Dept. of Comp. Sc. & Engg., NIT Durgapur

CSC 404 Object Oriented Programming

# Operator Overloading in C++ - III

- *subrata.nandi@cse.nitdgp.ac.in*

"Operator overloading is just syntactic sugar, which means it is simply another way for a user to make a function call" – *Bruce Eckel (Thinking in C++, Vol 1)*

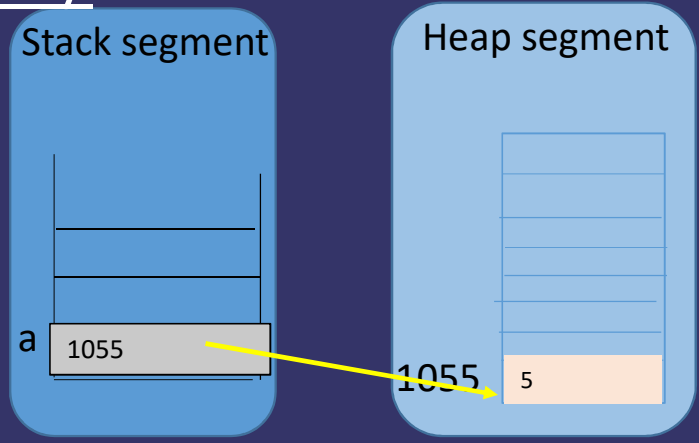
Some interesting overloading cases….

<< >> (insertion and extraction operator)
[] (subscript or array index operator)
= (assignment operator)
new
delete (memory allocation/deallocation op)

# Overloading new & delete operator

# Overloading new & delete operator – Why?

```
#include<iostream>

int main(void)
{
    int *a=new int(5);

    delete a;

    return 0;
}
```




Stack Segment hosts all local non-static var/objects

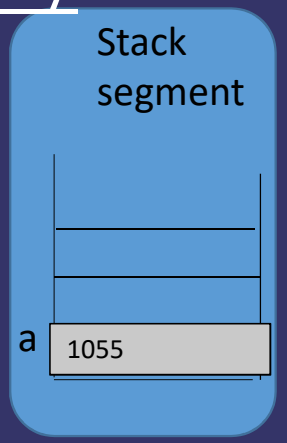
Heap Segment hosts all dynamically allocated (using new/malloc/calloc) var/objects

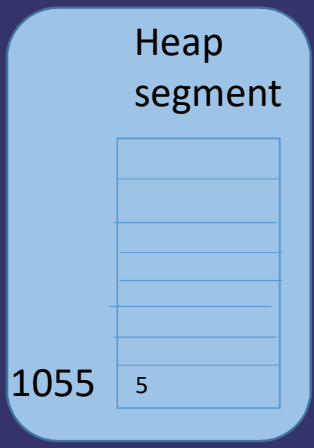
Data Segment hosts all global (static/non-static) and local static var/objects

Code Segment hosts the executable binary of all functions/methods

# Overloading new & delete operator – Why?

```
#include<iostream>

int main(void)
{
    int *a=new int(5);

    delete a;

    return 0;
}
```




After delete operation the memory becomes free;

however the contents of the location still holds the previous value;

may pose a security threat

**Can we make something extra during delete operation; some provision so that the location pointer by a is overwritten by 0 before deallocation?**

One can change the storage allocation functions operator new and operator delete (by overloading them), if requires

# Overloading new & delete operator – When?

a) Dynamic memory allocation calling overloaded operator new
b) Constructor gets executed

- A new-expression ( complex *p=new complex)

- A delete-expression (delete p), when executed two things occur:
  - First the destructor is called, secondly, object storage is deallocated from heap using the operator delete

