Basic Introduction to Classes & Objects

```
// Example showing use of class and objects
#include <iostream>
using namespace std;       // explained in next slide
#define SIZE 100
// This creates the class stack.
class stack {
  int stck[SIZE];              // default scope is private
  int tos;
public:                        //  scope : discussed later
  void init();                //  Function declaration
  void push(int i);
  int pop();
};                  //  we can declare an object here also but then it
                    //  will be a  global object
void stack::init() // function defination
{
  tos = 0;
}
void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}
int stack::pop()
{
  if(tos==0) {
    cout << "Stack underflow.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}
```

```
int main()
{
  stack stack1, stack2;          // create two stack objects
  stack1.init();                 // we access the class members
                                 // with the help of . operator
  stack2.init();
  stack1.push(1);
  stack2.push(2);
  stack1.push(3);
  stack2.push(4);
  cout << stack1.pop() << " ";
  cout << stack1.pop() << " ";
  cout << stack2.pop() << " ";
  cout << stack2.pop() << "\n";
  return 0;
}
```

What is namespace ?

A mechanism to group declarations that logically belong to each other

```
namespace physics {
  class vector;
  class unit;
  class oscillator;
  void sort(const vector& value);
}

namespace electronics {
  void sort(const vector& value);
  class oscillator;
}

namespace graphics {
  void sort(const vector& value);
  class unit;
}
```

Provides an easy way for logical separation of parts of a big project

Basically a 'scope' for a group of related declarations

How do I use **namespaces** ?

```
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) { return (a+b)/2.; }
}

namespace foobar {
  double mean(const double& a, const double& b) { return (a*a+b*b)/2.; }
}

int main() {
   double x = 3;
   double y = 4;

   double z1 = physics::mean(x,y);
   std::cout << "physics::mean(" << x << "," << y << ") = " << z1
             << std::endl;

   double z2 = foobar::mean(x,y);
   std::cout << "foobar::mean(" << x << "," << y << ") = " << z2
             << std::endl;
   return 0;
}
```

physics::mean

foobar::mean

Use "::" to specify the namespace

Defined in iostream

Common Errors with namespaces

```
// namespaceBad.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) {
    return (a+b)/2.;
  }
}

int main() {

   double x = 3;
   double y = 4;

   double z1 = mean(x,y); // forgot the namespace!
   cout << "physics::mean(" << x << "," << y << ") = " << z1
             << std::endl;

   return 0;
}
```

If you forget to specify the namespace the compiler doesn't know where to find the method

using namespace directive

```
// namespace2.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) {
    return (a+b)/2.;
  }
}

using namespace std; // make all names in std namespace available!

int main() {

   double x = 3;
   double y = 4;

   double z1 = physics::mean(x,y);
   cout << "physics::mean(" << x << "," << y << ") = " << z1
             << endl;

   return 0;
}
```

Provide default namespace for un-qualified names

Compiler looks for std::cout and std::endl;

Be careful with `using` directive!

```
// namespaceBad2.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) { return (a+b)/2.; }
}

namespace foobar {
  double mean(const double& a, const double& b) { return (a*a+b*b)/2.; }
}

using namespace foobar;
using namespace physics;
using namespace std;

int main() {
   double x = 3;
   double y = 4;

   double z1 = mean(x,y);
   double z2 = mean(x,y);

   return 0;
}
```

Ambiguous use of method mean!

Is it in foobar or in physics?

Some tips on using directive

Never use using directive in header files!

- These can be included in other files that do not want to use default namespaces specified by you!
- Limit use of using directive to the scope you need

```
/ namespace3.cc
include <iostream>

amespace physics {
 double mean(const double& a, const double& b) {
   return (a+b)/2.;
 }


oid printMean(const double& a, const double& b) {
 double z1 = physics::mean(a,b);

 using namespace std; // using std namespace within this method!
 cout << "physics::mean(" << a << "," << b << ") = " << z1 << endl;


nt main() {

  double x = 3;
  double y = 4;
  printMean(x,y);

  cout << "no namespace available in the main!" << endl;
  return 0;
```

Namespace defined only within printMean

Another Example on Scopes

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

using std::cout;
using std::endl;

int main() {

  double x = 1.2;

  cout << "in main before scope, x: " << x << endl;

  { // just a local scope
     x++;
     cout << "in local scope before int, x: " << x << endl;

     int x = 4;
     cout << "in local scope after int, x: " << x << endl;
  }

  cout << "in main after local scope, x: " << x << endl;

  return 0;
}
```

Another way to declare ONLY classes we are going to use instead of entire namespace

Another Example on Scopes

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

using std::cout;
using std::endl;

int main() {

  double x = 1.2;

  cout << "in main before scope, x: " << x << endl;

  { // just a local scope
     x++;
     cout << "in local scope before int, x: " << x << endl;

     int x = 4;
     cout << "in local scope after int, x: " << x << endl;
  }

  cout << "in main after local scope, x: " << x << endl;

  return 0;
}
```

Another way to declare ONLY classes we are going to use instead of entire namespace

Changed value of x from main scope

Define new variable in this scope

Back to the main scope

Constructors & Destructors

- How to initialize data members (specially private members) of an object → use constructors
  - A constructor is a special function that is a member of a class and has the same name as that of the class
  - Constructor do not return values & so constructor function has no return type
  - How constructor is called :: called when the object is declared. An object's constructor is called once for global or static local objects. For non static local objects, the constructor is called each time the object declaration is encountered.

- DESTRUCTORS
  - Syntax :: ~Class_name
  - Purpose : In many circumstances an object will need to perform some actions when it is destroyed.
  - An object may need to de-allocate memory that it had previously allocated or it may need to close a file that it had opened. So in C++, it is destructor's function that handles deactivation events.
  - when destructor is called :: when an object is destroyed, it's destructor is automatically called.
    - when an object is destroyed :: local objects are created when their block is entered & destroyed when the block is left
    - Global objects are destroyed when program terminates
  - like constructor, destructors also do not have return values

Example constructor & destructor

```
// Using a constructor and destructor.
#include <iostream>
using namespace std;
#define SIZE 100
// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();          // constructor declared
  ~stack();       // destructor declared
  void push(int i);
  int pop();
};

// stack's constructor function
stack::stack()     // no return value
{
  tos = 0;
  cout << "Stack Initialized\n";
}

// stack's destructor function
stack::~stack()        // no return value
{
  cout << "Stack Destroyed\n";
}

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}
```

```
int stack::pop()
{
  if(tos==0) {
    cout << "Stack underflow.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}


int main()
{
  stack a, b;  // create two stack objects
  a.push(1);
  b.push(2);
  a.push(3);
  b.push(4);
  cout << a.pop() << " ";
  cout << a.pop() << " ";
  cout << b.pop() << " ";
  cout << b.pop() << "\n";
  return 0;
}
```

output :
stack initialized
stack initialized
3 1 4 2
stack destroyed
stack destroyed

Classes & Objects

Classes

- Class is a *logical abstraction* whereas Object has *physical existence*
  - Object is an instance of a class

```
Class  class_name {
            Data members & member functions      // by default private
             int a;
             int b;
             access_specifier :

/ * public, private or protected, once an  access specifier has been used, it
    remains in effect until either another access specifier is encountered or
the end of the class declaration is reached data members & member
functions */
                    .
                    .                                     .
                    .
            access_specifier :

            data members & member functions
                    .
                    .                                     .
                    .
  } object_list ;
```

There are few restriction that apply to data members

- a non static member variable cannot have an initializer

```
Class ABC {
        int A = 2;     // not allowed
       }
```

- No member can be object of the class that is being declared. Although a member can be a pointer to the class that is being declared.

```
Class ABC {
        ABC BOX;          //not allowed
        ABC *next_box;     // allowed
       }
```

- No member can be declared as auto, extern or register (but can be static)

Access Specifiers :

