Person -- Root
class Student: public Person; // Base / Derived Class = Student
class Faculty: public Person; // Base / Derived Class = Faculty
class TA: public Student, public Faculty; // Derived Class = TA
```

- Student inherits properties and operations of Person
- Faculty inherits properties and operations of Person
- TA inherits properties and operations of both Student as well as Faculty
- TA, by transitivity, inherits properties and operations of Person



# Multiple Inheritance in C++: Diamond Problem

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise  
Design Choice  
Summary

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

class Person { // Data members of person
public: Person(int x) { cout << "Person::Person(int)" << endl; }
};

class Faculty : public Person { // data members of Faculty
public: Faculty(int x) :Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};

class Student : public Person { // data members of Student
public: Student(int x) :Person(x) { cout << "Student::Student(int)" << endl; }
};

class TA : public Faculty, public Student {
public: TA(int x) :Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};

int main() {
 TA ta(30);

 return 0;
}

Person::Person(int)
Faculty::Faculty(int)
Person::Person(int)
Student::Student(int)
TA::TA(int)
```

- Two instances of base class object (Person) in a TA object!



# Multiple Inheritance in C++: virtual Inheritance – virtual Base Class

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

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

class Person { // Data members of person
public: Person(int x) { cout << "Person::Person(int)" << endl; }
 Person() { cout << "Person::Person()" << endl; } // Default ctor for virtual inheritance
};

class Faculty : virtual public Person { // data members of Faculty
public: Faculty(int x) :Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};

class Student : virtual public Person { // data members of Student
public: Student(int x) :Person(x) { cout << "Student::Student(int)" << endl; }
};

class TA : public Faculty, public Student {
public: TA(int x) :Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};

int main() {
 TA ta(30);

 return 0;
}

Person::Person()
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)
```

- Introduce a default constructor for root base class Person
- Prefix every inheritance of Person with virtual
- Only one instance of base class object (Person) in a TA object!



# Multiple Inheritance in C++: virtual Inheritance with Parameterized Ctor

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

```
#include<iostream>
using namespace std;
class Person {
public: Person(int x) { cout << "Person::Person(int)" << endl; }
 Person() { cout << "Person::Person()" << endl; }
};
class Faculty : virtual public Person {
public: Faculty(int x) :Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student : virtual public Person {
public: Student(int x) :Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA : public Faculty, public Student {
public:
 TA(int x):Student(x), Faculty(x), Person(x) { cout << "TA::TA(int)" << endl; }
};
int main() {
 TA ta(30);

 return 0;
}

Person::Person(int)
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)
```

- Call parameterized constructor of root base class Person from constructor of TA class



# Multiple Inheritance in C++: Ambiguity

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

```
#include<iostream>
using namespace std;
class Person {
public: Person(int x) { cout << "Person::Person(int)" << endl; }
 Person() { cout << "Person::Person()" << endl; }
 virtual ~Person();
 virtual void teach() = 0;
};
class Faculty : virtual public Person {
public: Faculty(int x) :Person(x) { cout << "Faculty::Faculty(int)" << endl; }
 virtual void teach();
};
class Student : virtual public Person {
public: Student(int x) :Person(x) { cout << "Student::Student(int)" << endl; }
 virtual void teach();
};
class TA : public Faculty, public Student {
public:
 TA(int x):Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
 virtual void teach();
};
```

- In the absence of `TA::teach()`, which of `Student::teach()` or `Faculty::teach()` should be inherited?



# Multiple Inheritance in C++: Exercise

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members

Overrides and  
Overloads

protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

```
class A {
public:
 virtual ~A() { cout << "A::~A()" << endl; }
 virtual void foo() { cout << "A::foo()" << endl; }
};

class B : public virtual A {
public:
 virtual ~B() { cout << "B::~B()" << endl; }
 virtual void foo() { cout << "B::foo()" << endl; }
};

class C { // : public virtual A {
public:
 virtual ~C() { cout << "C::~C()" << endl; }
 virtual void foobar() { cout << "C::foobar()" << endl; }
};

