C++

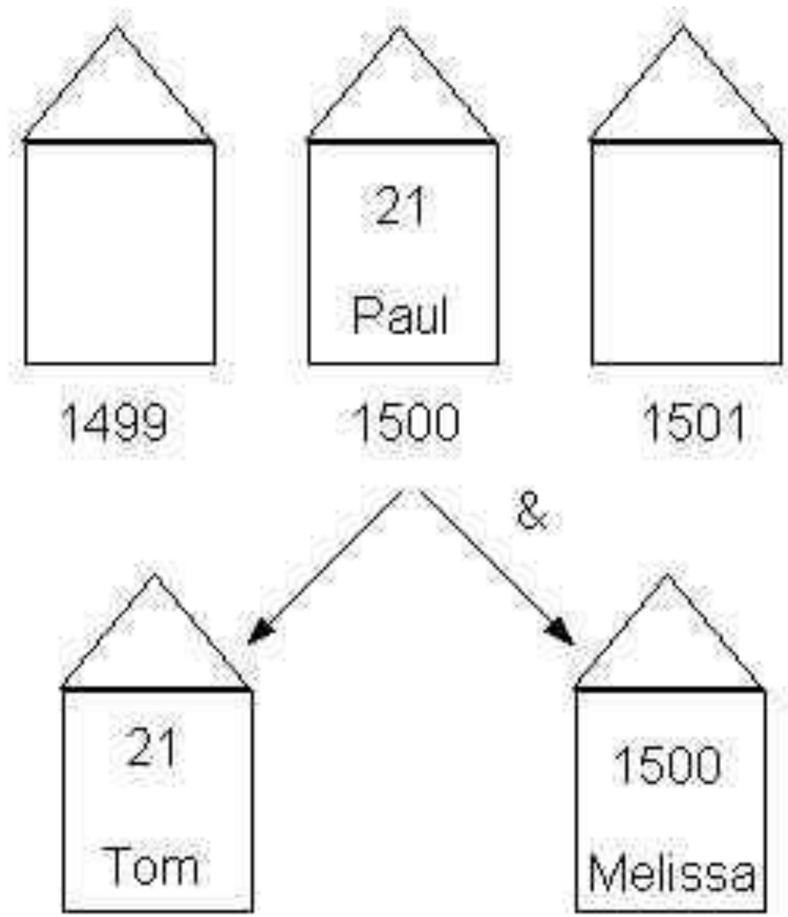
Lesson 3 : More on C++

Lesson Coverage

- Pointers revisit
- Function pointer
- Pointer to member functions
- Memory Management
  - new
  - delete
- Exception Handling
- Namespace
- string class
- Virtual base class
- Templates
- File Handling
  - Reading from file
  - Writing to file