- private :: private to class only
- Public:: access specifier allows functions or data to be accessible to other parts of your program.
- protected :: will be discussed later

Structures & Classes

- **Structure in C++ can have member functions**
- **Only difference b/w structure & Classes is that data members are public by default in structure. Whereas in class, data members are private by default.**

```
struct mystr {
        int a;    // public by default
        int b;
        void showa();
        void showb(char *s);
       };
void mystr :: showa() {
                    .
                    .
          }
```

- **C-like structure (without member functions) are generally referred as POD (plain old data)**
- **General rule is to use classes where necessary & use structure only in the POD style.**

Union & Classes

- **Like in C, the C++ union data members share the same location in memory**
- **like structure, union members are public by default**
- **like in C++ structure, Union can have their own constructors or destructors**

```
#include <iostream>
using namespace std;
union swap_byte {
  void swap();
  void set_byte(unsigned short i);
  void show_word();
  unsigned short u;
  unsigned char c[2];
};
void swap_byte::swap()
{
  unsigned char t;
  t = c[0];
  c[0] = c[1];
  c[1] = t;
}
void swap_byte::show_word()
{
  cout << u;
}
void swap_byte::set_byte(unsigned short i)
{
  u = i;
}
int main()
{
  swap_byte b;
  b.set_byte(49034);
  b.swap();
  b.show_word();
  return 0;
}
```

note : what is POD union : : C like union without the member functions

<u>Restriction in C++ on Unions</u>
→ union cannot inherit any other class
→ Union can not be a base class
→Union can not have virtual functions
→Union can not have static member variables
→Union can not use a reference member variables
→ A union cannot have as a member any object that overloads the = operator.
→ no object can be a member of a union if the object has an explicit constructor or destructor.

Note : In c we need to declare a variable of union as

union swap_byte b;

But in C++ we need not use union keyword. Same is the case with structures, enum & classes

Anonymous Union

- It is a special type of Union
- Anonymous Union does not include a type name and thus no objects of the anonymous Union can be declared
- The variables of Anonymous Union can be directly referred with out the normal dot operator.
- The scope of Anonymous Union Variable will be same as other local variables and thus anonymous Union variable name should not collide/conflict with the local variable names

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
            // define anonymous union
  union {
    long l;
   double d;
    char s[4];
  } ;
          // now, reference union elements directly
  l = 100000;
  cout << l << " ";
  d = 123.2342;
  cout << d << " ";
  strcpy(s, "hi");
  cout << s;
  return 0;
}
```

<u>RESTRICTIONS ON ANONYMOUS UNIONS</u>

- **All restriction involving unions apply to anonymous Unions also**
- **no member function allowed**
- **Anonymous Unions can not contain private or protected elements**
- **Global Anonymous unions must be specified as static**

Friend Function

- **A friend function has access to all private and protected members of the class for which it is a friend**

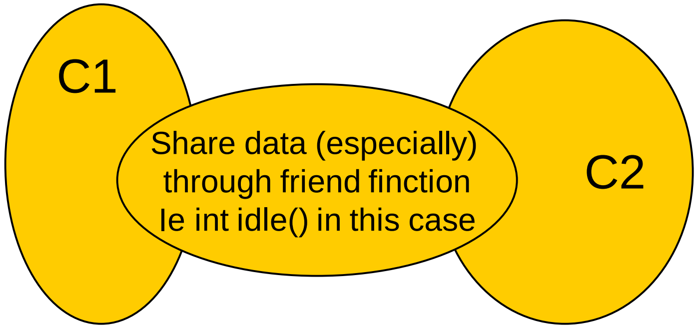
```
class B{
            int b1,b2,b3;
        public:
            void func (int a1, int a2);
            friend int funcglobal (int x1);
}

void B::func (int a1, int a2){
                -------
                -------
}

int funcglobal(int x1){
     B ob;
     ob.b1 = x1;     // here you can access the private(or protected ) members of B, like b1,b2,b3

     ob.b2 = x1;                                              }
```

- **<u>USE OF FRIEND FUNCTION</u>**
  - **Useful for overloading certain types of operators**
  - **Friend functions make the creation of some types of I/O function easier.**
  - **Friend function may also be desirable where two or more classes may contain members that are interrelated relative to other parts of the program.**

Example Friend Function

```
#include <iostream>
using namespace std;
const int IDLE = 0;
const int INUSE = 1;
class C2;
     /* forward declaration (required here since a class
        can't be referred to until it has been declared ) */
class C1 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
        /* or friend int C2 :: idle(C1 a) → if idle happened
           to be defined in class C2 */
};
class C2 {
  int status; // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};
void C1::set_status(int state)
{
  status = state;
}
void C2::set_status(int state)
{
  status = state;
}
int idle(C1 a, C2 b)
{
  if(a.status || b.status) return 0;
  else return 1;
}
```

```
int main()
{
  C1 x;
  C2 y;
  x.set_status(IDLE);
  y.set_status(IDLE);
  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";
  x.set_status(INUSE);
  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";
  return 0;
}
```



C1 & C2 are unrelated classes

Friend Classes

- **One class can be friend of another class. When this is the case, the friend class and all of its member function have access to the private members defined within the class.**

```
Class C1 {
        int a, b;
        public:
        friend class min;
};
class min {
        func1();      // all these can access the private members of C1
        func2();      //   ---------do---------------------------
        func3();      //  -----------do-----------------------
}
```

- **Note: It is critical to understand that when one class is a friend of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the friend class**

Inline Function

- **Normal function call → push arguments in stack, save various registers, transfer control and do vice versa on return → so more time consuming process**
- **Inline function → code is expanded inline so efficient (faster) but will result in larger code size → so generally very small functions are inlined.**
- **Note :: inline is actually is just a request, not a command to the compiler. The compiler can choose to ignore it. It is common for a compiler not to inline a recursive function. If a function cannot be inlined, it will simply be called as a normal function**

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{
  return a>b ? a : b;
}
int main()
{
  cout << max(10, 20);  // == cout << (10>20 ? 10 : 20)
  cout << " " << max(99, 88);   // == cout << " " << (99>88 ? 99 : 88)
  return 0;
}
```

```
// note → inline function may be class member
//functions
Class myclass {
                    _
                    _
                public :
                 void xyz()
                  int show ()
}

inline void myclass :: xyz() {
                    _
```

note : when a function is defined inside a class declaration it is automatically made into an inline function (if possible). It is not necessary (but not an error) to precede its declaration with the inline keyword

```
class myclass {
                    -
                    -

           void xyz(int i, int j) { a = i ; b = j; }          // automatic inline as defined within the class
            void show ()  { cout << a << " " << b <<  "\n"  // automatic inline as defined within the class
};
```

Constructors - inlined

- **constructors and destructors functions may also be inlined, (by default), If defined within their class or explicitly**

Parameterized constructors

```
Class myclass {
                    _
                    _
                    _
              int a, b;
          public :
             myclass(int i, int j) {a = i ; b = j ; } // this will be inline
   };
```

Constructor with one parameter

```
Class myclass {
                    int a ;
                         -
                         -
                         -
          public :
                    myclass(int i) { a = i; } // this is inline
    }
```

call to constructor ::

myclass ob(3,4); → this is generally used
or
myclass ob = myclass(3,4);

call to constructor with single argument

myclass ob = 99 →actually for compiler it is equivalent to

myclass ob = myclass(99);

we can also use → myclass ob(99); //generally this is used

myclass ob = 99 → is possible b/s whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.