class D : public B, public C {
public:
 virtual ~D() { cout << "D::~D()" << endl; }
 virtual void foo() { cout << "D::foo()" << endl; }
 virtual void foobar() { cout << "D::foobar()" << endl; }
};
```

- Consider the effect of calling **foo** and **foobar** for various objects and various pointers



# Design Choice: Inheritance or Composition

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

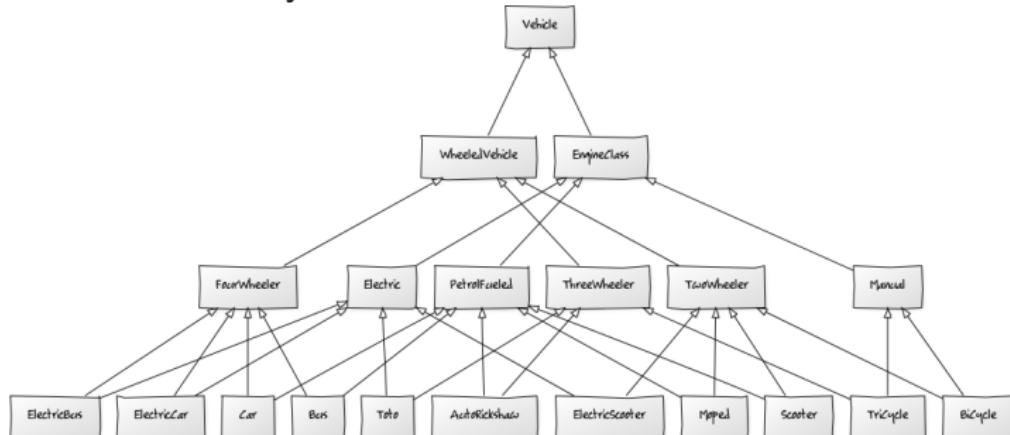
Semantics  
Data Members  
Overrides and  
Overloads  
protected  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

## • Vehicle Hierarchy



- Wheeled Hierarchy and Engine Hierarchy interact
- Large number of cross links!
- Multiplicative options make modeling difficult



# Design Choice: Inheritance or Composition

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

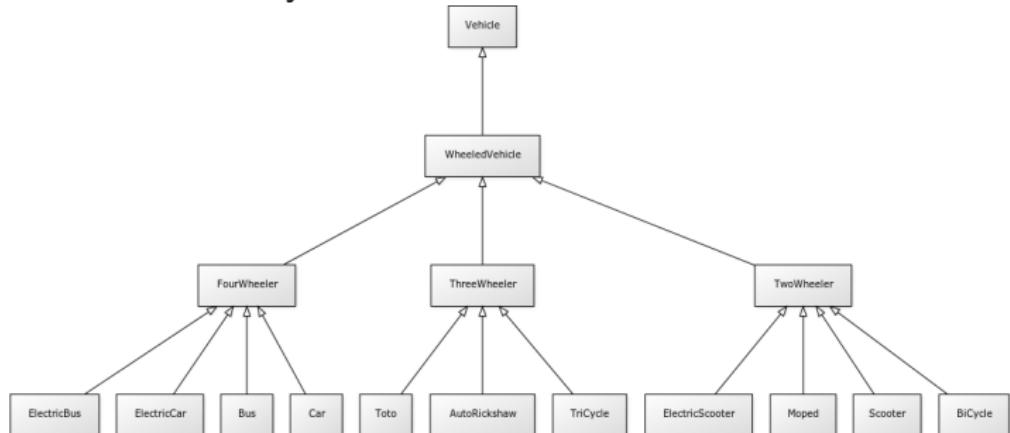
Semantics  
Data Members  
Overrides and  
Overloads  
`protected`  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

## • Vehicle Hierarchy



- Wheeled Hierarchy use Engine as Component
- Linear options to simplify models
- Is this dominant?



# Design Choice: Inheritance or Composition

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

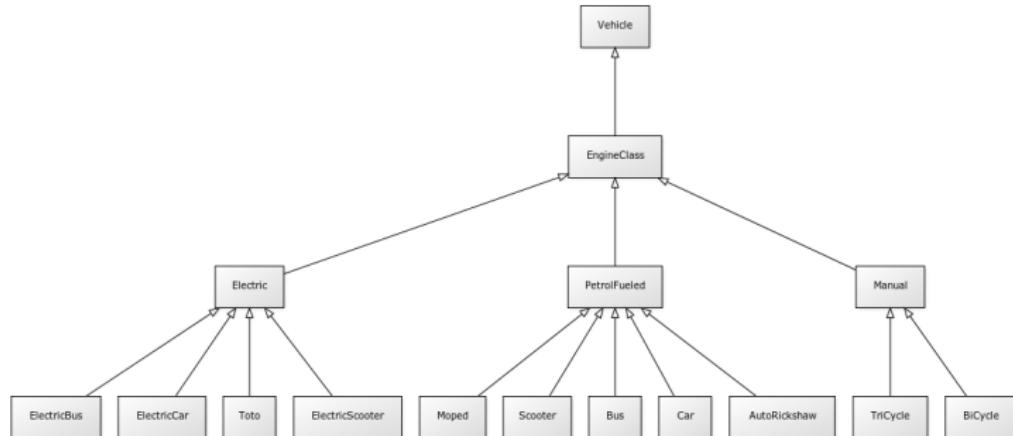
Semantics  
Data Members  
Overrides and  
Overloads  
`protected`  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

## • Vehicle Hierarchy



- Engine Hierarchy use Wheeled as Component
- Linear options to simplify models
- Is this dominant?



# Module Summary

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
`protected`  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem

Exercise

Design Choice

Summary

- Introduced the Semantics of Multiple Inheritance in C++
- Discussed the Diamond Problem and solution approaches
- Illustrated the design choice between inheritance and composition



# Instructor and TAs

Module 35

Partha Pratim  
Das

Objectives &  
Outline

Multiple  
Inheritance in  
C++

Semantics  
Data Members  
Overrides and  
Overloads  
`protected`  
Access  
Constructor &  
Destructor  
Object Lifetime

Diamond  
Problem  
Exercise

Design Choice

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

# Module 36: Programming C++

## Exceptions (Error handling in C): Part 1

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

- Understand the Error handling in C



# Module Outline

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params  
Local goto

C Standard  
Library Support

Global Variables  
Abnormal

Termination  
Conditional

Termination  
Non-Local goto

Signals  
Shortcomings

Summary

- Exception Fundamentals
  - Types of Exceptions
  - Exception Stages
- Exceptions in C
  - C Language Features
    - Return value & parameters
    - Local goto
  - C Standard Library Support
    - Global variables
    - Abnormal termination
    - Conditional termination
    - Non-local goto
    - Signal
  - Shortcomings
- Exceptions in C++
  - Exception Scope (try)
  - Exception Arguments (catch)
  - Exception Matching
  - Exception Raise (throw)
  - Advantages



# What are Exceptions?

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables

Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

- Conditions that arise
  - Infrequently and Unexpectedly
  - Generally betray a Program Error
  - Require a considered Programmatic Response
  - Run-time Anomalies – yes, but not necessarily

- Leading to
  - Crippling the Program
  - May pull the entire System down
  - Defensive Technique
    - Crashing Exceptions verses Tangled Design and Code



# Exception Causes

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

- Unexpected Systems State
  - Exhaustion of Resources
    - Low Free Store Memory
    - Low Disk Space
  - Pushing to a Full Stack
- External Events
  - C
  - Socket Event
- Logical Errors
  - Pop from an Empty Stack
  - Resource Errors – like Memory Read/Write
- Run time Errors
  - Arithmetic Overflow / Underflow
  - Out of Range
- Undefined Operation
  - Division by Zero



# Exception Handling?

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

- Exception Handling is a mechanism that separates the detection and handling of circumstantial **Exceptional Flow** from **Normal Flow**
- Current state saved in a pre-defined location
- Execution switched to a pre-defined handler

Exceptions are C++'s means of separating **error reporting** from **error handling**

– Bjarne Stroustrup



# Types of Exceptions

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions

Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto

Signals

Shortcomings

Summary

## ● **Asynchronous Exceptions:**

- Exceptions that come Unexpectedly
- Example - an Interrupt in a Program
- Takes control away from the Executing Thread context to a context that is different from that which caused the exception

## ● **Synchronous Exceptions:**

- Planned Exceptions
- Handled in an organized manner
- The most common type of Synchronous Exception is implemented as a `throw`



# Exception Stages

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

## ① Error Incidence

- Synchronous (S/W) Logical Error
- Asynchronous (H/W) Interrupt (S/W Interrupt)

## ② Create Object & Raise Exception

- An Exception Object can be of any Complete Type
- An int to a full blown C++ class object

## ③ Detect Exception

- Polling Software Tests
- Notification Control (Stack) Adjustments

## ④ Handle Exception

- Ignore: hope someone else handles it, that is, Do Not Catch
- Act: but allow others to handle it afterwards, that is, Catch, Handle and Re-Throw
- Own: take complete ownership, that is, Catch and Handle

## ⑤ Recover from Exception

- Continue Execution: If handled inside the program
- Abort Execution: If handled outside the program



# Exception Stages

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions

Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables

Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

```
int f() {
 int error;
 /* ... */
 if (error) /* Stage 1: error occurred */
 return -1; /* Stage 2: generate exception object */
 /* ... */
}

int main(void) {
 if (f() != 0) /* Stage 3: detect exception */
 {
 /* Stage 4: handle exception */
 }
 /* Stage 5: recover */
}
```



# Support for Exceptions in C

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto

Signals  
Shortcomings

Summary

- Language Features
  - Return Value & Parameters
  - Local goto
- Standard Library Support
  - Global Variables
  - Abnormal Termination
  - Conditional Termination
  - Non-Local goto
  - Signals



# Return Value & Parameters

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions

Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto

Signals

Shortcomings

Summary

## ● Function Return Value Mechanism

- **Created** by the Callee as Temporary Objects
- **Passed** onto the Caller
- **Caller checks** for Error Conditions
- **Return Values can be ignored and lost**
- **Return Values are temporary**

## ● Function (output) Parameter Mechanism

- Outbound Parameters, bound to arguments, offer multiple logical Return Values



# Example: Return Value & Parameters

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

```
int Push(int i) {
 if (top_ == size-1) // Incidence
 return 0; // Raise
 else
 stack_[++top_] = i;

 return 1;
}

int main() {
 int x;
 // ...
 if (!Push(x)) // Detect {
 // Handling
 }
 // Recovery
}
```



# Local goto

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

**Local goto**

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto  
Signals

Shortcomings

Summary

- Local goto Mechanism

- (At Source) Escapes: Gets Control out of a Deep Nested Loop
- (At Destination) Refactors: Actions from Multiple Points of Error Inception

- A group of C Features

- `goto Label;`
- `break;` & `continue;`
- `default` `switch case`



# Example: Local goto

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto

Signals

Shortcomings

Summary

```
_PHNDLR _cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
 ...
 /* Check for sigact support */
 if ((sigact == ...)) goto sigreterror;

 /* Not exceptions in the host OS. */
 if ((signum == ...)) { ... goto sigreterror; }
 else { ... goto sigretok; }

 /* Exceptions in the host OS. */
 if ((signum ...)) goto sigreterror;
 ...

sigretok:
 return(oldsigact);

sigreterror:
 errno = EINVAL;
 return(SIG_ERR);
}
```



# Example: Local goto

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params  
Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto  
Signals  
Shortcomings

Summary

```
_PHNDLR __cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
... /* Check for sigact support */
 if ((sigact == ...)) goto sigreterror;