- Possible cases when overloading new/delete is required:
  - You might write a custom operator delete that overwrites deallocated memory with zeros in order to increase the security of application data
  - Dealing with heap fragmentation (allocating objects of different sizes may possibly to breaks up the heap so that you although storage might be available, but because of fragmentation no piece is big enough to satisfy your need
  - Embedded and real-time systems (program runs for very long time) require that memory allocation always take the same amount of time, and there's no allowance for heap exhaustion or fragmentation
  - Exception handling routine can be added in overloaded new operator function when space allocation fails
  - Creating and destroying too many node objects of a particular linked-list class may become a speed bottleneck; we may have custom node allocation scheme that uses avail list

- The new/delete operator may be overloaded globally / for a class

# Overloading Global new & delete operator

- If we overload the global versions, the defaults becomes completely inaccessible for all classes (inbuilt/user-defined); calling the defaults inside the redefinitions is also impossible – alternatively malloc/calloc can be used

- Syntax for overloading the new operator: void* operator new(size_t)

  - The input argument size_t is generated and passed by the compiler and is the size of the object to be allocated
  - The return value is a void* (if allocation is successful) i.e. a pointer to any particular type; as space allocation is completed and object construction is yet to happen;
  - (if allocation is unsuccessful) throw an exception to signal that there was a alloc problem; or do some thing else

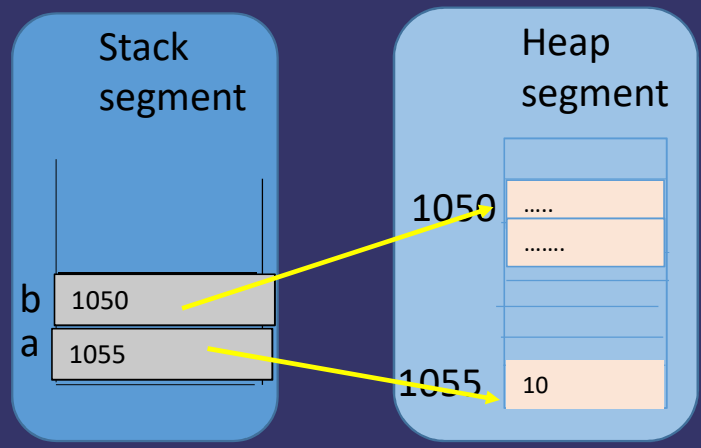
- Syntax for overloading the delete operator void delete(void *)

  - Input argument is void* to memory that was allocated by operator new; as it's called after the destructor destroys the object-ness of the space

# Overloading Global new & delete operator

```
#include<iostream>
#include<cstdio>
#include<cstdlib>

// global new delete operator behaviours
void  *operator new(size_t sz)
{
   printf("\n\tGlobal Overloaded new; size: %d bytes\n",(int)sz);

  void *x;
  x=malloc(sz);

  if(!x)
     printf("\n\t Out of memory");   // may throw an exception here

    return x;
}

void operator delete(void *x)
{
   printf("\n\tGlobal Overloaded delete");

  if(!x)
    free(x);
}
```

```
int main(void)
{
     int *a=new int(10);

     int   *b=new int[2];


     delete a;
     delete b;

     return 0;
}
```




Output:

Global Overloaded new; size: 4 bytes

Global Overloaded new; size: 20 bytes

Global Overloaded delete
Global Overloaded delete

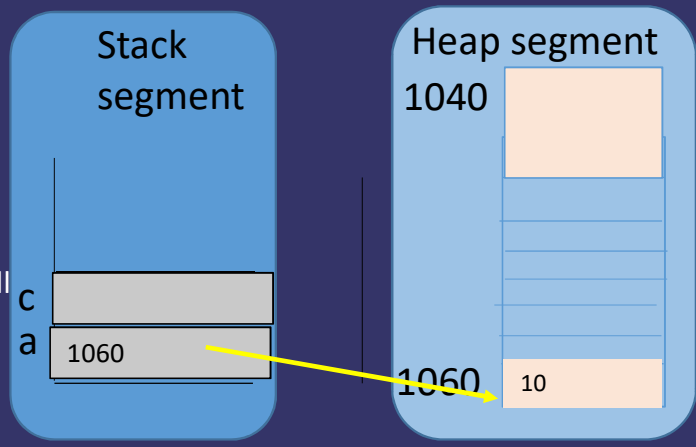
Note:

- Default new/delete becomes inaccessible for built-in data types
- Value of sz is estimated and passed by compiler
- cout can't be used as this iostream object too requires new to allocate memory
- Works for both single object or array of objects

# Overloading Global new & delete operator

```
#include<iostream>
#include<cstdio>
#include<cstdlib>

// global new delete operator behaviours
void  *operator new(size_t sz)
{
   printf("\n\tGlobal Overloaded new; size: %d bytes\n",(int)sz);

  void *x;
  x=malloc(sz);

  if(!x)
     printf("\n\t Out of memory");    // may throw an exception here

    return x;
}

void operator delete(void *x)
{
    printf("\n\tGlobal Overloaded delete");

  if(!x)
    free(x);
}
```

```
class complex
{
  float rl, img;
 public:
   complex(float f1=1.0,float f2=1.0)
  {
      cout<<"Complex Object Constructor"<<endl
      rl=f1; img=f2;
  }

~complex()
 {
     cout<<"Complex Object Destructor"<<endl;
 }
 // other methods
};  // End of class complex

  int main(void)
  {
       int *a=new int(10);
       delete a;

       complex *c=new complex;
       complex *d=new complex[2];


       delete  c;
       delete  [] d;

       return 0;
```



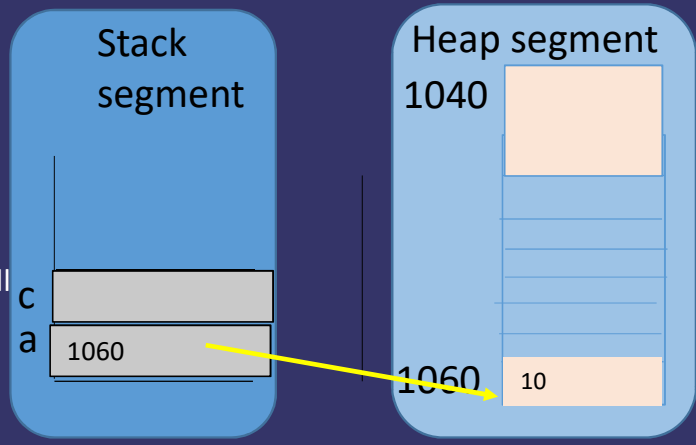
# Overloading Global new & delete operator

```
#include<iostream>
#include<cstdio>
#include<cstdlib>

// global new delete operator behaviours
void  *operator new(size_t sz)
{
  printf("\n\tGlobal Overloaded new; size: %d bytes\n",(int)sz);

  void *x;
  x=malloc(sz);

  if(!x)
    printf("\n\t Out of memory");   // may throw an exception here

   return x;
}

voi
{
  p

  if(
   fr
}
```

Note:

- Overloaded new/delete works for User-defined type too
- Works for both single object and array of complex objects
- When created the overloaded new is called first followed by constructor
- When destroyed destructor called followed by overloaded delete
- In array of objects the number of bytes sz requested that extra memory to store information (inside the array) about the number of objects it holds

```
class complex
{
  float rl, img;
 public:
  complex(float f1=1.0,float f2=1.0)
  {
     cout<<"Complex Object Constructor"<<endl
     rl=f1; img=f2;
  }

~complex()
 {
    cout<<"Complex Object Destructor"<<endl;
 }
 // other methods
};  // End of class complex

  int main(void)
  {
      int *a=new int(10);
      delete a;

    complex *c=new complex;
    complex *d=new complex[2];


    delete  c;
    delete  [] d;

    return 0;
```