Static Data Members

```
Class shared {
            Static int a ;    // only a declaration so no space is
                              //allocated as such
               -
               -
        };

      int shared :: a; // definition, so now space is allocated

/* note : static  → means you are telling the compiler that only
one copy of that variable will exist & that all objects of the class
will share that variable. All static variables (by default) are
initialized to 0 before the first object is created.*/

int main()
        {
          shared :: a = 99  // we have not created any object, still
                            // we can use static variable a

          count << shared :: a ; // prints 99
          shared x;
          count << x.a;     // prints 99, use of static variable a
                            //with object
           return 0;
         }
```

```
/* Another use of static member variable is to keep track
of the no. of objects of a particular class type that are in
existence */
listing 26//static
#include <iostream>
using namespace std;
class Counter {
public:
  static int count;
  Counter() { count++; }
  ~Counter() { count--; }
};
int Counter::count;
void f();
int main(void)
{
  Counter o1;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  Counter o2;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  f();
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  return 0;
}
void f()
{
  Counter temp;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  // temp is destroyed when f() returns
}
```

Static Member function

- Static member function can be called with class name or with object name

```
#include <iostream>
using namespace std;
class cl {
  static int resource;
public:
  static int get_resource();
  void free_resource() { resource = 0; }
};
int cl::resource;              // define resource
int cl::get_resource()
{
  if(resource)  return 0;    // resource already in use
  else {
    resource = 1;
    return 1;                  // resource allocated to this object
  }
}
int main()
{
  cl ob1, ob2;
  /* get_resource() is static so may be called independent
     of any object. */
  if(cl::get_resource()) cout << "ob1 has resource\n";
  if(!cl::get_resource()) cout << "ob2 denied resource\n";
  ob1.free_resource();

/* we can also call static member function getr_resource
using object syntax also */
  if(ob2.get_resource())
  cout << "ob2 can now use resource\n";
  return 0;
}
```

Restrictions on static member functions

- **They can only directly refer to other static member of the class (but they can use non static global functions and data)**
- **A static member function can not use this pointer**
- **There cannot be a static and non static versions of the same function**
- **A static member function may not be virtual**
- **Static member functions cannot be declared as constant or volatile**

/* Static member functions have limited application, but one good use of them is to "preinitialize" private static data before any object is actually created. */

```
#include <iostream>
using namespace std;
class static_type {
  static int i;
public:
  static void init(int x) { i = x; }
  void show() { cout << i; }
};
int static_type::i; // define i
int main()
{
  // init static data before object creation
  static_type::init(100);
  static_type x;
  x.show(); // displays 100
  return 0;
}
```

WHEN CONSTRUCTORS AND DESTRUCTORS ARE EXECUTED

- A local object constructor function is executed when the objects declaration statement is encountered. Objects destructor is called when the life time of the object is about to end.
- Global objects have their constructor function execute before main() begins execution. Global constructors are executed in order of their declaration, within the same file. You cannot know the order of execution of global constructors spread among several files. Global destructors execute in reverse order after main() has terminated

```
#include <iostream>
using namespace std;
class myclass {
public:
  int who;
  myclass(int id);
  ~myclass();
} glob_ob1(1), glob_ob2(2);
myclass::myclass(int id)
{
  cout << "Initializing " << id << "\n";
  who = id;
}
myclass::~myclass()
{
  cout << "Destructing " << who << "\n";
}
int main()
{
  myclass local_ob1(3);
  cout << "This will not be first line displayed.\n";
  myclass local_ob2(4);
  return 0;
}
```

o/p :
Initializing 1
Initializing 2
Initializing 3
This will not be the first line displayed
Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1

THE SCOPE RESOLUTION OPERATOR ::

```
int i;  // global i
void f()
{
  int i; // local i
  ::i = 10; // now refers to global i
  .
  .
  .
}
```

Local Classes

```
#include <iostream.h>
using namespace std;
void f();
int main()   {
                f();
                return 0;
}
void f()  {
                class myclass
                {
                        int i;
                  public:
                        void put_i (int x)  {
                                i = x;
                          }
                        int get_i (){
                        return i;
                        }
                }ob;
        ob.put_i (10);
        cout<<ob.get_i();
}
```

Restriction on local classes

- **Member functions must be defined within the class declaration in local classes (i.e. they are inline)**
- **The local class may not use or access local variables of the function in which it is declared but can use the static local variable of the function.**
- **Local class may access type names & enumerators defined by the enclosing function.**
- **No static variable may be declared inside a local class.**
- **Because of these problems local classes are not common in C++**

Passing Objects to functions

- Objects are passed by value
- So new object needs to be created when passing objects as parameter
- So whether constructor is called → no →rather it will be a bitwise copy { if constructor is called then it is as good as creating a new object but we want to actually pass the old object → so no constructor should be called
- Although destructor for the object (parameter) will be called when the function ends → necessary → as separate memory is occupied by the parameter object (object's copy) & it need to be freed
- Bit wise copy : Exact bit by bit copy →but will lead to problem of side effect in case like → if an object used as an argument allocates memory & frees that memory when it is destroyed, then its local copy (parameter object) inside the function will free the same memory when its destructor is called & this will leave the original object damaged & effectively useless → what is the solution → use copy constructor (to be discussed later)

```
// Passing an object to a function.
#include <iostream>
using namespace std;
class myclass {
  int i;
public:
  myclass(int n);
  ~myclass();
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};
myclass::myclass(int n)
{
  i = n;
  cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
  cout << "Destroying " << i << "\n";
}
void f(myclass ob);
int main()
{
  myclass o(1);
  f(o);
  cout << "This is i in main: ";
  cout << o.get_i() << "\n";
  return 0;
}
void f(myclass ob)
{
  ob.set_i(2);
  cout << "This is local i: " << ob.get_i();
  cout << "\n";
}
```

RETURNING OBJECTS

- When an object is returned by a function (object local to function), a temporary object is automatically created that holds the return value
- It is this object that is actually returned by the function.
- After the value has been returned, this object is destroyed.
- The destructor of temporary object may cause side effects

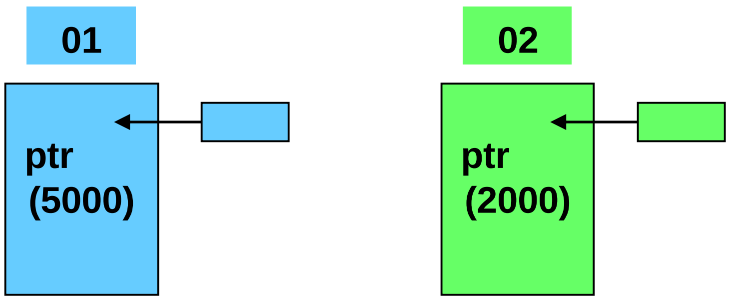
Side effects→ like → if the object returned by the function has a destructor that free dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it.

soln: →overloading the assignment operator

```
// Returning objects from a function.
#include <iostream>
using namespace std;
class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};
myclass f();        // return object of type myclass
int main()
{
  myclass o;
  o = f();
  cout << o.get_i() << "\n";
  return 0;
}
myclass f()
{
  myclass x;
  x.set_i(1);
  return x;
}
```

Object Assignment

```
// Assigning objects.
#include <iostream>
using namespace std;
class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};
int main()
{
  myclass ob1, ob2;
  ob1.set_i(99);
  ob2 = ob1; // assign data from ob1 to ob2 (bitwise copy)
  cout << "This is ob2's i: " << ob2.get_i();
  return 0;
}
```



o1=o2 // o1.ptr =2000

Note : We can avoid bit – by – bit copy by overloading the assignment operator & define some other assignment procedure

Const member functions & constant object

Const function arguments

- **If we want that *argument passed by reference should not be changed* then we should define these argument in function declaration & definition as const.**

```
void func (int &a, const int &b); //declaration
        void func (int &a, const int &b)
        {
                ---------

        //definition
                ---------
        }
```

- No problem in passing a const argument by value, because the function can't modify the original value anyway.