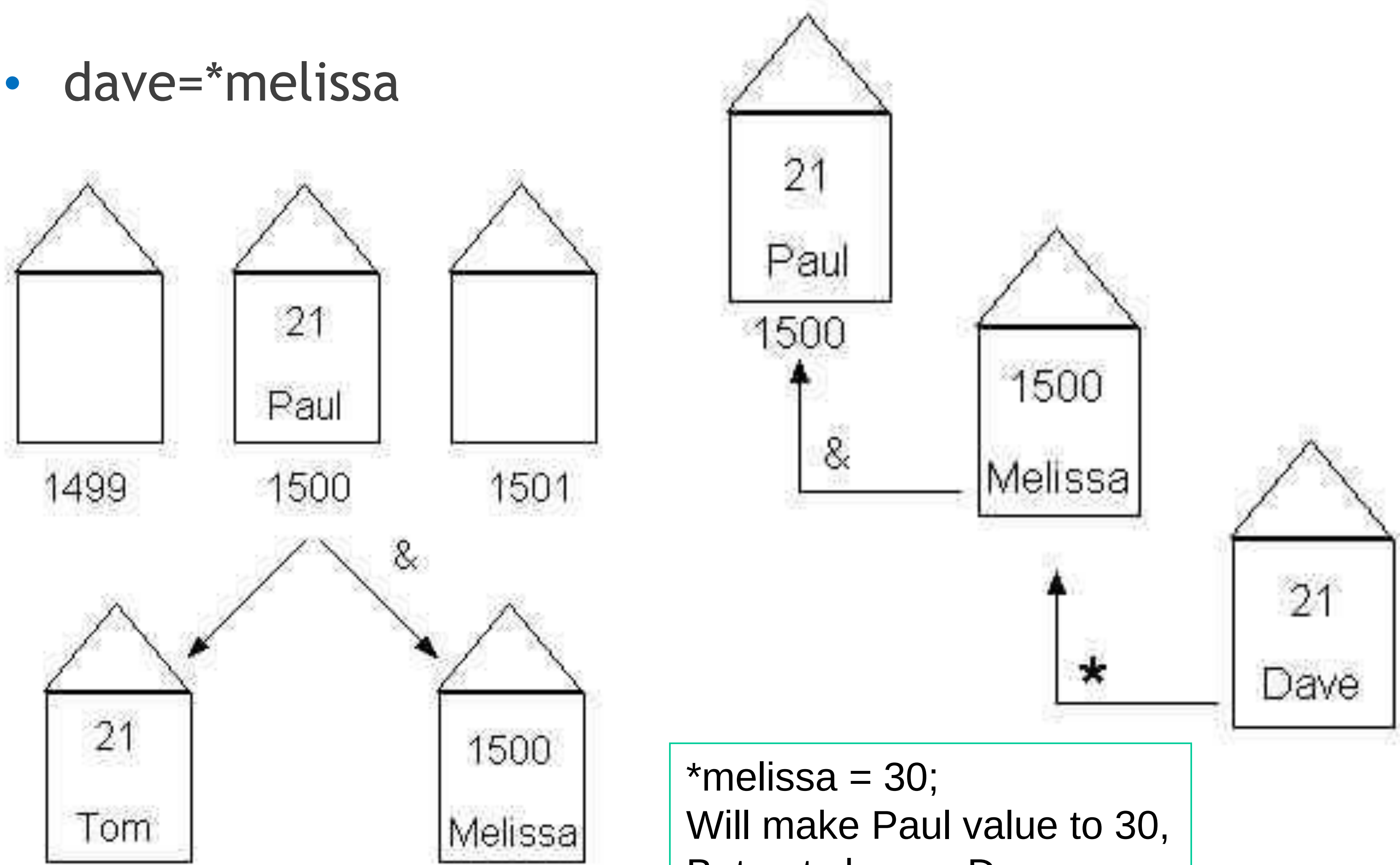
Pointers

- Understanding - &
- paul = 21;
  tom = paul;
  melissa = &paul;



Pointers

- Understanding - &
- dave=*melissa



*melissa = 30;
Will make Paul value to 30,
But not change Dave

Pointers

- Creating pointer variable

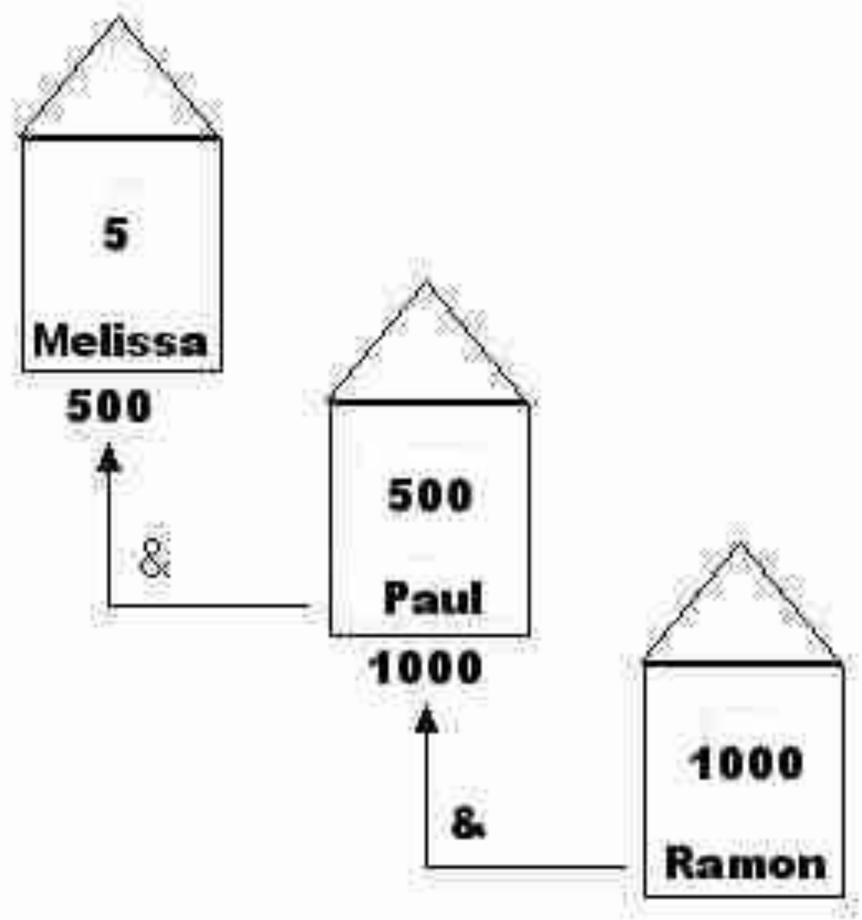
```
int a = 5, b = 10;
int *p1, *p2;
p1 = &a;
p2 = &b;
*p1 = 10;
p1 = p2;
*p1 = 20;
cout<<“a = “<< a;
cout<<“b = “<< b;
```