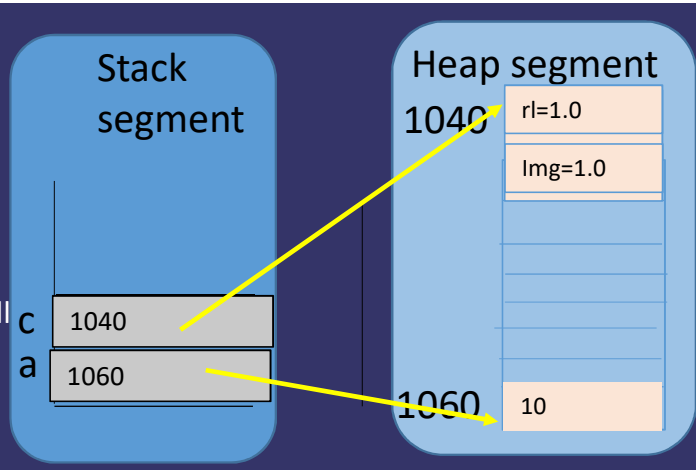
# Overloading Global new & delete operator

```
#include<iostream>
#include<cstdio>
#include<cstdlib>

// global new delete operator behaviours
void  *operator new(size_t sz)
{
  printf("\n\tGlobal Overloaded new; size: %d bytes\n",(int)sz);

  void *x;
  x=malloc(sz);

  if(!x)
    printf("\n\t Out of memory");   // may throw an exception here

   return x;
}

voi
{
  p

  if(
   fr
}
```

Note:

- Overloaded new/delete works for User-defined type too
- Works for both single object and array of complex objects
- When created the overloaded new is called first followed by constructor
- When destroyed destructor called followed by overloaded delete
- In array of objects the number of bytes sz requested that extra memory to store information (inside the array) about the number of objects it holds

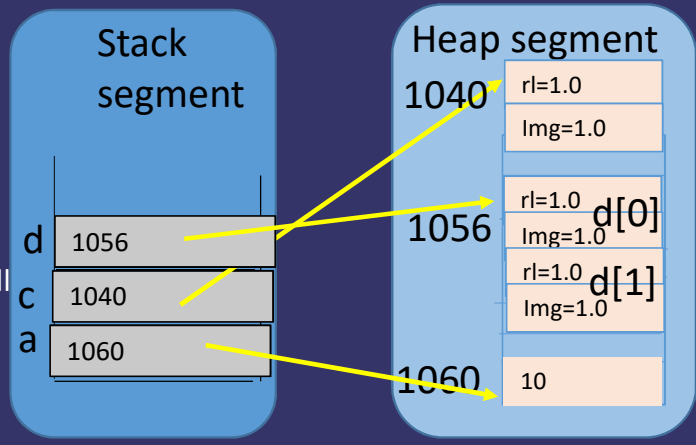
```
class complex
{
  float rl, img;
 public:
   complex(float f1=1.0,float f2=1.0)
  {
     cout<<"Complex Object Constructor"<<endl;
     rl=f1; img=f2;
  }

~complex()
 {
    cout<<"Complex Object Destructor"<<endl;
 }
 // other methods
};  // End of class complex
  int main(void)
  {
      int *a=new int(10);
       delete a;

    complex *c=new complex;
    complex *d=new complex[2];

    delete  c;
    delete  [] d;

    return 0;
```



# Overloading Global new & delete operator

```
#include<iostream>
#include<cstdio>
#include<cstdlib>

// global new delete operator behaviours
void  *operator new(size_t sz)
{
  printf("\n\tGlobal Overloaded new; size: %d bytes\n",(int)sz);

  void *x;
  x=malloc(sz);

  if(!x)
    printf("\n\t Out of memory");   // may throw an exception here

   return x;
}

void
{
  p

  if(
   fr
}
```

```
class complex
{
  float rl, img;
 public:
   complex(float f1=1.0,float f2=1.0)
   {
      cout<<"Complex Object Constructor"<<endl
      rl=f1; img=f2;
   }

~complex()
 {
    cout<<"Complex Object Destructor"<<endl;
 }
 // other methods
};  // End of class complex

int main(void)
{
    int *a=new int(10);
    delete a;

   complex *c=new complex;
   delete c;

   complex *d=new complex[2];
   delete  [] d;

   return 0;
```