***Const member function* : member function that guarantees that it will never modify any of its class member data**

```
void constfunc () const  //the const will come in both member
                         //function declaration & definition

distance add_dist(const distance &d2) const          //***
{
   distance temp;
   feet = 0; //error as feet is class data member of invoking
            //object
  d2.feet = 0; //error, as d2 is passed as constant
  temp.feet = temp.feet + d2.feet; // allowed
  return temp;
}
```

****** if you want that the argument passed by reference to this function should not be modified then it should be declared as const reference argument in the member function.***

constant objects

- when an object is declared as const then you can't modify it. It also means that you can only use those member function with this object which are declared as const as they are the only one that guarantees not to modify it.

```
Class Distance {
                :
                :
        Public :
        Distance(…, …) { ……. }
        Void getlist() { …. }
        Void showdist() const
                { . . . . . .  }
 };
int main(){
        const Distance ft(300, 0)
        // ft.getlist(); → not allowed
        ft.showdist(); // allowed
                        }
```

Containership & Nested Classes

Containership :: A class has an object of different class as its data member

```
Class A{
    -
    -
};
class B{
    A obA; // define obA as an object of
           //class A
};
```

→member function of B can access the public members of A with the help of dot (.) operator.

Nested Class :: it is possible to define one class within another class.

```
class A  {
        class B             // Nested
                            //class
                {

                }
        }
```

→A nested class is only valid within the scope of the enclosing class.
→Nested classes are rarely used & are not generally required.

Arrays, Pointers, References & Dynamic Memory Allocation

Array of Objects

```
// →In case of classes with no constructor function

Class C1 {
                    .
                    .
 };

int main()
{
 C1 ob[3]; // array of 3 objects
                    .
                    .
 }
```

```
//→In case of constructor with two arguments and no constructor
//→with single argument
Class C1 {
                    int h;
                    int i;
                 public :
                   C1(int j, int k) { h = j; i=k; }
};
int main()  {
C1 ob[3] = { C1(1, 2), C1(3, 4), C1(5, 6) };
C1 ob[3] // invalid → can't have uninitialized array here
}
```

```
//→In case of class with constructor of single argument

Class C1 {
               int i;
           public :
              C1(int j) { i = j; }
                                             .
                                             .
  };
int main()
{
C1 ob[3] = { 1, 2, 3 };
// the above statement is equivalent to C1 ob[3] = { C1(1), C1(2), C1(3) }
// and C1 ob[3];  → will be an error here because we don't have a zero
// argument constructor
                         .
                         .
  }
```

```
//→Creating initialized & uninitialized array
// → if we want to create uninitialized array along with
//initialized array then along with other constructor, we
//have to explicitly create a parameter less constructor
//also.
Class C1 {
                int i;
            public :
                C1() {i = 0;}// parameter less constructor
                C1(int j) { i = j;}
};
int main()  {
C1 a1[3] = {3, 5, 6 }; // initialized array  → valid statement
C1 a2[4]; // un initialized array → valid statement
 }
```

Pointers to objects

- When accessing members of a class, given a pointer to an object, use the arrow (→) operator instead of the dot operator.

```
class C1  {
            int i;
        public:
            C1 (int j) {  i=j;   }
            int get_i()  {  return i;  }
};
int main()  {
            C1 ob(88), *p;
            p = &ob;
            cout  <<  p→get_i();
            return 0;
}

OR

int main()  {
            C1 ob[3] = {1,2,3};
            C1 *p;
            int i;
            p = ob; // get start of array
            for (i=0; i<3;i++)   {
            cout << p→get_i() << "\n";
            p++; //point to the next element of array
 // in this case p++ →then the pointer points to the next
//object in the array
            }
return 0;
}
```

```
//→ we can assign the address of a public member of an
//object to a pointer and then access that member by using
//the  pointer

#include <iostream>
using namespace std;
class cl {
                public:
                        int i;
                        cl(int j) { i=j; }
};
int main()   {
  cl ob(1);
  int *p;
  p = &ob.i; // get address of ob.i
  cout << *p; // access ob.i via p
  return 0;
}
```

```
//→Type checking C++ pointer
//→you may assign one pointer to another only if the two pointer
// types are compatible

int *pi;
float *pf;
pi = pf; //error - type mismatch
// Note:- Of course, you can override any type in compatibilities using a cast
pi = (int *) pf; //Ok
```

this pointer

- **When a member function is called, it is automatically passed an implicit argument that is pointer to the invoking object (that is, the object on which the function is called)**

```
Class C1{
                int i;
        public:
                get_i() {
                return this.->i; // if we say return i then also it
means
                                //the same
                }
};
int main()  {
                C1 ob1;
                cout<<ob1.get_i();
}

class alpha  {
                int data;
         public:
                -------
                -------
                alpha &operator = (alpha &a)  {
                        data=a.data
                        return *this;
                }
};
int main()  {
                alpha a1(37);
                alpha a2,a3;
                a3 = a2 = a1; // cascading of = was possible b/s
                                //this pointer was returned
}
```

Usefulness of this

- any member function can find out the address of the object of which it is a member
- *this* is actually useful when operators are overloaded and whenever a member function must utilize a pointer to the object that invoked it
- Note:- *friend function* are not member of a class and therefore are *not passed a this pointer*
- *Note: static member function* do not have a this pointer

Pointers to a derived type

- let B is base class & D be the derived class
- A pointer of type B* (ie say B*ptr) may also point to an object of the type D
- A pointer of type *D (ie say D *ptr1) may not be able to point to an object of type B
- Although we can use a base pointer to point to a derived object, we can access only the members of the derived type that were imported/inherited from the base. that is, we won't be able to access any members added by the derived class {although we can cast a base pointer into derived pointer & gain full access to the entire derived class}

```
class base {
        int i;
    public:
        void set_i(int num) { i=num; }
        int get_i() { return i; }
};
class derived: public base {
        int j;
    public:
        void set_j(int num) { j=num; }
        int get_j() { return j; }
};
```

```
int main()  {
  base *bp;
  derived d;
  bp = &d; // base pointer points to derived object
           // access derived object using base pointer
  bp->set_i(10);
  cout << bp->get_i() << " ";
/* The following won't work. You can't access element of a
derived class using a base class pointer. */

bp->set_j(88);     // error ---  ((derived *)bp)->set_j(88);//ok
cout << bp->get_j();   //error--cout << ((derived *)bp) >get_j();//ok
 return 0;
}
```

Pointers to a derived type

- **It is important to remember that *pointer arithmetic is relative to the base type of the pointer*. for this reason, when a base pointer is pointing to a derived object, incrementing the pointer does not cause it to point to the next object of the derived type. instead it will point to what it thinks is the next object of the base type.**

```
// This program contains an error.
#include <iostream>
using namespace std;
class base {
                    int i;
            public:
                    void set_i(int num) { i=num; }
                    int get_i() { return i; }
};
class derived: public base {
                    int j;
            public:
                    void set_j(int num) {j=num;}
                    int get_j() {return j;}
};
int main()
{
  base *bp;
  derived d[2];
  bp = d;
  d[0].set_i(1);
  d[1].set_i(2);
  cout << bp->get_i() << " ";
  bp++; // relative to base, not derived
  cout << bp->get_i(); // garbage value displayed
  return 0;
}
```

➢The use of base pointer to derived type is most useful when creating run-time polymorphism (through the mechanism of virtual functions)

References

- it is basically an implicit pointer or in other words it is an alias (different name) for a variable
- **Independent references**
  - **An independent reference must be initialized when they are created (you need something to point to)**

```
#include <iostream>
using namespace std;
int main() {
  int a;
  int &ref = a;
/* independent reference mu                                          aration */
  a = 10;
  cout << a << " " << ref << "
/* a & ref both refer to the sa
  ref = 100;
  cout << a << " " << ref << "\n"; // a=100, ref=100
  int b = 19;
  ref = b; // this puts b's value into a
  cout << a << " " << ref << "\n"; // a=19, ref = 19
  ref--; // this decrements a  it does not affect what ref refers to
  cout << a << " " << ref << "\n"; // a=18,ref=18
  return 0;
}
```

```
int i=4;

int &j; //error

Int &j=4;//ok

J=i;
```

```
int n[10];
int &x = n[10]; // x is alias for n[10]
char &a = '\n'; // initialised reference
                  //to a literal
int x;
int *p = &x;
int &m = *p; // so new m refers to x
               //(which is pointed to by
               // pointer P)
int &n = 50;
// creates a int object with value 50
//and name n
```