 /* Not exceptions in the host OS. */
 if ((signum == ...) { ... goto sigreterror; }
else { ... goto sigretok; }

 /* Exceptions in the host OS. */
 if ((signum ...)) goto sigreterror;
...
sigretok:
 return(oldsigact);

sigreterror:
 errno = EINVAL;
 return(SIG_ERR);
}
```



# Example: Local goto

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params  
Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto  
Signals

Shortcomings

Summary

```
PHNDLR __cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
... /* Check for sigact support */
 if ((sigact == ...)) goto sigreterror;

 /* Not exceptions in the host OS. */
 if ((signum == ...)) { ... goto sigreterror; }
 else { ... goto sigretok; }

 /* Exceptions in the host OS. */
 if ((signum ...)) goto sigreterror;
...
sigretok:
 return(oldsigact);

sigrerror:
 errno = EINVAL;
 return(SIG_ERR);
}
```



# Example: Local goto

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params  
Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto  
Signals

Shortcomings

```
_PHNDLR __cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
... /* Check for sigact support */
 if ((sigact == ...)) goto sigreterror;

 /* Not exceptions in the host OS. */
 if ((signum == ...)) { ... goto sigreterror; }
 else { ... goto sigretok; }

 /* Exceptions in the host OS. */
 if ((signum ...)) goto sigreterror;
...
sigretok:
 return(oldsigact);

sigrerror:
 errno = EINVAL;
 return(SIG_ERR);
}
```

Summary



# Global Variables

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
**Global Variables**

Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

- GV Mechanism
  - Use a designated Global Error Variable
  - Set it on Error
  - Poll / Check it for Detection
- Standard Library GV Mechanism
  - `<errno.h>` / `<cerrno>`



# Example: Global Variables

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto

Signals

Shortcomings

Summary

```
#include <errno.h>
#include <math.h>
#include <stdio.h>

int main() {
 double x, y, result;
 /*... somehow set 'x' and 'y' ...*/
 errno = 0;

 result = pow (x, y);

 if (errno == EDOM)
 printf("Domain error on x/y pair \n");
 else if (errno == ERANGE)
 printf("range error in result \n");
 else
 printf("x to the y = %d \n", (int) result);

 return 0;
}
```



# Abnormal Termination

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables

Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

- Program Halting Functions provided by
  - `<stdlib.h>` / `<cstdlib>`
- `abort()`
  - Catastrophic Program Failure
- `exit()`
  - Code Clean up via `atexit()` Registrations
- `atexit()`
  - Handlers called in reverse order of their Registrations



# Example: Abnormal Termination

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions

Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto

Signals

Shortcomings

Summary

```
#include<stdio.h>
#include<stdlib.h>
static void atexit_handler_1(void) {
 printf("within 'atexit_handler_1' \n");
}

static void atexit_handler_2(void) {
 printf("within 'atexit_handler_2' \n");
}

int main(){
 atexit(atexit_handler_1);
 atexit(atexit_handler_2);
 exit(EXIT_SUCCESS);

 printf("This line should never appear \n");

 return 0;
}
/* On Execution Output: within 'atexit_handler_2'
 within 'atexit_handler_1'
 and returns a success code to calling environment */
```



# Conditional Termination

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

- Diagnostic ASSERT macro defined in
  - <assert.h> / <cassert>
- Assertions valid when NDEBUG macro is not defined (debug build is done)
- Assert calls internal function, reports the source file details and then Terminates



# Example: Conditional Termination

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

```
/* Debug version */
#ifndef NDEBUG
#include <cassert>
#include <cstdlib>
#include <cstdio>
using namespace std;

int main() {
 int i = 0;
 assert(++i == 0); // Assert 0 here

 printf(" i is %d \n", i);

 return 0;
}
/* When run - Asserts */

void _assert(int test, char const * test_image,
 char const * file, int line) {

 if (!test){
 printf("assertion failed: %s , file %s , line %d\n",
 test_image, file, line);
 abort();
 }
}
```



# Example: Conditional Termination

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination

Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

The screenshot shows a Microsoft Visual Studio interface. A debugger window is open, displaying the following message:

```
D:\My Courses\Software Engineering [CS20006]\2016.H1.Sp... - X
Assertion failed: ++i == 0, file d:\my courses\software engineering [cs20006]\2016.h1.spring\codes\moocs - programming in c++\module 36\assertion.cpp, line 10
```

The source code in the editor is:

```
3 return 0;
4 }
5 /* When run yields 'i' is 0 */
6
```

A 'Microsoft Visual C++ Runtime Library' dialog box is displayed, indicating a 'Debug Error!'. The details are:

**R6010**  
- abort() has been called

(Press Retry to debug the application)

Buttons at the bottom of the dialog are: Abort, Retry, Ignore.



# Example: Conditional Termination

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

```
/* Release version */
#define NDEBUG
#include <cassert>
#include <cstdlib>
#include <cstdio>
using namespace std;

int main(){
 int i = 0;
 assert(++i == 0); // Assert 0 here

 printf(" i is %d \n", i);

 return 0;
}
/* When run yields 'i' is 0 */

void _assert(int test, char const * test_image,
 char const * file, int line) {

 if (!test){
 printf("assertion failed: %s , file %s , line %d\n",
 test_image, file, line);
 abort();
 }
}
```



# Non-Local goto

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto

Signals

Shortcomings

Summary

- `setjmp()` and `longjmp()` functions provided in `<setjmp.h>` Header along with collateral type `jmp_buf`
- `setjmp(jmp_buf)`
  - Sets the Jump point filling up the `jmp_buf` object with the current program context
- `longjmp(jmp_buf, int)`
  - Effects a Jump to the context of the `jmp_buf` object
  - Control return to `setjmp` call last called on `jmp_buf`



# Example: Non-Local goto: The Dynamics

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables

Abnormal  
Termination  
Conditional  
Termination

Non-Local goto  
Signals  
Shortcomings

Summary

| Caller                                                                                                                                                                        | Callee                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <pre>void f() {     A a;     if (setjmp(jbuf) == 0)     {         B b;         g();         h();     }     else {         cout &lt;&lt; ex.what();     }     return ; }</pre> | <pre>jmp_buf jbuf ;  void g(){     A a;     UsrExcp ex ("From g()");     longjmp(jbuf, 1);     return ; }</pre> |



# Example: Non-Local goto

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables

Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

```
void f() {
 A a;
 if (setjmp(jbuf) == 0)
 {
 B b;
 g();
 h();
 }
 else {
 cout <<
 ex.what();
 }
 return;
}

jmp_buf jbuf;

void g()
{
 A a;
 UsrExcp ex("From g()");

 longjmp(jbuf, 1);

 return;
}
```

- g() called



# Example: Non-Local goto

## Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables

Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

```
void f() {
 A a;
 if (setjmp(jbuf) == 0)
 {
 B b;
 g();
 h();
 }
 else {
 cout << ex.what();
 }
 return;
}

jmp_buf jbuf;

void g()
{
 A a;
 UsrExcp ex("From g()");

 longjmp(jbuf, 1);
}

return;
```

- g() successfully returns



# Example: Non-Local goto

## Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto  
Signals  
Shortcomings

```
void f() {
 A a;
 if (setjmp(jbuf) == 0)
 {
 B b;
 g();
 h();
 }
 else {
 cout <<
 ex.what();
 }
 return;
}

jmp_buf jbuf;

void g()
{
 A a;
 UsrExcp ex("From g()");
 longjmp(jbuf, 1);
}

return;
```

- g() called and longjmp() executed
- setjmp() takes to exception part

Summary



# Example: Non-Local goto

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions

Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

Local goto

C Standard

Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto

Signals

Shortcomings

Summary