Output:

```
Global Overloaded new; size: 4 bytes
      Global Overloaded delete

      Global Overloaded new; size: 8 bytes
Complex Object Constructor
Complex Object Destructor
      Global Overloaded delete

      Global Overloaded new; size: 24 bytes
Complex Object Constructor
Complex Object Constructor
Complex Object Destructor
Complex Object Destructor
      Global Overloaded delete
```

Note:

- Overloaded new/delete works for User-defined type too
- Works for both single object and array of complex objects
- When created the overloaded new is called first followed by constructor
- When destroyed destructor called followed by overloaded delete
- In array of objects the number of bytes sz requested that extra memory to store information (inside the array) about the number of objects it holds

# Overloading new & delete operator for a class

- Overload new and delete operators for a class, are treated as static members of the class; although explicit mentioned of static is not necessary

- The compiler chooses the member operator new (if available) over the global version. However, the global versions of new and delete are used for all other types of objects (unless they have their own new and delete)

# Overloading new & delete operator for a class

```
#include<iostream>
#include<cstdio>
#include<cstdlib>

class complex
{
  float rl, img;
 public:
  complex(float f1=1.0,float f2=1.0)
  {
     cout<<"Complex Object Constructor"<<endl;
     rl=f1; img=f2;
  }

~complex()
 {
    cout<<"Complex Object Destructor"<<endl;
 }
//  new delete operator member functions
void  *operator new(size_t sz)
{
  printf("\n\tGlobal Overloaded new; size: %d bytes\n",(int)sz);

  void *x;
  x=malloc(sz);

  if(!x)    printf("\n\t Out of memory");   // may throw an exception here

   return x;
}

void operator delete(void *x)
{
   printf("\n\tGlobal Overloaded delete");

  if(!x)   free(x);
}

};   // End of class complex
```

```
int main(void)
{
   int *a=new int; //calls default
  delete a;

  complex *c=new complex;
  delete c;

  complex *d=new complex[2]; //calls default
  delete []d;
}
```

Output:

```
      Complex Overloaded new; size: 8 bytes
Complex Object Constructor
Complex Object Destructor
      Complex Overloaded delete

Complex Object Constructor
Complex Object Constructor
Complex Object Destructor
Complex Object Destructor
```

Note:

- user-defined data type for single object allocation use overloaded new/delete
- Built-in types and user-defined data type for arrays use default new/delete
- We can use ::operator new(sz) instead of malloc and ::operator delete(x) instead of free
- cout can be used instead of printf
- Mention of keyword static for new/delete operator function is optional

# Overloading new[] & delete[] operator for a class

```
#include<iostream>
#include<cstdio>
#include<cstdlib>

class complex
{
  float rl, img;
 public:
   complex(float f1=1.0,float f2=1.0)
  {
     cout<<"Complex Object Constructor"<<endl;
     rl=f1; img=f2;
  }

~complex()
 {
    cout<<"Complex Object Destructor"<<endl;
 }
//  new delete operator member functions
void  *operator new(size_t sz)
{
  printf("\n\tGlobal Overloaded new; size: %d bytes\n",(int)sz);

  void *x;
  x=malloc(sz);

  if(!x)    printf("\n\t Out of memory");   // may throw an exception here

   return x;
}

void operato
{
   printf("\n\

  if(!x)   free(x);
}
```

Note:
- overloaded new/delete for single object allocation and arrays of user-defined objects
- Built-in types only use the default new/delete