Reference

```
// Manually create a call-by-reference using a pointer.
#include <iostream>
using namespace std;
void neg(int *i);
int main() {
  int x;
  x = 10;
  cout << x << " negated is ";
  neg(&x);
  cout << x << "\n";
  return 0;
}
void neg(int *i)
{
  *i = -*i;
}
```

```
// Call by reference using reference variable
// Use a reference parameter.
#include <iostream>
using namespace std;
void neg(int &i); // i now a reference
int main()
{
  int x;
  x = 10;
  cout << x << " negated is ";
  neg(x); // no longer need the & operator
  cout << x << "\n";
  return 0;
}
void neg(int &i)// at the time of call → int &i = x
{
  i = -i; // i is now a reference, don't need *
}
```

```
#include <iostream>
using namespace std;
void swap(int &i, int &j);
int main()
{
  int a, b, c, d;
  a = 1;
  b = 2;
  c = 3;
  d = 4;
  cout << "a and b: " << a << " " << b << "\n";
  swap(a, b); // no & operator needed
  cout << "a and b: " << a << " " << b << "\n";
  cout << "c and d: " << c << " " << d << "\n";
  swap(c, d);
  cout << "c and d: " << c << " " << d << "\n";
  return 0;
}
void swap(int &i, int &j)
{
  int t;
  t = i; // no * operator needed
  i = j;
  j = t;
}
```

Reference

- **<u>Passing reference to object</u>**
  - **when we pass an object by reference, no bit-wise copy of the object is made. this means that no object used as a parameter is destroyed when the function terminates, and so the parameter's destructor is not called.**

```
#include <iostream>
using namespace std;
class cl {
                 int id;
           public:
                 int i;
                 cl(int i);
                 ~cl();
                 void neg(cl &o) { o.i = -o.i; }
                  // no temporary object is created
};
cl::cl(int num) {
  cout << "Constructing " << num << "\n";
  id = num;
}
cl::~cl() {
  cout << "Destructing " << id << "\n";
}
int main() {
  cl o(1);
  o.i = 10;
  o.neg(o);
  cout << o.i << "\n";
  return 0;
}
```

Output:- constructing 1
-10
destruction 1

→note:- destructor is called only once ie when main() terminates

→ Passing object by reference is faster than passing object by value. as there is no need of making any copy & putting on to the stack

Reference

Function returning reference can be used on the left side (as well as on right side) of an assignment statement. In other words when a function returns a reference, the function call can exist in any context where a reference can exist

```
//the program replaces hello there with helloxthere
#include <iostream>
using namespace std;
char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
  replace(5) = 'X'; // assign X to space after Hello
  cout << s;
  return 0;
}
char &replace(int i)
{
  return s[i];
}
```

Note :: One thing to be careful about when returning reference is that the object being referenced to should not go out of scope after the function terminates. That is do not try to return local variable by reference

→ Reference is an implicit constant pointer ie once a reference variable has been defined to refer to a particular variable, it can not refer to any other variable. that is, once the variable and the reference are linked they are tied together inseparably.

→ We can create reference to a pointer

```
char *p = "Hello";
char * &q = p;
```

→ A variable can have multiple reference. changing the value of one of them effects a change in all others.

Restriction on Reference

- **You can't reference another reference (ie you can't obtain the address of a reference)**
- **You can not create array of references**
- **You can not create a pointer to a reference**
- **You can not reference a bit- field**
- **You can not have a null reference**
- **A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter or a return value.**

References --- Q & A

- **Q 1. When should we make a call by reference?**
- **Hint:- A call by pointer or a call by reference is useful in two situations:**
  - **When we intend to change the values of actual arguments through the called function.**
  - **When we want to save memory by preventing the creation of large structure variable that are being passed to the function**

  To achieve these purpose reference offer a cleaner and more elegant way as compared to pointers, as with references, we are not required to use the *and → operators.

- **Q 2. Is reference a pointer?**
  - **Hint:- A reference is a const pointer . hence once initialized a reference cannot be made to refer to another variable. Unlike a pointer a reference gets automatically de-referenced.**

- **Q3. Is reference to a reference like shown below allowed?**

```
int i;
int &j = i;
int &k = j;   // reference to a reference.
```

  - **Hint : no, because when we try to assign a reference to a reference the new reference starts referring to the same variable the first reference is referring to**

- **Q 4. Can we create a pointer to a reference?**
  - **Hint:- No. This can be explained with the help of following code:**

```
int i;
int &p = i;
int *j = &p; // not a pointer to a reference
```

  - **Here it seems that j is a pointer to the reference p, but actually it is pointing to the variable i. this is because a reference is automatically de-referenced, i.e., &p internally becomes &*p. Thus in j what gets stored is address of i.**

References --- Q & A

- **Q 5. How would the compiler interpret the following statements?**

```
int i = 9;
int &p = i;
int &q = p;
```

  - **Hint:- The compiler would interpret statement as shown below.**

```
int i = 9;
int *const p = &i; // address of 9
int *const q = &*p;  // address of 9
```

  - **As a reference is nothing but a const pointer the address of the variable gets stored in a reference. Hence in the second statement address of i gets stored in const pointer p. And as reference are automatically de-referenced p becomes &*p in the third statement.**

References --- Q & A

- **Q 7 What are the advantages of pointer over reference?**
  - Hint:- Reference being a const pointer cannot be reassigned. On the other hand pointers can be reassigned. This is shown in the following example:

```
main () {
        int i,j;
        int *p = &i;
        p = &j;
}
```

  - →Also, arithmetic operation cannot be performed on a reference.This is shown in the following example:

```
main ()  {
        int i;
        int &r = i;
        r++; // will not increment r but will increment the value of i
        int *p = &i;
        p++;
}
```

  - Here r++ would not increment the value of r. But it will increment value of i. This means that when an arithmetic operation is performed on a reference, it gets performed on a referent. But when p++ is done, the value of p is incremented.
- **Q 8 . Why should we not return a reference or an address of a local variable.**
  - Hint:- When we return a reference of a local variable, the variable would die once control returns to the calling function. Hence, calling function would be referencing to a variable that no longer exists.
- **Q 9. Can we create a reference to an array?**
  - Hint:- Yes, a reference to an array is allowed. For example:

```
int a[] = {3,7,6,9,5};
int(&p)[5] = a;  // reference to an array
```

Note: int (*pt)[7] :: pointer to an array of 7 integers

int *pt[7] ; Array of 7 pointers

- **Q 10 Why is it so that when we print the address of a reference the address of a referent gets printed?**

```
int i;
int &r = i;
cout << &r;
```

→When we write &r it is actually treated as &*r which is nothing but address of the values stored in r, that is address of i.

References --- Q & A

- State whether the following statements are True or False:
- It is possible to create an array of reference.
  - ANS: False, A reference is not an object. Hence we cannot find address of a reference, nor can we create an array of references. And for the same reason we can not have pointer to the reference also.

- Once a reference is tied with a variable it cannot be tied with another variable
  - ANS: True. A reference being a const pointer, once initialized its value cannot be changed.