```
#include<setjmp.h>
#include<stdio.h>
jmp_buf j ;
void raise_exception(){
 printf("Exception raised \n");
 longjmp(j, 1) ; /* Jump to exception handler */
 printf("This line should never appear \n");
}
int main(){
 if (setjmp == 0) {
 printf(" 'setjmp' is initializing j \n ");
 raise_exception();
 printf("This line should never appear \n");
 }
 else
 printf(" 'setjmp' was just jumped into \n")
 /* This code is the exception handler */
 return 0 ;
}
/* On execution : 'setjmp' is initializing j ,
exception raised and 'setjmp' was just jumped into */
```



# Signals

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables

Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
**Signals**  
Shortcomings

Summary

- Header <signal.h>
- `raise()`
  - Sends a signal to the executing program
- `signal()`
  - Registers interrupt signal handler
  - Returns the previous handler associated with the given signal
- Converts h/w interrupts to s/w interrupts



# Example: Signals

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions

Exception Stages

Exceptions in  
C

C Language  
Features

RV & Params

Local goto

C Standard  
Library Support

Global Variables

Abnormal  
Termination

Conditional  
Termination

Non-Local goto

Signals

Shortcomings

Summary

```
// Use signal to attach a signal
// handler to the abort routine
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <tchar.h>
void SignalHandler(int signal) {
 printf("Application aborting...\n");
}

int main() {
 typedef void (*SignalHandlerPointer)(int);

 SignalHandlerPointer previousHandler;

 previousHandler = signal(SIGABRT, SignalHandler);

 abort();

 return 0;
}
```



# Shortcomings

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals

Shortcomings  
Summary

- **Destructor-ignorant:**
  - Cannot release Local Objects i.e. Resources Leak
- **Obtrusive:**
  - Interrogating RV or GV results in Code Clutter
- **Inflexible:**
  - Spoils Normal Function Semantics
- **Non-native:**
  - Require Library Support outside Core Language



# Module Summary

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

- Introduced the concept of exceptions
- Discussed exception (error) handling in C
- Illustrated various language features and library support in C for handling errors
- Demonstrated with examples



# Instructor and TAs

Module 36

Partha Pratim  
Das

Objective &  
Outline

Exception  
Fundamentals

Types of  
Exceptions  
Exception Stages

Exceptions in  
C

C Language  
Features  
RV & Params  
Local goto  
C Standard  
Library Support  
Global Variables  
Abnormal  
Termination  
Conditional  
Termination  
Non-Local goto  
Signals  
Shortcomings

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception  
Arguments  
(catch)  
Exception  
Matching  
Exception Raise  
(throw)  
Advantages

Summary

# Module 37: Programming C++

## Exceptions (Error handling in C++): Part 2

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception  
Arguments  
(catch)  
Exception  
Matching  
Exception Raise  
(throw)  
Advantages

Summary

- Understand the Error handling in C++



# Module Outline

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

- Exception Fundamentals
  - Types of Exceptions
  - Exception Stages
- Exceptions in C
  - C Language Features
    - Return value & parameters
    - Local goto
  - C Standard Library Support
    - Global variables
    - Abnormal termination
    - Conditional termination
    - Non-local goto
    - Signal
  - Shortcomings
- Exceptions in C++
  - Exception Scope (try)
  - Exception Arguments (catch)
  - Exception Matching
  - Exception Raise (throw)
  - Advantages



# Expectations

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception  
Arguments  
(catch)  
Exception  
Matching  
Exception Raise  
(throw)  
Advantages

Summary

- Separate Error-Handling code from Ordinary code
- Language Mechanism rather than of the Library
- Compiler for Tracking Automatic Variables
- Schemes for Destruction of Dynamic Memory
- Less Overhead for the Designer
- Exception Propagation from the deepest of levels
- Various Exceptions handled by a single Handler



# try-throw-catch

## Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception  
Arguments  
(catch)  
Exception  
Matching  
Exception Raise  
(throw)  
Advantages

Summary

```
void f() {
 A a;
 try {
 B b;
 g();
 h();
 }
 catch (UsrExcp& ex) {
 cout <<
 ex.what();
 }

 return;
}

class UsrExcp:
 public exceptions {}

void g()
{
 A a;
 UsrExcp ex("From g()");

 throw ex;

 return;
}
```

- g() called



# try-throw-catch

## Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception Arguments  
(catch)  
Exception Matching  
Exception Raise  
(throw)  
Advantages

Summary

```
void f() {
 A a;
 try {
 B b;
 g();
 h();
 }
 catch (UsrExcp& ex){
 cout <<
 ex.what();
 }

 return;
}

class UsrExcp:
 public exceptions {}

void g()
{
 A a;
 UsrExcp ex("From g()");

 throw ex;

 return;
}
```

- g() successfully returns



# try-throw-catch

## Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception  
Arguments  
(catch)  
Exception  
Matching  
Exception Raise  
(throw)  
Advantages

Summary

```
void f() {
 A a;
 try {
 B b;
 g();
 h();
 }
 catch (UsrExcp& ex) {
 cout <<
 ex.what();
 }

 return;
}

class UsrExcp:
 public exceptions {}

void g()
{
 A a;
 UsrExcp ex("From g()");
 throw ex;

 return;
}
```

- g() called and exception raised
- Exception caught by catch clause



# Exception Flow

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

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

class MyException : public exception {};
class MyClass {};

void h() { MyClass a;
 //throw 1;
 //throw 2.5;
 //throw MyException();
 //throw exception();
 //throw MyClass();
}

void g() { MyClass a;
 try {
 h();
 }
 catch (int) { cout << "int"; }
 catch (double) { cout << "double"; }
 catch (...) { throw; }
}

void f() { MyClass a;
 try {
 g();
 }
 catch (MyException) { cout << "MyException";
 catch (exception) { cout << "exception"; }
 catch (...) { throw; }
}

int main() {
 try {
 f();
 }
 catch (...) { cout << "Unknown"; }
 return 0;
}
```



# try Block: Exception Scope

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

- try block
  - Consolidate areas that might throw exceptions
- function try block
  - Area for detection is the entire function body
- Nested try block
  - Semantically equivalent to nested function calls

## Function try

```
void f()
 try {
 throw E();
 }
 catch (E& e) {
 }
```

## Nested try

```
try {
 try { throw E(); }
 catch (E& e) {
 }
}
```



# try-throw-catch

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception  
Arguments  
(catch)  
Exception  
Matching  
Exception Raise  
(throw)  
Advantages

Summary

```
void f() {
 A a;
 try {
 B b;
 g();
 h();
 }
 catch (UsrExcp& ex) {
 cout <<
 ex.what();
 }
 return;
}
```

```
class UsrExcp:
 public exceptions {}

void g()
{
 A a;
 UsrExcp ex("From g()");

 throw ex;

 return;
}
```

## ● try Block



# catch Block: Exception Arguments

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

## ● catch block

- Name for the Exception Handler
- Catching an Exception is like invoking a function
- Immediately follows the try block
- Unique Formal Parameter for each Handler
- Can be simply a Type Name to distinguish its Handler from others



# try-throw-catch

## Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

```
void f() {
 A a;
 try {
 B b;
 g();
 h();
 }
 catch (UsrExcp& ex) {
 cout <<
 ex.what();
 }
 return;
}
```

```
class UsrExcp:
 public exceptions {}

void g()
{
 A a;
 UsrExcp ex("From g()");

 throw ex;

 return;
}
```

## ● catch Block



# try-catch: Exception Matching

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

- Exact Match

- The catch argument type matches the type of the thrown object
  - No implicit conversion is allowed

- Generalization / Specialization

- The catch argument is a public base class of the thrown class object

- Pointer

- Pointer types – convertible by standard conversion



# try-throw-catch

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

```
void f() {
 A a;
 try {
 B b;
 g();
 h();
 } catch (UsrExcp& ex) {
 cout <<
 ex.what();
 }
 return;
}
```

```
class UsrExcp:
 public exceptions {}

void g()
{
 A a;
 UsrExcp ex("From g()");

 throw ex;

 return;
}
```