Pointers to Array

```
int a[5] = {1,2,3,4,5};

int *p1;

p1 = &a[1]; // gets address of this element

cout<<"*p1 = "<<*p1;

p1++; // point to the next element

cout<<"*p1 = "<<*p1;
```

Pointer of pointer

```
int **ramon;
int *paul;
int melissa = 5;
paul = &melissa;
ramon = &paul;
printf("ramon = %d\n", ramon);
printf("&paul = %d\n", &paul);
printf("*ramon = %d\n", *ramon);
printf("&melissa = %d\n", &melissa)
printf("**ramon = %d\n", **ramon);
```



Pointer as Argument

```
// function declaration:
double getAverage(int *arr, int size);
 int main (){
int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
   avg = getAverage( balance, 5 ) ;
   cout << "Average value is: " << avg << endl;
}

double getAverage(int *arr, int size){
  int    i, sum = 0;
  double avg;
  for (i = 0; i < size; ++i)  {
    sum += arr[i];
   }
  avg = double(sum) / size;
  return avg;
}
```

Returning a pointer

```
// function to generate and retrun
numbers.
int * getRandom( )
{
  static int  r[10];

for (int i = 0; i < 10; ++i)
  {
    r[i] = i;
    cout << r[i] << endl;
  }

  return r;
}
```

```
// main function to call above defined
function.
int main ()
{
   // a pointer to an int.
   int *p;

   p = getRandom();
   for ( int i = 0; i < 10; i++ )
   {
       cout << "*(p + " << i << ") : ";
       cout << *(p + i) << endl;
   }

   return 0;
}
```

Function pointer

- **Why FP?**
- **Callbacks?**

```
void create_button( int x, int y, const char *text, void (*foo)(int)){
Foo(2)
}
```

Create_buttin(10,20,"abc",&my_int_func);

- **Declaring function pointers**

```
void (*foo)(int);  // this point to similar declaration functions

void *(*foo)(int *);

typedef float (*MyFuncPtrType)(int, char *);
MyFuncPtrType my_func_ptr;
```

Function pointer - example

```
void my_int_func(int x)
{
    cout<<x ;
}
int main()
{
    void (*foo)(int);
    foo = &my_int_func;

    /* call my_int_func (note that you do not need to write (*foo)(2) )
    */
    foo( 2 );
    /* but if you want to, you may */
    (*foo)( 2 );

    return 0;
}
```

Pointer to member function

- Declaring pointers-to-member-functions
  - You declare a pointer-to-member-function just like a pointer-to-function, except that the syntax is a different:
- Declaring a pointer to an ordinary function:
  - return_type (*pointer_name) (parameter types)
- Declaring a pointer to a member function:
  - return_type (class_name::*pointer_name) (parameter types)

Pointer to member function

- Setting a pointer-to-member-function
  - You set a pointer-to-member-function variable by assigning it to the address of the class-qualified function name, similar to an ordinary function pointer.
- Setting an ordinary function pointer to point to a function:
  - pointer_name = function_name;   // simple form
  - pointer_name = &function_name; // verbose form

- Setting a member function pointer to point to a member function:
  - pointer_name = &class_name::member_function_name;

Pointer to member function

```
class A {
void f();
void g();
};
void A::f(){              // declare pmf as pointer to A member function,
                          // taking no args and returning void
void (A::*pmf)();
pmf = &A::g;               // set pmf to point to A's member function g
// call the member function pointed to by pmf points on this object
(this->*pmf)();           // calls A::g on this object
}
```

- **Using a typedef**

```
typedef void (A::*A_pmf_t)();
void A::f()
{
A_pmf_t p = &A::g;
(this->*p)();                 // calls A::g on this object
}
```