- A variable can be tied with several references.
  - ANS: True. This is because there is no limitation on storing the address of a variable in multiple pointers. Since references are const pointer this works. For example:

```
int a = 10;
int &b = a;
int &c = a;
```

- In c++ a function call can occur even on the left-hand side of an assignment operator.
  - ANS: True. If a function returns a reference its call can exist on the left-hand side of an assignment operator. This is shown in the following code:

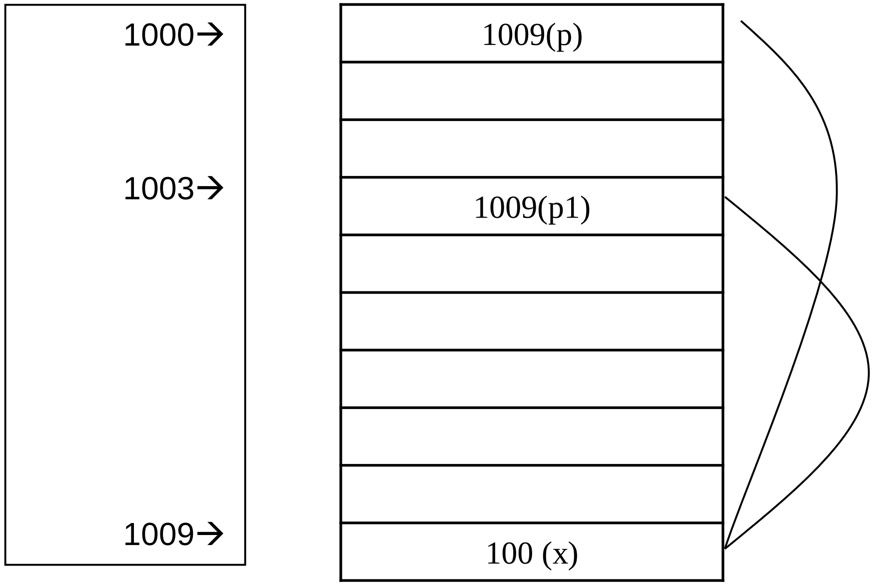
```
# include <iostream.h>
int i;
void main() {
int &fun();
fun() = 10;
}

int &fun() {
    i = 2;
    return i;
}
```

Here, the function fun() returns a reference to a integer variable i. the returned reference replace the functions calls. Hence the statement becomes (reference of i) = 10. hence i would be assigned a value 10, as can be verified by output of cout.

- It is unsafe to return a local variable by reference.
  - ANS: True. As soon as the function returns, local variables die. If a reference (or address)of a local variable is returned, it means we would be referring to the dead variable.

Dynamic memory allocation in C

```
 main()
{
double x=100, y;
int *p, *p1;
p =&x;
p1=p;
printf("%d", x);→100
printf("%d", *p); →100
printf("%d", *p1); →100
printf("%u", p); →1009
printf("%u", p1); →1009

       }
```



- Pointer arithmetic
  - We can add or subtract integer to or from pointers (p1 = p1 + 3)
  - We can subtract same type of pointer from another same type of pointer (to find the no. of elements separating the two pointers in the array)
  - Pointers can be compared (<, <=, >, >=, ==, !=)
  - Pointes can not be multiplied or divided (p1/3, p*3, p1/p2, p1*p2 → all these are illegal operations on the pointer)
  - Can not add two pointers →(p1 + p2 →illegal)
  - We can not add or subtract type float or double to or from pointers (p + 2.14, p - 2.14 →illegal)

Dynamic memory allocation in C

Arrays & Pointers

Name of the array is actually the address of the first element of the array

```
int num[] = { 24, 34, 12, 44, 56, 77 };
//num == &num[0] == 1000
num[i] ==*(num + i) == *( i + num) == i[num]
```

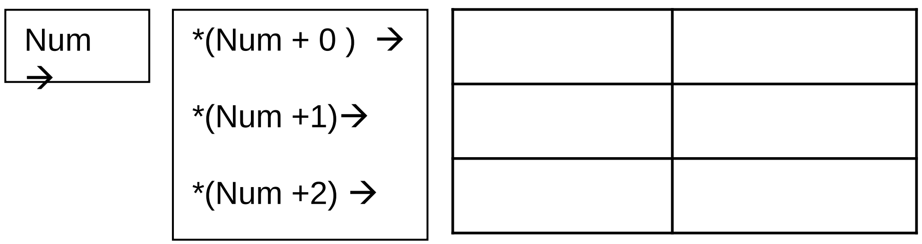
Address	Array elements
(&num[0]) →1000	24 (num[0])
(&num[1]) → 1004	34 (num[1])
(&num[2]) →1008	12 (num[2])
(&num[3]) →1012	44 (num[3])
(&num[4]) →1016	56 (num[4])
(&num[5]) →1020	77 (num[5])

2 –D Array

```
main() {
int num[3][2];
int i,j;
for(i=0; i <= 2; i++)
{
      for(j =0; j <= 1; j++)
      {
            scanf("%d", &num[i][j]);
       }
}
}
```

Address	Array elements
(&num[0][0]) →1000	24 (num[0][0])
(&num[0][1]) → 1004	34 (num[0][1])
(&num[1][0]) →1008	12 (num[1][0])
(&num[1][1]) →1012	44 (num[1][1])
(&num[2][0]) →1016	56 (num[2][0])
(&num[2][1]) →1020	77 (num[2][1])

Dynamic memory allocation in C



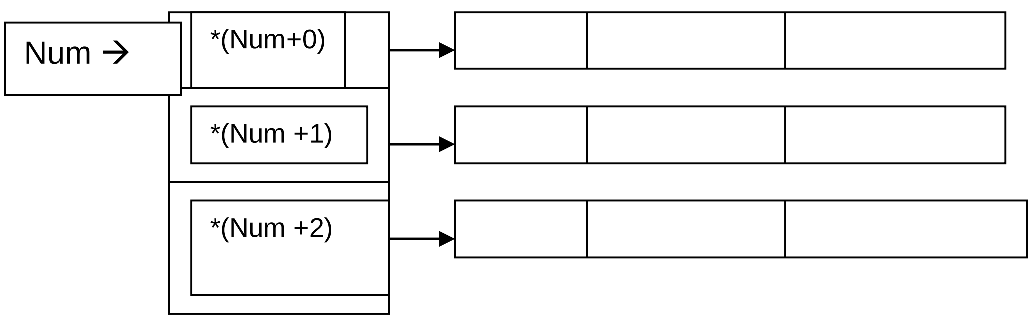
num[2][1] == *(num[2] + 1) == * (*(num + 2) + 1)

num + i == pointer to the ith row

***(num + i) → pointer to the 1st element in the ith row**

***(num + i) + j → pointer to the jth element in the ith row**

***(*(num + i) + j) → value stored in num[i][j]**



- **Num is the address of first element . now first element here is a pointer to array of int**

int num[3][3] → int(*num)[3] == int ptr[][3]

Dynamic memory allocation in C

```
main() {
int num[3][2];
call_func(num);
}

call_func(int ptr [][3]) or call_func(int (*ptr)[3])
{
.
.
.
}
```

- NULL → defined in stdio.h and stddef.h
- basically means that NULL assigned pointer can not point to any object by mistake
- NULL pointer is a predefined position in memory. If this memory location contents are tried to be changed or some pointer try to access this area then NULL pointer assignment error is generated.

Dynamic memory allocation in C

Malloc

→ To allocate memory dynamically

```
#include <stdlib.h> / #include <alloc.h>
void *malloc(unsigned int no_of_bytes)


char *p;
p = (char *)malloc(1000); // allocate space for 1000 bytes
 // & returns the pointer the first element of the block

int *p;
p = (int *)malloc(50 * sizeof(int)); // 50*2 if size of int = 2 bytes

void free(void *p) // p is a pointer to a memory that was
                         //previously allocated using malloc()
main() {
                  char *s;
                  char *fun();
                  s = fun();                    }
char *fun() {
                  char buffer[30];
                  strcpy(buffer, "hello");
                  return(buffer);               }
```

→ The scope & life time of buffer ends as the fun() returns . so problem

soln.