## ● Expression Matching



# try-catch: Exception Matching

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

- In the order of appearance with matching
- If Base Class catch block precedes Derived Class catch block
  - Compiler issues a warning and continues
  - Unreachable code (derived class handler) ignored
- `catch(...)` block must be the last catch block because it catches all exceptions
- If no matching Handler is found in the current scope, the search continues to find a matching handler in a dynamically surrounding try block
  - Stack Unwinds
- If eventually no handler is found, `terminate()` is called



# throw Expression: Exception Raise

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

- *Expression* is treated the same way as
  - A function argument in a call or the operand of a return statement
- Exception Context
  - `class Exception ;`
- The *Expression*
  - Generate an Exception object to throw
    - `throw Exception();`
  - Or, Copies an existing Exception object to throw
    - `Exception ex;`
    - `...`
    - `throw ex; // Exception(ex);`
- Exception object is created on the Free Store



# try-throw-catch

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception  
Arguments  
(catch)  
Exception  
Matching  
Exception Raise  
(throw)  
Advantages

Summary

```
void f() {
 A a;
 try {
 B b;
 g();
 h();
 }
 catch (UsrExcp& ex) {
 cout <<
 ex.what();
 }
 return;
}
```

```
class UsrExcp:
 public exceptions {}

void g()
{
 A a;
 UsrExcp ex("From g()");
 throw ex;
 return;
}
```

## ● throw Expression



# throw Expression: Restrictions

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

- For a UDT Expression
  - Copy Constructor and Destructor should be supported
- The type of Expression cannot be
  - An incomplete type (like void, array of unknown size or of elements of incomplete type, Declared but not Defined struct / union / enum / class Objects or Pointers to such Objects)
  - A pointer to an Incomplete type, except void\*, const void\*, volatile void\*, const volatile void\*



# (re)-throw: Throwing Again?

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

## • Re-throw

- catch may pass on the exception after handling
- Re-throw is not same as throwing again!

### Throws again

```
try { ... }
catch (Exception& ex) {
 // Handle and
 ...
 // Raise again
 throw ex;
 // ex copied
 // ex destructed
}
```

### Re-throw

```
try { ... }
catch (Exception& ex) {
 // Handle and
 ...
 // Pass-on
 throw;
 // No copy
 // No Destruction
}
```



# Advantages

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

- **Destructor-savvy:**

- Stack unwinds; Orderly destruction of Local-objects

- **Unobtrusive:**

- Exception Handling is implicit and automatic
- No clutter of error checks

- **Precise:**

- Exception Object Type designed using semantics

- **Native and Standard:**

- EH is part of the C++ language
- EH is available in all standard C++ compilers



# Advantages

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

## ● Scalable:

- Each function can have multiple try blocks
- Each try block can have a single Handler or a group of Handlers
- Each Handler can catch a single type, a group of types, or all types

## ● Fault-tolerant:

- Functions can specify the exception types to throw; Handlers can specify the exception types to catch
- Violation behavior of these specifications is predictable and user-configurable



# Module Summary

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)

Exception  
Arguments  
(catch)

Exception  
Matching

Exception Raise  
(throw)

Advantages

Summary

- Discussed exception (error) handling in C++
- Illustrated try-throw-catch feature in C++ for handling errors
- Demonstrated with examples



# Instructor and TAs

Module 37

Partha Pratim  
Das

Objective &  
Outline

Exceptions in  
C++

Exception Scope  
(try)  
Exception  
Arguments  
(catch)  
Exception  
Matching  
Exception Raise  
(throw)  
Advantages

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Definition

Instantiation

Template

Argument

Deduction

Example

`typename`

Summary

# Module 38: Programming in C++

## Template (Function Template): Part 1

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

`ppd@cse.iitkgp.ernet.in`

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

`typename`

Summary

- Understand Templates in C++



# Module Outline

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

- What is a Template?
- Function Template
  - Function Template Definition
  - Instantiation
  - Template Argument Deduction
  - Example
- typename
- Class Template
  - Class Template Definition
  - Instantiation
  - Partial Template Instantiation & Default Template Parameters
  - Inheritance



# What is a Template?

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

- Templates are specifications of a collection of functions or classes which are parameterized by types
- Examples:
  - Function search, min etc.
    - The basic algorithms in these functions are the same independent of types
    - Yet, we need to write different versions of these functions for strong type checking in C++
  - Classes list, queue etc.
    - The data members and the methods are almost the same for list of numbers, list of objects
    - Yet, we need to define different classes



# Function Template: Code reuse in Algorithms

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

- We need to compute the maximum of two values that can be of:

- int
- double
- char \* (C-String)
- Complex (user-defined class for complex numbers)
- ...

- We can do this with overloaded Max functions:

```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

With every new type, we need to add an overloaded function in the library!

- Issues in Max function

- Same algorithm (compare two value using the appropriate operator of the type and return the larger value)
- Different code versions of these functions for strong type checking in C++



# Max as Overload

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation

Template  
Argument  
Deduction  
Example

typename

Summary

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

// Overloads
int Max(int x, int y) { return x > y ? x : y; }
double Max(double x, double y) { return x > y ? x : y; }
char *Max(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() {
 int a = 3, b = 5, iMax;
 double c = 2.1, d = 3.7, dMax;

 cout << "Max(" << a << ", " << b << ") = " << Max(a, b) << endl;
 cout << "Max(" << c << ", " << d << ") = " << Max(c, d) << endl;

 char *s1 = new char[6], *s2 = new char[6];
 strcpy(s1, "black"); strcpy(s2, "white");
 cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl;
 strcpy(s1, "white"); strcpy(s2, "black");
 cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl;

 return 0;
}
```

- Overloaded solutions work
- In some cases (C-string), similar algorithms have exceptions
- With every new type, a new overloaded Max is needed
- Can we make Max generic and make a library to work with future types?
- How about macros?



# Max as a Macro

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

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

#define Max(x, y) (((x) > (y))? x: y)

int main() {
 int a = 3, b = 5;
 double c = 2.1, d = 3.7;

 cout << "Max(" << a << ", " << b << ") = " << Max(a, b) << endl;
 // Output: Max(3, 5) = 5

 cout << "Max(" << c << ", " << d << ") = " << Max(c, d) << endl;
 // Output: Max(2.1, 3.7) = 3.7

 return 0;
}
```

- Max, being a macro, is type oblivious – can be used for int as well as double, etc.
- Note the parentheses around parameters to protect precedence
- Note the parentheses around the whole expression to protect precedence
- Looks like a function – but does not behave as such



# Max as a Macro: Pitfalls

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

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

#define Max(x, y) (((x) > (y))? x: y)

int main() {
 int a = 3, b = 5;
 double c = 2.1, d = 3.7;

 // Side Effects
 cout << "Max(" << a << ", " << b << ") = " ; // Output: Max(3, 5) = 6
 cout << Max(a++, b++) << endl;
 cout << "a = " << a << ", b = " << b << endl; // Output: a = 4, b = 7

 // C-String Comparison
 char *s1 = new char[6], *s2 = new char[6];
 strcpy(s1, "black"); strcpy(s2, "white");
 cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl;
 // Output: Max(black, white) = white

 strcpy(s1, "white"); strcpy(s2, "black");
 cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl;
 // Output: Max(white, black) = black

 return 0;
}
```

- In "Side Effects" – the result is wrong, the larger values gets incremented twice
- In "C-String Comparison" – swapping parameters changes the result – actually compares pointers