Three Categories of Memory

- Static: storage requirements are known prior to run time; lifetime is the entire program execution

- Run-time stack: memory associated with active functions
  - Structured as stack frames (activation records)

- Heap: dynamically allocated storage; the least organized and most dynamic storage area

Static Data Memory

- Simplest type of memory to manage.
- Consists of anything that can be completely determined at compile time; *e.g.*, global variables, constants (perhaps), code.
- Characteristics:
  - Storage requirements known prior to execution
  - Size of static storage area is constant throughout execution

Run-Time Stack

- The stack is a contiguous memory region that grows and shrinks as a program runs.
- Its purpose: to support method calls
- It grows (storage is allocated) when the activation record (or stack frame) is pushed on the stack at the time a method is called (activated).
- It shrinks when the method terminates and storage is de-allocated.

Run-Time Stack

- The stack frame has storage for local variables, parameters, and return linkage.
- The size and structure of a stack frame is known at compile time, but actual contents and time of allocation is unknown until runtime.
- How is variable lifetime affected by stack management techniques?

Heap Memory

- Heap objects are allocated/deallocated dynamically as the program runs (not associated with specific event such as function entry/exit).
- The kind of data found on the heap depends on the language
  - Strings, dynamic arrays, objects, and linked structures are typically located here.
  - Java and C/C++ have different policies.

Heap Memory

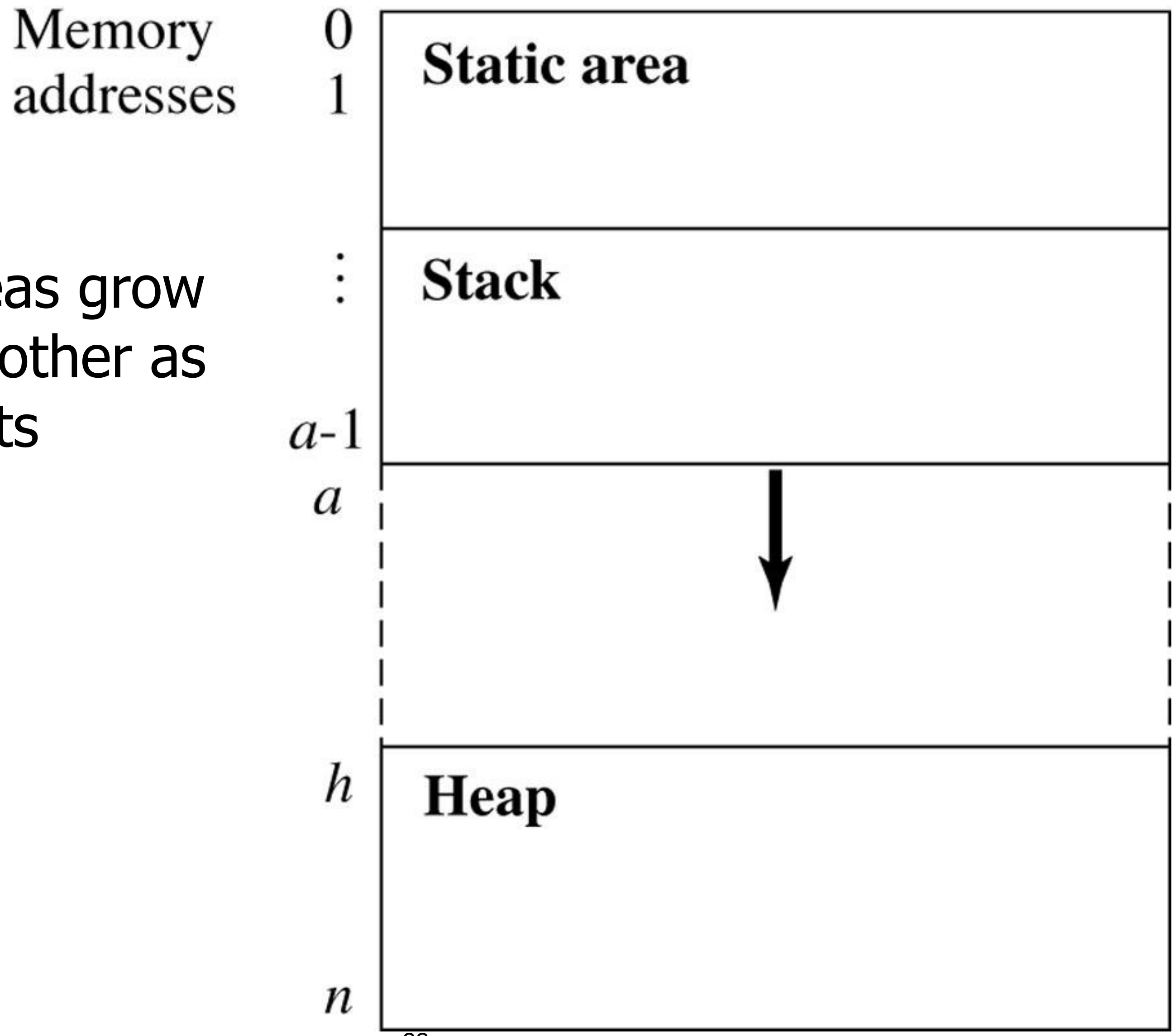
- Special operations (e.g., `malloc, new)` may be needed to allocate heap storage.
- When a program deallocates storage (`free, delete`) the space is returned to the heap to be re-used.
- Space is allocated in variable sized blocks, so deallocation may leave "holes" in the heap (fragmentation).
  - Compare to deallocation of stack storage