→ static char buffer[30];

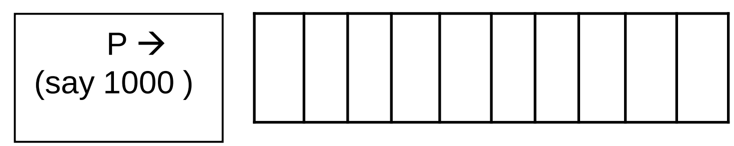
→ or buffer is a global pointer

→ or

```
    char *ptr
    ptr = (char *)malloc(30);
    strcpy(ptr, ". . . . . . .. . . .");
    return(ptr);
```

Q. How to allocate a 1-D array of int

```
main() {
    int *p,i;
    p = (int *)malloc(10*sizeof(int));
     for(i=0;i<10;i++) {
                           p[i] = i; //*(p+i)
                           printf("%d", p[i]);
                         }
}
```



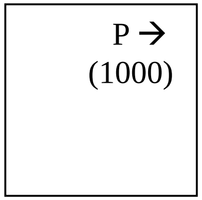
Dynamic memory allocation in C

Q. how to dynamically allocate a 2-D array of int

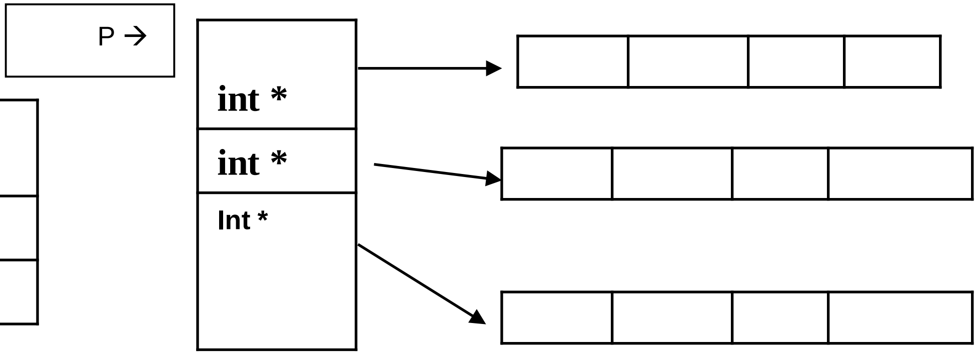
```
main() {
      int *p, i, j;
      p = (int *) malloc(3*4*sizeof(int))
      for(i=0; i < 3; i++ )  {
                 for(j =0; j < 4; j++)
                {
                      p[i*4 + j] = i;
                      printf("%d", p[i*4 + j]);
                }
                pintf("\n");
          }
}
```

Q. how to do proper 2 – D dynamic array allocation so that we can use arr[i][j]

```
main()  {
int **p, i, j;
p = (int **) malloc(3*sizeof(int *)); //  3 = no. of rows
for(i = 0; i < 3; i++ )
p[i] = (int * )malloc(4 * sizeof(int));// 4 = no. of columns
for(i = 0; i < 3 ; i++ )
{
               for(j=0; j < 4; j++)
               {
               p[i][j] = i;// *(*(p+i) + j)
               printf("%d", p[i][j]);
               }
  printf("\n");
  }
}

//to free memory free each p[i] first →free(p[i])
// then free(p);
```



0 /1000	0 /1002	0/1004	0/1006
1/1008	1/1010	1/1012	1/1014
2/1016	2/1018	2/1020	2/1022



C++'s Dynamic memory allocation operators

- ***new & delete***
  - *malloc () & free ()* of C for dynamic allocation of memory are also available in C++ for compatibility with C but is not recommended to be used in C++ programs.

- ***new*** :- The *new* operator allocates memory & returns a pointer to the start of it.
- ***delete*** :- The *delete* operator frees memory previously allocated using new.

```
p_var = new type;
delete p_var;
```

- ***new*** - If no memory is available for allocation (from the heap) a ***bad-alloc*** exception is raised, which your program should catch & handle otherwise the program will be terminated abnormally.

- Older compilers (some still do) return *Null* on failing to allocated memory.

```
#include <iostream>
#include <new>
using namespace std;
int main() {
  int *p;
  try {
    p = new int;          // allocate space for an int
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }
  *p = 100;
  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";
  delete p;
  return 0;
}
```

C++'s Dynamic memory allocation operators

Advantages of *new* & *delete* over *malloc* & *free*

- ***new*** **automatically allocates enough memory to hold an object of the specified type & so no need of sizeof operator**
- ***new*** **automatically returns a pointer to the specified type & we don't need to do explict type cast as in malloc ()**
- **Both *new & delete* can be overloaded, allowing you to create customized allocation system**

Initializing allocated memory

```
p_var = new var_type (initializer)
    int *p;
    p = new int (87); // initialize to 87
```

Allocating array

```
p_var = new array_type[size];
delete[] p_var;    // [] inform that array
                   //is being released
int *p
p = new int[10];   // allocate 10 integer
                   // array
delete[] p;        // release the array
```

Note: when arrays are allocated using news, we can not give them any initial value

C++'s Dynamic memory allocation operators

Allocating objects:

We can allocate objects dynamically by using new. when we do this, an object is created & a pointer is returned to it. when object is created dynamically using new, its constructor function (if it has one) is called. when the object is freed, its destructor function is executed.

```
class c1{
        --------
        --------
        fun(int,int)
}
int main() {
     c1 *p;
     ---------
     ---------
    p= new c1;     // parameter less constructor if any will
                   //be called

    p-> fun (23,78);
    delete p;   // destructor if any is called
}
```

```
class c1{
                --------
                --------
        public:
                c1(double n, char *s) {

                ----------
                }
                ~c1() {

                -----------
                }
int main()
c1 *p;
p= new c1 (123.7,"good work");
-----
-----
delete p;
}
```

C++'s Dynamic memory allocation operators

Array of objects

Allocation using new, we can allocate array of object but no array allocated by new can have an *initializer.*

so we should make sure that if the class contains constructor function, one will be parameter less (if you don't c++ compiler will not find a matching constructor when you attempt to allocate the array & will not compile your program)

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;
class balance {
        double cur_bal;
    char name[80];
      public:
    balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
 }
    balance() {  }   // parameterless constructor
    ~balance() {
  cout << "Destructing ";
  cout << name << "\n";
        }
void set(double n, char *s) {
   cur_bal = n;
   strcpy(name, s);
  }
  void get_bal(double &n, char *s) {
   n = cur_bal;
   strcpy(s, name);
  }
};
int main() {
  balance *p;
  char s[80];
  double n;
  int i;
  try {
   p = new balance [3];      // allocate entire array
   } catch (bad_alloc xa) {
   cout << "Allocation Failure\n";
   return 1;
    }
      // note use of dot, not arrow operators
  p[0].set(12387.87, "Ralph Wilson");
  p[1].set(144.00, "A. C. Conners");
  p[2].set(-11.23, "I. M. Overdrawn");
  for(i=0; i<3; i++) {
   p[i].get_bal(n, s);
   cout << s << "'s balance is: " << n;
   cout << "\n";
  }
  delete [] p;
  return 0;
}
```

C++'s Dynamic memory allocation operators

```
// Demonstrate nothrow version of new.
#include <iostream>
#include <new>
using namespace std;
int main() {
        int *p, i;
        p = new(nothrow) int[32]; // use nothrow option
        if(!p) {
        cout << "Allocation failure.\n";
        return 1;
        }
```

Few end notes to remember

1D dynamic array in C++