# Function Template

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename  
Summary

## ● A function template

- describes how a function should be built
- supplies the definition of the function using some arbitrary types, (as place holders)
  - a **parameterized** definition
- can be considered the definition for a **set of overloaded versions** of a function
- is identified by the **keyword template**
  - followed by comma-separated list of **parameter** identifiers (each preceded by **keyword class** or **keyword typename**)
  - enclosed between **<** and **>** delimiters
  - followed by the signature the function
- Note that every template parameter is a **built-in type** or **class – type** parameters



# Max as a Function Template

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation

Template  
Argument  
Deduction  
Example

typename

Summary

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

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

int main() {
 int a = 3, b = 5, iMax;
 double c = 2.1, d = 3.7, dMax;

 iMax = Max<int>(a, b);
 cout << "Max(" << a << ", " << b << ") = " << iMax << endl;
 // Output: Max(3, 5) = 5

 dMax = Max<double>(c, d);
 cout << "Max(" << c << ", " << d << ") = " << dMax << endl;
 // Output: Max(2.1, 3.7) = 3.7

 return 0;
}
```

- Max, now, knows the type
- Template type parameter T explicitly specified in instantiation of Max<int>, Max<double>



# Max as a Function Template: Pitfall "Side Effects" – Solved

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

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

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

int main() {
 int a = 3, b = 5, iMax;

 // Side Effects
 cout << "Max(" << a << ", " << b << ") = ";
 iMax = Max<int>(a++, b++);
 cout << iMax << endl;
 // Output: Max(3, 5) = 5

 cout << "a = " << a << ", b = " << b << endl;
 // Output: a = 4, b = 6

 return 0;
}
```

- Max is now a proper function call – no side effect



# Max as a Function Template: Pitfall "C-String Comparison" – Solved

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

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

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

template<> // Template specialization for 'char *' type
char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() {
 char *s1 = new char[6], *s2 = new char[6];

 strcpy(s1, "black"); strcpy(s2, "white");
 cout << "Max(" << s1 << ", " << s2 << ") = " << Max<char>(s1, s2) << endl;
 // Output: Max(black, white) = white

 strcpy(s1, "white"); strcpy(s2, "black");
 cout << "Max(" << s1 << ", " << s2 << ") = " << Max<char>(s1, s2) << endl;
 // Output: Max(black, white) = white

 return 0;
}
```

- Generic template code does not work for C-Strings as it compares pointers, not the strings pointed by them
- We provide a specialization to compare pointers using comparison of strings
- Need to specify type explicitly is bothersome



# Max as a Function Template: Implicit Instantiation

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation

Template  
Argument  
Deduction  
Example

typename

Summary

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

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

int main() {
 int a = 3, b = 5, iMax;
 double c = 2.1, d = 3.7, dMax;

 iMax = Max(a, b); // Type 'int' inferred from 'a' and 'b' parameters types
 cout << "Max(" << a << ", " << b << ") = " << iMax << endl;
 // Output: Max(3, 5) = 5

 dMax = Max(c, d); // Type 'double' inferred from 'c' and 'd' parameters types
 cout << "Max(" << c << ", " << d << ") = " << dMax << endl;
 // Output: Max(2.1, 3.7) = 3.7

 return 0;
}
```

- Often template type parameter T may be inferred from the type of parameters in the instance
- If the compiler cannot infer or infers wrongly, we use explicit instantiation



# Template Argument Deduction: Implicit Instantiation

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation

Template  
Argument  
Deduction  
Example

typename

Summary

- Each item in the template parameter list is a template argument
- When a template function is invoked, the values of the template arguments are determined by seeing the types of the function arguments

```
template<class T> T Max(T x, T y);
template<> char *Max<char *>(char *x, char *y);
template <class T, int size> Type Max(T x[size]);

int a, b; Max(a, b); // Binds to Max<int>(int, int);
double c, d; Max(c, d); // Binds to Max<double>(double, double);
char *s1, *s2; Max(s1, s2); // Binds to Max<char*>(char*, char*);

int pval[9]; Max(pval); //Error!
```

- Three kinds of conversions are allowed
  - L-value transformation (for example, Array-to-pointer conversion)
  - Qualification conversion
  - Conversion to a base class instantiation from a class template
- If the same template parameter are found for more than one function argument, template argument deduction from each function argument must be the same



# Max as a Function Template: With User-Defined Class

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

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

class Complex { double re_; double im_;
public:
 Complex(double re = 0.0, double im = 0.0) : re_(re), im_(im) {};
 double norm() const { return sqrt(re_*re_+im_*im_); }
 friend bool operator>(const Complex& c1, const Complex& c2) {
 return c1.norm() > c2.norm();
 }
 friend ostream& operator<<(ostream& os, const Complex& c) {
 os << "(" << c.re_ << ", " << c.im_ << ")"; return os;
 }
};
template<class T> T Max(T x, T y) { return x > y ? x : y; }
template<> char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() { Complex c1(2.1, 3.2), c2(6.2, 7.2);

 cout << "Max(" << c1 << ", " << c2 << ") = " << Max(c1, c2) << endl;
 // Output: Max((2.1, 3.2), (6.2, 7.2)) = (6.2, 7.2)

 return 0;
}
```

- When Max is instantiated with class Complex, we need comparison operator for Complex
- The code, therefore, will not compile without bool operator>(const Complex&, const Complex&)
- Traits of type variable T include bool operator>(T, T) which the instantiating type must fulfill



# Max as a Function Template: Overloads

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

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

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

template<> char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

template<class T, int size> T Max(T x[size]) { T t = x[0];
 for (int i = 0; i < size; ++i) { if (x[i] > t) t = x[i]; }

 return t;
}

int main() {
 int arr[] = { 2, 5, 6, 3, 7, 9, 4 };
 cout << "Max(arr) = " << Max<int, 7>(arr) << endl; // Output: Max(arr) = 9

 return 0;
}
```

- Template function can be overloaded
- A template parameter can be non-type (int) constant



# Swap as a Function Template

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

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

template<class T> void Swap(T& one, T& other)
{
 T temp;
 temp = one; one = other; other = temp;
}

int main() {
 int i = 10, j = 20;
 cout << "i = " << i << ", j = " << j << endl;
 Swap(i, j);
 cout << "i = " << i << ", j = " << j << endl;

 string s1("abc"), s2("def");

 cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
 Swap(s1, s2);
 cout << "s1 = " << s1 << ", s2 = " << s2 << endl;

 return 0;
}
```

- The traits of type variable T include
  - default constructor (`T::T()`) and
  - copy assignment operator (`T operator=(const T&)`)
- Our template function cannot be called swap, as std namespace has such a function



# typename Keyword

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename  
Summary

- Consider:

```
template <class T> f (T x) {
 T::name * p;
}
```

- What does it mean?

- T::name is a type and p is a pointer to that type
- T::name and p are variables and this is a multiplication

- To resolve, we use keyword **typename**:

```
template <class T> f (T x) { T::name * p; } // Multiplication

template <class T> f (T x) { typename T::name * p; } // Type
```

- The keywords **class** and **typename** have almost the same meaning in a template parameter
- typename** is also used to tell the compiler that an expression is a type expression



# Module Summary

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

typename

Summary

- Introduced the templates in C++
- Discussed function templates as generic algorithmic solution for code reuse
- Explained templates argument deduction for implicit instantiation
- Illustrated with examples



# Instructor and TAs

Module 38

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template  
Definition  
Instantiation  
Template  
Argument  
Deduction  
Example

`typename`

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

# Module 39: Programming in C++

## Template (Class Template): Part 2

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

- Understand Templates in C++



# Module Outline

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

- What is a Template?
- Function Template
  - Function Template Definition
  - Instantiation
  - Template Argument Deduction
  - Example
- typename
- Class Template
  - Class Template Definition
  - Instantiation
  - Partial Template Instantiation & Default Template Parameters
  - Inheritance



# What is a Template?: RECAP (Module 38)

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

- Templates are specifications of a collection of functions or classes which are parameterized by types
- Examples:
  - Function search, min etc.
    - The basic algorithms in these functions are the same independent of types
    - Yet, we need to write different versions of these functions for strong type checking in C++
  - Classes list, queue etc.
    - The data members and the methods are almost the same for list of numbers, list of objects
    - Yet, we need to define different classes



# Function Template: Code reuse in Algorithms: RECAP (Module 38)

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template

Definition

Instantiation

Partial Template

Instantiation &

Default

Template

Parameters

Inheritance

Summary

- We need to compute the maximum of two values that can be of:

- int
- double
- char \* (C-String)
- Complex (user-defined class for complex numbers)
- ...

- We can do this with overloaded Max functions:

```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

With every new type, we need to add an overloaded function in the library!