Heap Management

- Some languages (e.g. C, C++) leave heap storage deallocation to the programmer
  - `Delete`

The Structure of Run-Time Memory

These two areas grow towards each other as program events require.



Dynamic Memory Management

- **Memory allocation in C:**
  - `calloc()`
  - `malloc()`
  - `realloc()`
  - Deallocated using the free() function.

- **Memory allocation in C++**
  - using the new operator.
  - Deallocated using the delete operator.

Memory Management Functions - 1

- `malloc(size_t size);`
  - Allocates size bytes and returns a pointer to the allocated memory.
  - The memory is not cleared.

- `free(void * p);`
  - Frees the memory space pointed to by p, which must have been returned by a previous call to **`malloc()`**, **`calloc()`**, or **`realloc()`**.
  - If **`free(p)`** has already been called before, undefined behavior occurs.
  - If p is NULL, no operation is performed.

Methods to do Dynamic Storage Allocation - 1

- Best-fit method –
  - An area with *m* bytes is selected, where *m* is the smallest available chunk of contiguous memory equal to or larger than *n*.

- First-fit method –
  - Returns the first chunk encountered containing *n* or more bytes.

- Prevention of fragmentation,
  - a memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful.

Methods to do Dynamic Storage Allocation - 2

- Memory managers
  - return chunks to the available space list as soon as they become free and consolidate adjacent areas.

- Boundary tags
  - Help consolidate adjoining chunks of free memory so that fragmentation is avoided.

- The size field simplifies navigation between chunks.

Checking Return Codes from malloc()

```
int *i_ptr;
i_ptr =
  (int*)malloc(sizeof(int)*nelements_wanted);
if (i_ptr != NULL) {
   i_ptr[i] = i;
}
else {
   /* Couldn't get the memory - recover */
}
```

Improperly Paired Memory Management Functions

- Memory management functions must be properly paired.
- If new is used to obtain storage, delete should be used to free it.
- If `malloc()` is used to obtain storage, `free()` should be used to free it.
- Using `free()` with new or `malloc()` with `delete()` is a bad practice.

Improperly Paired Memory Management Functions – Example Program

```
1. int *ip = new int(12);
   . . .
2. free(ip); // wrong!
3. ip = static_cast<int *>(malloc(sizeof(int)));
4. *ip = 12;
   . . .
5. delete ip; // wrong!
```

•

Failure to Distinguish Scalars and Arrays

- The new and delete operators are used to allocate and deallocate scalars:

```
Widget *w = new Widget(arg);
delete w;
```

- The new [] and delete [] operators are used to allocate

  and free arrays:

```
w = new Widget[n];
delete [] w;
```

Shallow & Deep copy

- **Deep copy** involves using the contents of one object to create another instance of the same class.
  - In a deep copy, the two objects may contain the same information but the target object will have its own buffers and resources.

- **Shallow copy** involves copying the contents of one object into another instance of the same class thus creating a mirror image.
  - Owing to straight copying of references and pointers, the two objects will share the same externally contained contents of the other object to be unpredictable.