```
int* ary = new int[Size]
```

2D dynamic array in C++

```
int** ary = new int*[rowCount];

for(int i = 0; i < rowCount; ++i)
    ary[i] = new int[colCount];
```

Function overloading

int myfunc (int i);
double myfunc (double i);

→functions are overloaded beside they have different type of arguments.

int myfunc (int i)
int myfunc (int i, int j)

→overloaded as they differ in number of parameters

int myfunc (int i)
float myfunc (int i)

→ Error : differing returns types are insufficient when overloading

void f (int *p);
void f (int p[]);

→int *p & int p[] basically same so can not be overloaded

Oveloading Constructor function

→Constructor functions can also be overloaded

→Dynamically allocated Array (of objects) can not be initialized so a parameter less constructor is a must in such cases

→but if you need initialized version of objects also then you should have (with parameters) constructor also

Note : example on next slide

Constructor Overloading

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

class powers {

        int x;
public:
      // overload constructor two ways
       powers() { x = 0; }  // no initializer
                                  // parameter less constructor
      powers(int n) { x = n; } // initializer
                                     //with parameter const.
      int getx() { return x; }
      void setx(int i) { x = i; }
};

int main()
{
  powers ofTwo[] = {1, 2, 4, 8, 16};// initialized
   /* statically allocated arrays   */

  powers ofThree[5]; // uninitialized
  powers *p;
  int i;
  // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
    cout << ofTwo[i].getx() << " ";
  }
cout << "\n\n";
                        // set powers of three
  ofThree[0].setx(1);
  ofThree[1].setx(3);
  ofThree[2].setx(9);
  ofThree[3].setx(27);
  ofThree[4].setx(81);
                        // show powers of three
  cout << "Powers of three: ";
  for(i=0; i<5; i++) {
    cout << ofThree[i].getx() << " ";
  }
  cout << "\n\n";
                      // dynamically allocate an array
  try {
    p = new powers[5];      // no initialization
  } catch (bad_alloc xa) {
      cout << "Allocation Failure\n";
      return 1;
  }
                    // initialize dynamic array with powers of two
  for(i=0; i<5; i++) {
    p[i].setx(ofTwo[i].getx());
  }
                  // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
    cout << p[i].getx() << " ";
  }
  cout << "\n\n";
  delete [] p;
  return 0;
}
```

Copy constructor

→ By default when one object is used to initialize another, C++ performs a bitwise copy.

→ There are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created

→ Solution of the above problem -> Copy constructor { when copy constructor exists, the default copy is bypassed.

→class name(const class name & o){
// body of the constructor
}

→ It is important to understand that C++ defined two distinct type of situation in which the value of one object is given to another. → Assignment & initialization

Copy constructor applies only to initialization

→initialization occour in three different ways
When one object explicity initializes another, such as in a declaration

→ myclass x = y; // y explicitly initializes x

→When a copy of an objectis made to be passed to a function

func(y) // y passed as a parameter.

→When a temparary object is genareted (most commonly as a return value)???

y = func(); // y receives a temporary (returned) //object(copy constructor is called //here). However at the time of //assignment of the returned //value(object) to y, the overloaded //assignment operator if any will be //called

Copy Constructor initialization

myclass x = y; // y explicitly initializating x

func(y); // y passed as a parameter

y = func(); // temporary object generated as return //value

Assignment

```
class c1{
        -------------
}
int main (){
        c1 ob1, ob2;
        -------------
        ob1=-ob2;       //Assignment
```

Copy constructor

```
/* This program creates a "safe" array class.  Since space
   for the array is allocated using new, a copy constructor
   is provided to allocate memory when one array object is
   used to initialize another.                    */
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

class array {
                  int *p;
                  int size;
           public:
                  array(int sz) {
                  try {
                         p = new int[sz];
                  } catch (bad_alloc xa) {
                  cout << "Allocation Failure\n";
                  exit(EXIT_FAILURE);
                  }
                  size = sz;
                  }
                  ~array() { delete [] p; }
                  // copy constructor
                  array(const array &a);
                  void put(int i, int j) {
                  if(i>=0 && i<size) p[i] = j;
                  }
                  int get(int i) {
                  return p[i];
                  }
};
```

```
// Copy Constructor
array::array(const array &a) {
                  int i;
                   try {
                  p = new int[a.size];
                  } catch (bad_alloc xa) {
                  cout << "Allocation Failure\n";
                  exit(EXIT_FAILURE);
                  }
                  for(i=0; i<a.size; i++) p[i] = a.p[i];
                  }
int main()
{
  array num(10);
  int i;
  for(i=0; i<10; i++) num.put(i, i);
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "\n";
  // create another array and initialize with num
  array x(num); // invokes copy constructor
  for(i=0; i<10; i++) cout << x.get(i);
  return 0;
}
```

Copy constructor in Brief

1. array a = num; // copy constructor will be called

2. func1 (array a){ }
   main{
   func1(Num); // Copy constructor will be called while passing the arguments
   }

3. funct1{
   array Num;

   ----
   return Num;
   }

```
main() {
    array a;
    -----
    ----
    a= funct1();
}

array a (10)
array b(10);
```

b=a; // does not call copy constructor, rather it is an assignment so bitwise copy will take place & in several condition to avoid bitwise copy we need to overload the = operator

Finding the address of an overloaded function

```
#include <iostream>
using namespace std;
int myfunc(int a);
int myfunc(int a, int b);
int main() {
  int (*fp)(int a);                  // pointer to int f(int)
  fp = myfunc;                       // points to myfunc(int)
  cout << fp(5);
  return 0;
}

int myfunc(int a) {
  return a;
}

int myfunc(int a, int b) {
  return a*b;
}
```

Default function arguments

```
void myfunc (double d = 0.0)
{
}
```

→myfunc(198.234); // pass an explict value
→myfunc(); // lets function use default

→Reason & use - If a function uses lot of arguments but don't require all the arguments in all the calls then default arguments are quite a facility to use in such cases.

→When you are creating functions that have default arguments, it is important to remember that the default values must be specified only once, and this must be the first time the function is declared within the file.

→We can specify different default arguments for each version of an overloaded function

All parameters that take default values must appear to the right of those that don't

```
void iputs (int indent = -1, char *str);//error

int myfunc(float f, char *str, int i=0, int j); //Error
```

→Default parameters can also be used in constructors of an object

```
class cube {
        int x,y,z;
        public:
        cube (inti=0, int j=0, int k=0){
            ----------
        }
};
main()  {
cube a(3,3,4);
cube b; // call constructor with default arguments
```

Default arguments v/s Overloading

Ambiguity due to C++ automatic type conversion

→C++ Automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function call

```
int myfunc (double d){

}
cout<<myfunc('C'); // not an error, conversion applied
                    //(character will be converted to double
```

```
#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);
int main()  {
 cout << myfunc(10.1) << " ";// unambiguous,calls to
                            //myfunc(double) as
                            //constant fraction no. by
                            //default are double
  cout << myfunc(10); // ambiguous
  return 0;
}
float myfunc(float i)
{
  return i;
}
double myfunc(double i)
{
  return -i;
}
```

```
#include <iostream>
using namespace std;
char myfunc(unsigned char ch);
char myfunc(char ch);
int main() {
  cout << myfunc('c'); // this calls myfunc(char)
  cout << myfunc(88) << " "; // ambiguous
  return 0;
}
char myfunc(unsigned char ch) {
  return ch-1;
}
char myfunc(char ch) {
  return ch+1;
}
```

Ambiguity due to default arguments in an overloaded function

```
#include <iostream>
using namespace std;
int myfunc(int i);
int myfunc(int i, int j=1);
int main()
{
  cout << myfunc(4, 5) << " "; // unambiguous
  cout << myfunc(10); // ambiguous
  return 0;
}
int myfunc(int i) {
  return i;
}
int myfunc(int i, int j) {
  return i*j;
}
```

Some types of overloaded functions are simply inherently ambiauous

```
// This program contains an error.

#include <iostream>
using namespace std;
void f(int x);
void f(int &x); // error
int main() {
  int a=10;
  f(a); // error, which f()?
  return 0;
}
void f(int x) {
  cout << "In f(int)\n";
}
void f(int &x){
  cout << "In f(int &)\n";
}
```