- Issues in Max function

- Same algorithm (compare two value using the appropriate operator of the type and return the larger value)
- Different code versions of these functions for strong type checking in C++



# Class Template: Code Reuse in Data Structure

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template

Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

- Solution of several problems needs stack (LIFO)
  - Reverse string (char)
  - Convert infix expression to postfix (char)
  - Evaluate postfix expression (int / double / Complex ...)
  - Depth-first traversal (Node \*)
  - ...
- Solution of several problems needs queue (FIFO)
  - Task Scheduling (Task \*)
  - Process Scheduling (Process \*)
  - ...
- Solution of several problems needs list (ordered)
  - Implementing stack, queue (int / char / ...)
  - Implementing object collections (UDT)
  - ...
- Solution of several problems needs ...
- **Issues in Data Structure**
  - **Data Structures are generic - same interface, same algorithms**
  - **C++ implementations are different due to element type**



# Stack of char and int

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template

Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

```
class Stack { class Stack {
 char data_[100]; // Has type int data_[100]; // Has type
 int top_;
public:
 Stack() :top_(-1) {} Stack() :top_(-1) {}
 ~Stack() {}

 void push(const char& item) // Has type void push(const int& item) // Has type
 { data_[++top_] = item; }

 void pop()
 { --top_; }

 const char& top() const // Has type const int& top() const // Has type
 { return data_[top_]; }

 bool empty() const // Has type bool empty() const // Has type
 { return top_ == -1; }
};
```

- Stack of char
- Stack of int
- Can we combine these Stack codes using a type variable T?



# Class Template

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

## ● A class template

- describes how a class should be built
- Supplies the class description and the definition of the member functions using some arbitrary type name, (as a place holder)
- is a:
  - **parameterized** type with
  - **parameterized** member functions
- can be considered the definition for a **unbounded set** of class types
- is identified by the **keyword template**
  - followed by comma-separated list of **parameter** identifiers (each preceded by **keyword class** or **keyword typename**)
  - enclosed between **<** and **>** delimiters
  - followed by the definition of the class
- is often used for **container** classes
- Note that every template parameter is a **built-in type** or **class – type parameters**



# Stack as a Class Template:

## Stack.h

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template

Definition

Instantiation

Partial Template

Instantiation &

Default

Template

Parameters

Inheritance

Summary

```
template<class T>
class Stack {
 T data_[100];
 int top_;
public:
 Stack() :top_(-1) {}
 ~Stack() {}

 void push(const T& item)
 { data_[++top_] = item; }

 void pop()
 { --top_; }

 const T& top() const
 { return data_[top_]; }

 bool empty() const
 { return top_ == -1; }
};

- Stack of type variable T
- The traits of type variable T include
 - copy assignment operator (T operator=(const T&))
- We do not call our template class as stack because std namespace has a class stack

```



# Reverse String: Using Stack template

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition

Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() {
 char str[10] = "ABCDE";

 Stack<char> s; // Instantiated for char

 for (unsigned int i = 0; i < strlen(str); ++i)
 s.push(str[i]);

 cout << "Reversed String: ";
 while (!s.empty()) {
 cout << s.top();
 s.pop();
 }

 return 0;
}

● Stack of type char
```



# Postfix Expression Evaluation: Using Stack template

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition

Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() {
 // Postfix expression: 1 2 3 * + 9 -
 unsigned int postfix[] = { '1', '2', '3', '*', '+', '9', '-' }, ch;

 Stack<int> s; // Instantiated for int

 for (unsigned int i = 0; i < sizeof(postfix) / sizeof(unsigned int); ++i) {
 ch = postfix[i];
 if (isdigit(ch)) { s.push(ch - '0'); }
 else {
 int op1 = s.top(); s.pop();
 int op2 = s.top(); s.pop();
 switch (ch) {
 case '*': s.push(op2 * op1); break;
 case '/': s.push(op2 / op1); break;
 case '+': s.push(op2 + op1); break;
 case '-': s.push(op2 - op1); break;
 }
 }
 }
 cout << "\nEvaluation " << s.top();

 return 0;
}

• Stack of type int
```



# Template Parameter Traits

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

## ● Parameter Types

- may be of any type (including user defined types)
- may be parameterized types, (that is, templates)
- MUST support the methods used by the template functions:
  - What are the required constructors?
  - The required operator functions?
  - What are the necessary defining operations?



# Function Template Instantiation: RECAP (Module 38)

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition

Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

- Each item in the template parameter list is a template argument
- When a template function is invoked, the values of the template arguments are determined by seeing the types of the function arguments

```
template<class T> T Max(T x, T y);
template<> char *Max<char *>(char *x, char *y);
template <class T, int size> Type Max(T x[size]);

int a, b; Max(a, b); // Binds to Max<int>(int, int);
double c, d; Max(c, d); // Binds to Max<double>(double, double);
char *s1, *s2; Max(s1, s2); // Binds to Max<char*>(char*, char*);

int pval[9]; Max(pval); //Error!
```

- Three kinds of conversions are allowed
  - L-value transformation (for example, Array-to-pointer conversion)
  - Qualification conversion
  - Conversion to a base class instantiation from a class template
- If the same template parameter are found for more than one function argument, template argument deduction from each function argument must be the same



# Class Template Instantiation

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation

Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

- Class Template is instantiated only when it is required:
  - `template<class T> class Stack;` is a forward declaration
  - `Stack<char> s;` is an error
  - `Stack<char> *ps;` is okay
  - `void ReverseString(Stack<char>& s, char *str);` is okay
- Class template is instantiated before
  - An object is defined with class template instantiation
  - If a pointer or a reference is dereferenced (for example, a method is invoked)
- A template definition can refer to a class template or its instances but a non-template can only refer to template instances



# Class Template Instantiation Example

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation

Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

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

template<class T> class Stack; // Forward declaration

void ReverseString(Stack<char>& s, char *str); // Stack template definition is not needed

template<class T> // Definition
class Stack { T data_[100]; int top_;
public: Stack() :top_(-1) {} ~Stack() {}

 void push(const T& item) { data_[++top_] = item; }
 void pop() { --top_; }
 const T& top() const { return data_[top_]; }
 bool empty() const { return top_ == -1; }
};

int main() {
 char str[10] = "ABCDE";
 Stack<char> s; // Stack template definition is needed