Shallow & Deep copy

```
struct sample
{
    char * ptr;
}
void shallowcpy(sample & dest, sample & src)
{
    dest.ptr=src.ptr;
}
void deepcpy(sample & dest, sample & src)
{
    dest.ptr=malloc(strlen(src.ptr)+1);
    memcpy(dest.ptr,src.ptr);
}
```

Shallow & Deep copy

```
class MyString{
private:
    char *m_pchString;
    int m_nLength;
public:
    MyString(char *pchString="“)    {
      m_nLength = strlen(pchString) + 1;
       m_pchString= new char[m_nLength];
                                                            // Copy the parameter into our internal buffer
        strncpy(m_pchString, pchString, m_nLength);
                                                             // Make sure the string is terminated
        m_pchString[m_nLength-1] = '\0';
    }

    ~MyString() {                                    // destructor
                                                     // We need to deallocate our buffer
        delete[] m_pchString;

        // Set m_pchString to null just in case
        m_pchString = 0;
    }

    char* GetString() { return m_pchString; }
    int GetLength() { return m_nLength; }
};
```

Shallow & Deep copy

```
MyString cHello("Hello, world!");



   MyString cCopy = cHello; // use default copy constructor
 // cHello goes out of scope here

std::cout << cCopy.GetString() << std::endl; // this will crash
```

Shallow & Deep copy

```
// Copy constructor
MyString::MyString(const MyString& cSource){
    // because m_nLength is not a pointer, we can shallow copy it
    m_nLength = cSource.m_nLength;
    // m_pchString is a pointer, so we need to deep copy it if it is non-null
    if (cSource.m_pchString)    {
        // allocate memory for our copy
        m_pchString = new char[m_nLength];

        // Copy the string into our newly allocated memory
        strncpy(m_pchString, cSource.m_pchString, m_nLength);
    }
    else
        m_pchString = 0;
}
```

Exceptions

- Exception
- throw

```
try{

}catch(){

}
```

- Creating user-define exceptions

Exception

```
int main () {
  char myarray[10];
  try
  {
    for (int n=0; n<=10; n++)    {
      if (n>9) throw "Out of range";
      myarray[n]='z';
     cout<<myarray[n]<<" ";
    }
  }
  catch (char const * str)  {
    cout << "Exception: " << str << endl;
  }
  return 0;
}
```

Throwing Exceptions

```
double division(int a, int b)
{
   if( b == 0 )   {
      throw "Division by zero condition!";
   }
   return (a/b);
}
int main (){
   int x = 50;
   int y = 0;
   double z = 0;

   try {
     z = division(x, y);
     cout << z << endl;
   }catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

User-defined exception

```
#include <exception>
using namespace std;
struct MyException : public exception{
  const char * what () const throw ()  {
    return "C++ Exception";
  }
};
int main(){
  try  {
    throw MyException();
  }
  catch(MyException& e)  {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
    //Other errors
  }
}
```

Namespace

- Used to avoid **Namespace collision**
- A Namespace allows the same name to be used in different contexts without conflicts arising.
- eg:

```
//vendor1.h
    class String {
          ...
    };

//vendor2.h
    class String {
          . . .
    };
```

Namespace

```
#include "vendor1.h"
#include "vendor2.h“
```

This usage in a program will trigger a compilation error because class String is defined twice

This problem can be solved by using Namespace

```
namespace Vendor1 {                namespace Vendor2 {
        class String {             class String {
            ...                    . . .
        };   }                  }; }
```

using Keyword

Syntax:

```
using namespace name;
using name::member;
```

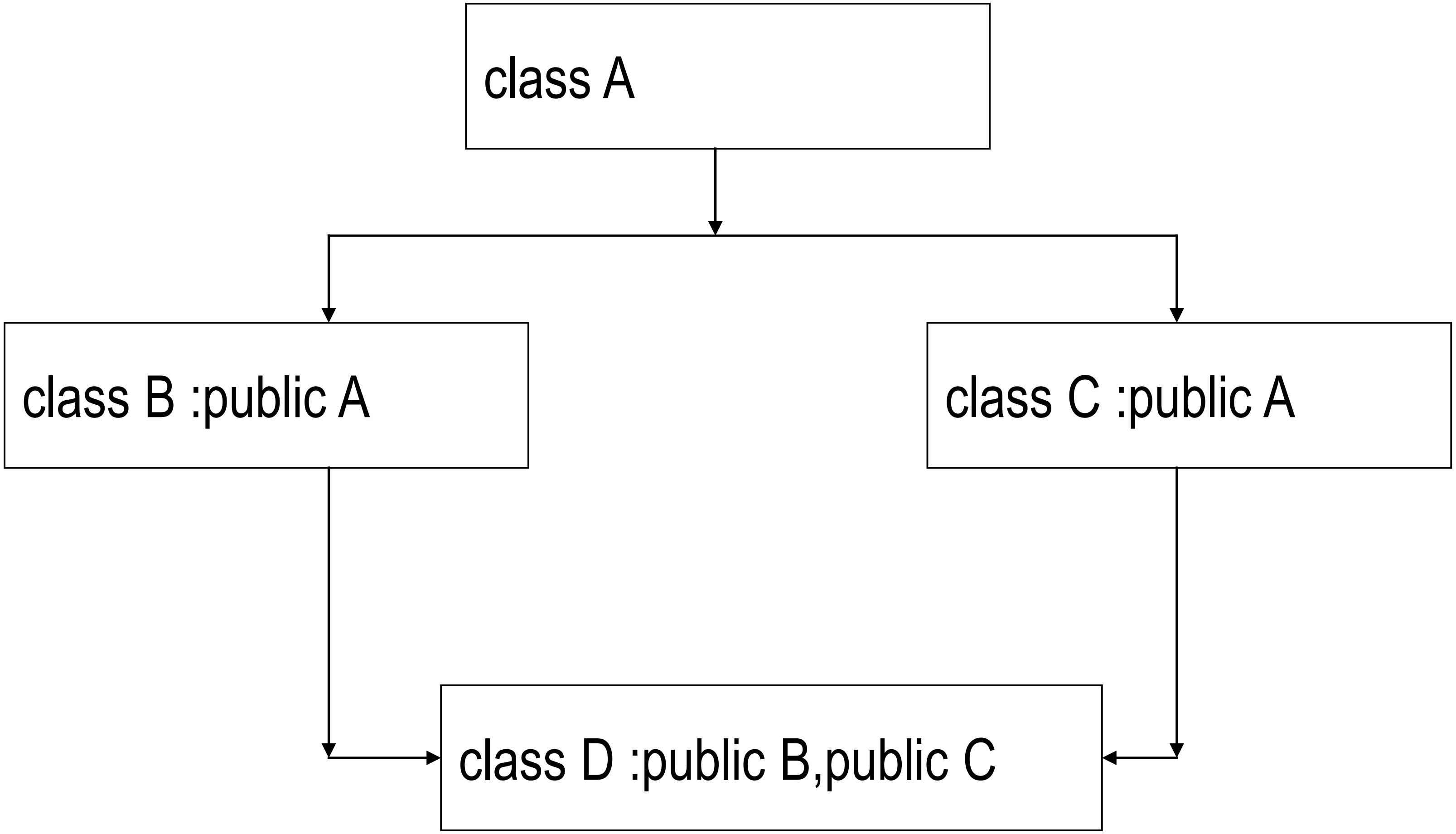
Example:

```
using SampleNameSpace::lowerbound; // only lowerbound is visible
lowerbound = 10; // OK because lowerbound is visible

using namespace SampleNameSpace; // all members are visible
upperbound = 100; // OK because all members are now visible
```

std Namespace

```
int main()
{
    int val;
    std::cout << "Enter a number: ";
    std::cin >> val;
    std::cout << "This is your number: ";
    std::cout << std::hex << val;
    return 0;
}
```

Virtual base class



All members of class A are inherited twice by class D

Virtual base class

```
class student
{
   int rno;
  public:
   void getnumber()
   {
           cout<<"Enter Roll No:";
           cin>>rno;
   }
   void putnumber()
   {
           cout<<"\n\n\tRoll No:"<<rno<<"\n";
   }
};
```

Virtual base class

```
class test:virtual public student
{

  public:
   int part1,part2;
   void getmarks()
   {
            cout<<"Enter Marks\n";
            cout<<"Part1:";
            cin>>part1;
            cout<<"Part2:";
            cin>>part2;
   }
   void putmarks()
   {
            cout<<"\tMarks Obtained\n";
            cout<<"\n\tPart1:"<<part1;
            cout<<"\n\tPart2:"<<part2;
   }
};
```

Virtual base class

```
class sports:public virtual student
{

  public:
    int score;
    void getscore()
    {
            cout<<"Enter Sports Score:";
            cin>>score;
    }
    void putscore()
    {
            cout<<"\n\tSports Score is:"<<score;
    }
};
```

Virtual base class

```
class result:public test,public sports
{
    int total;
  public:
   void display()
   {
      total=part1+part2+score;
      putnumber();
      putmarks();
      putscore();
      cout<<"\n\tTotal Score:"<<total;
   }
};
```

Virtual base class