 ReverseString(s, str);

 return 0;
}
void ReverseString(Stack<char>& s, char *str) { // Stack template definition is needed
 for (unsigned int i = 0; i < strlen(str); ++i) s.push(str[i]);

 cout << "Reversed String: ";
 while (!s.empty()) { cout << s.top(); s.pop(); }
}
```



# Partial Template Instantiation and Default Template Parameters

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

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

template<class T1 = int, class T2 = string> // Version 1 with default parameters
class Student { T1 roll_; T2 name_;
public: Student(T1 r, T2 n) : roll_(r), name_(n) {}
 void Print() const { cout << "Version 1: (" << name_ << ", " << roll_ << ")" << endl; }
};

template<class T1> // Version 2: Partial Template Specialization
class Student<T1, char*> { T1 roll_; char *name_;
public: Student(T1 r, char *n) : roll_(r), name_(strncpy(new char[strlen(n) + 1], n)) {}
 void Print() const { cout << "Version 2: (" << name_ << ", " << roll_ << ")" << endl; }
};

int main() {
 Student<int, string> s1(2, "Ramesh"); // Version 1: T1 = int, T2 = string
 Student<int> s2(11, "Shampa"); // Version 1: T1 = int, defa T2 = string
 Student<> s3(7, "Gagan"); // Version 1: defa T1 = int, defa T2 = string
 Student<string> s4("X9", "Lalita"); // Version 1: T1 = string, defa T2 = string
 Student<int, char*> s5(3, "Gouri"); // Version 2: T1 = int, T2 = char*

 s1.Print(); s2.Print(); s3.Print(); s4.Print(); s5.Print();

 return 0;
}

Version 1: (Ramesh, 2)
Version 1: (Shampa, 11)
Version 1: (Gagan, 7)
Version 1: (Lalita, X9)
Version 2: (Gouri, 3)
```



# Templates and Inheritance: Example (List.h)

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template

Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

```
#ifndef __LIST_H
#define __LIST_H

#include <vector>
using namespace std;

template<class T>
class List {
public:
 void put(const T &val) { items.push_back(val); }
 int length() { return items.size(); }
 bool find(const T &val) {
 for (unsigned int i = 0; i < items.size(); ++i)
 if (items[i] == val) return true;
 return false;
 }
private:
 vector<T> items;
};

#endif // __LIST_H
```

- List is basic container class



# Templates and Inheritance: Example (Set.h)

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

```
#ifndef __SET_H
#define __SET_H

#include "List.h"

template<class T>
class Set {
public:
 Set() { };
 virtual void add(const T &val);
 int length();
 bool find(const T &val);
private:
 List<T> items;
};

template<class T>
void Set<T> :: add(const T &val)
{
 if (items.find(val)) return;
 items.put(val);
}
template<class T> int Set<T> :: length() { return items.length(); }
template<class T> bool Set<T> :: find(const T &val) { return items.find(val); }
#endif // __SET_H
```

- Set is a base class for a set
- Set uses List for container



# Templates and Inheritance: Example (BoundSet .h)

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

```
#ifndef __BOUND_SET_H
#define __BOUND_SET_H

#include "Set.h"

template<class T>
class BoundSet : public Set<T> {
public:
 BoundSet(const T &lower, const T &upper);
 void add(const T &val);
private:
 T min;
 T max;
};

template<class T> BoundSet<T>::BoundSet(const T &lower, const T &upper)
 : min(lower), max(upper) { }

template<class T> void BoundSet<T>::add(const T &val) {
 if (find(val)) return;
 if ((val <= max) && (val >= min))
 Set<T>::add(val);
}

#endif // __BOUND_SET_H
```

- BoundSet is a specialization of Set
- BoundSet is a set of bounded items



# Templates and Inheritance: Example (Bounded Set Application)

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template

Definition

Instantiation

Partial Template

Instantiation &

Default

Template

Parameters

Inheritance

Summary

```
#include <iostream>
using namespace std;
#include "BoundSet.h"

int main() {
 int i;
 BoundSet<int> bsi(3, 21);
 Set<int> *setptr = &bsi;

 for (i = 0; i < 25; i++) setptr->add(i);

 if (bsi.find(4))
 cout << "We found an expected value\n";

 if (bsi.find(0) || bsi.find(25)) {
 cout << "We found an Unexpected value\n";
 return -1;
 }
 else
 cout << "We found NO unexpected value\n";

 return 0;
}

We found an expected value
We found NO unexpected value
```

- Uses BoundSet to maintain and search elements



# Module Summary

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

- Introduced the templates in C++
- Discussed class templates as generic solution for data structure reuse
- Explained partial template instantiation and default template parameters
- Demonstrated templates on inheritance hierarchy
- Illustrated with examples



# Instructor and TAs

Module 39

Partha Pratim  
Das

Objectives &  
Outline

What is a  
Template?

Function  
Template

Class  
Template  
Definition  
Instantiation  
Partial Template  
Instantiation &  
Default  
Template  
Parameters  
Inheritance

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |



Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward

Summary

## Module 40: Programming C++

Closing Comments

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward

Summary

- Review C++ Course
- Information for Examination
- What next?



# Module Outline

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward

Summary

- Course Summary
- Key Take-back
- Prepare for Examination
- Road Forward



# What we covered

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward  
Summary

- **Programming in C++ is Fun:** Build and execute a C programs in C++, Write equivalent programs in C++
- **C++ as Better C:** Procedural Extensions of C
- **Object-Oriented Programming in C++:** Classes, Encapsulation, Overloading, friend, static, and namespace
- **Inheritance:** Generalization / Specialization of Object Modeling in C++
- **Polymorphism:** Static and Dynamic Binding, Virtual Function Table, Multiple Inheritance
- **Type Casting:** Cast Operators
- **Exceptions:** Error Handling in C & C++
- **Templates:** Generic Programming in C++



# What we did not cover?

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward  
Summary

- **Functors:** Function Objects
- **STL:** Standard Template Library of C++
- **Resource Management:** Smart Pointers, Memory Handling
- **C++ Coding Styles:** How to write good code?
- **Design Patterns:** Reusable Designs
- **Mixing C and C++:** Smart Pointers, Memory Handling
- **Source Code Management:** How to organize files, libraries?
- **C++ Tools:** Analysis, Version Control etc.
- ...



# What have we learnt?

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward  
Summary

- C++ is multi-paradigm
  - Procedural: Better C
  - Object-Oriented: Encapsulation, Inheritance, and Polymorphism
  - Generic: Templates
- Reuse is Key
  - Macros
  - Library functions
  - Function Overloading (Static Polymorphism)
  - Inheritance & Dynamic Polymorphism
  - Templates & STL
  - Design Patterns
- Designing good data types is a key for good programming in C++
- While programming in C++, we should keep an eye on:
  - Efficiency
  - Safety
  - Clarity

**Do not write C-style programs using C++ compiler**



# Prepare for Examination

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward

Summary

- Watch the Videos
- Revise the Assignments and Solutions
- Practice lots and lots of coding with every feature
- Design and implement complete data types – Complex, Fraction, Vector, Matrix, Polynomial etc.
- Study Books, try examples
  - The C++ Programming Language by Bjarne Stroustrup
  - Effective C++ & More Effective C++ by Scott Meyers



# Road Forward

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward  
Summary

- Learn the topics not covered
- Breathe programming – regularly code and implement systems
- Read lots and lots of programs by good coders
- Learn Python / Java
- Study **Object Oriented Analysis and Design**
- Study **Unified Modeling Language**
- Study **Software Engineering**
- Study Books
  - The C++ Programming Language by Bjarne Stroustrup
  - Effective C++ & More Effective C++ by Scott Meyers
  - Exceptional C++ & More Exceptional C++ by Herb Sutter
  - Modern C++ Design by Andrei Alexandrescu
  - Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides
  - Learning UML 2.0 – A Pragmatic Introduction to UML by Russ Miles & Kim Hamilton (O'Reilly)



# Module Summary

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward

Summary

- Course on C++ concluded



# Instructor and TAs

Module 40

Partha Pratim  
Das

Objectives &  
Outline

Course  
Summary

Key Take-back

Prepare for  
Examination

Road Forward

Summary

| Name                                 | Mail                      | Mobile     |
|--------------------------------------|---------------------------|------------|
| Partha Pratim Das, <i>Instructor</i> | ppd@cse.iitkgp.ernet.in   | 9830030880 |
| Tanwi Mallick, <i>TA</i>             | tanwimallick@gmail.com    | 9674277774 |
| Srijoni Majumdar, <i>TA</i>          | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, <i>TA</i>      | himadribhuyan@gmail.com   | 9438911655 |