```
void main()
{
   result obj;
   clrscr();
   obj.getnumber();
   obj.getmarks();
   obj.getscore();
   obj.display();
   getch();
}
```

String class

String class is part of the C++ Standard Library

To use the string class, #include the header file:

#include <string>

Constructors:

- string ()
- string (other_string)
- string (other_string, position, count)
- string (count, character)

String class-constant Member functions

- const char * data ()
  - returns a C-style null-terminated string of characters representing the contents of the string

- unsigned int length ()
- returns the length of the string

- unsigned int size ()
  - returns the length of the string (i.e., same as the length function)

- bool empty ()
- returns true if the string is empty, false otherwise

Operators defined for String

- *Assign* =

  string s1;

  string s2;

  ...

  s1 = s2; // the contents of s2 is **copied** to s1

- *Append* +=

  string s1("abc");

  string s2("def");

  ...

  s1 += s2; // s1 = "abcdef" now

Operators defined for String

- *Indexing* []
  string s("def");
  char c = s[2]; // c = 'f' now
  s[0] = s[1]; // s = "eef" now

- *Concatenate* +
  string s1("abc");
  string s2("def");
  string s3;
  s3 = s1 + s2; // s3 = "abcdef" now

Operators defined for String

- *Equality* ==

```
string s1( "abc" );
string s2( "def" );
string s3( "abc" );
...
bool flag1 = ( s1 == s2 ); // flag1 = false now
bool flag2 = ( s1 == s3 ); // flag2 = true now
```

Operators defined for String

- *Inequality* !=
  - the inverse of equality

- *Comparison* <, >, <=, >=
  - performs case-insensitive comparison

```
string s1 = "abc";
string s2 = "ABC";
string s3 = "abcdef";
...
bool flag1 = ( s1 < s2 ); // flag1 = false now
bool flag2 = ( s2 < s3 ); // flag2 = true now
```

Member functions

- void swap (other_string)
  - swaps the contents of this string with the contents of other_string.
  string s1("abc");
  string s2("def");
  s1.swap(s2); // s1 = "def", s2 = "abc" now

- string & append (other_string)
  - appends other_string to this string, and returns a reference to the result string.

Member functions

- string & insert (position, other_string)
  - inserts other_string into this string at the given position, and returns a reference to the result string

- string & erase (position, count)
  - removes count characters from this string, starting with the character at the given position. If count is omitted (only one argument is given), the characters up to the end of the string are removed. If both position and count are omitted (no arguments are given), the string is cleared (it becomes the empty string)

Member functions

- unsigned int find (other_string, position)
  - finds other_string inside this string and returns its position. If position is given, the search starts there in this string, otherwise it starts at the beginning of this string.

- string substr (position, count)
  - returns the substring starting at position and of length count from this string

Templates in C++

- Replacement for function overloading
- Creating generic functionality
- Creating Libraries
  - Class Templates
  - Function Templates

Function Templates

```
template<class T>
T id(T x ){
  return x;
}

template<class T1, class T2>
T2 ex(T1 x, T2 y){
  cout<<x<<endl;;
  return y;
}

template<class T>
int f(){
  T y;
  return y;
}
```

```
main(){
  cout<<id(12)<<endl;
  cout<<id("string")<<endl;
  cout<<id(true)<<endl;
  cout<<ex(1,2)<<endl;
  cout<<ex("abc","def")<<endl;
  cout<<f<double>()<<endl;; //must supply
the type of the template
  cout<<f<int>()<<endl;;
}
```

Function Template

```
#include<string>

template<class T>
T mymin(T a,T b){
    return (a<b?a:b);
}
char* mymin(char *a,char *b){
    return (strcmp(a,b)<0?a:b);
}

int main()
{
  double a=3.56,b=8.23;
 int a=3,b=8;

  char* s1="Hello";
  char* s2="Good";

  cout<<" "<<a<<", "<<b<<" the min is "<<mymin(a,b)<<endl;
  cout<<" "<<s1<<", "<<s2<<" the min is "<<mymin(s1,s2)<<endl;
}
```

Class Templates

```
template<class X>
class input{
  X data;
public:
input(char *s,X min,X max);
  //...
};
template<class X>
input<X>::input(char *s,X min,X max){
   do{
      cout<<s<<":";
      cin>>data;
     }while(data<min||data>max);
}
```

```
main()
{
  input<int>i("enter int",0,10);
  input<char>c("enter char",'A','Z');
  cout<<"--end--";
  return 0;
}
```