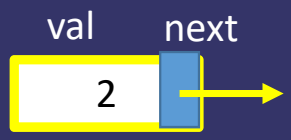
```
//for dy allocating an array of object
void  *operator new[](size_t sz)
{
     printf("\n\tComplex Overloaded new for array; size: %d bytes\n",(int)sz);

     void *x=malloc(sz);
       if(!x)     printf("\n\t Out of memory");
   return x;
}

//for deallocating an array of dy allocated objects
void operator delete[](void *x)
{
    printf("\n\tComplex Overloaded delete for arrays\n");
    if(!x)    free(x);
}

}; // End of class complex

int main(void)
{
int *a=new int;
  delete a;

  complex *c=new complex;
  delete c;

  complex *d=new complex[2];
  delete []d;
}
```

Output:

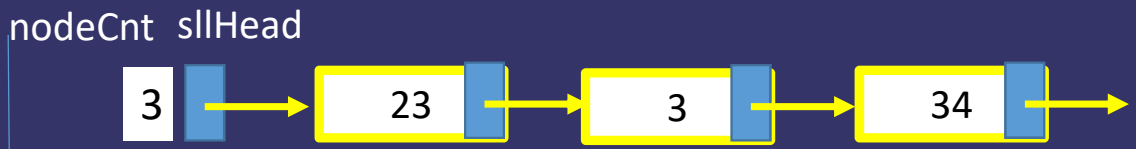
```
      Complex Overloaded new; size: 8 bytes
Complex Object Constructor
Complex Object Destructor
      Complex Overloaded delete

      Complex Overloaded new for array; size: 24 bytes
Complex Object Constructor
Complex Object Constructor
Complex Object Destructor
Complex Object Destructor
      Complex Overloaded delete for arrays
```

# Example – Singly linear linked list

# Example – Singly linear linked list

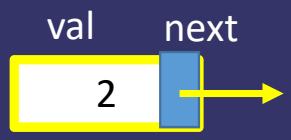



**sllnode object**
**(A node with single link )**



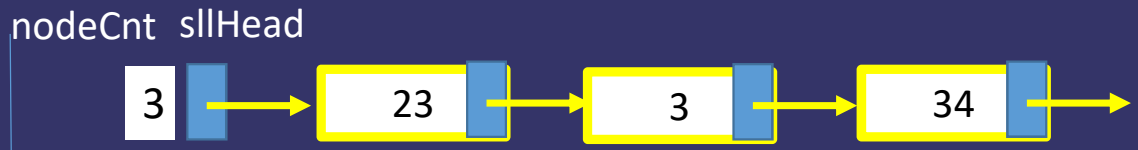

**sll object**
**(A singly linear linked list is composed of a linked collection of zero/few sllnode objects)**

# Example – Singly linear linked list



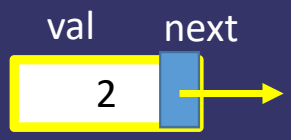

**sllnode object**
**(A node with single link )**

```
class   sllnode
{   public:
     int    val;
     sllnode  *next;

     sllnode(int  x=-1, sllnode *p=NULL);

     ~sllnode();

};
```



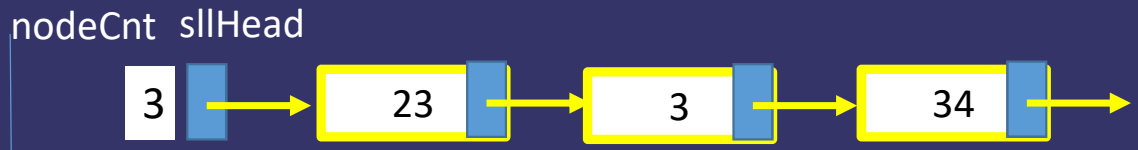

**sll object**
**(A singly linear linked list is composed of a linked collection of zero/few sllnode objects)**

# Example – Singly linear linked list




**sllnode object**
**(A node with single link )**

```
class   sllnode
{   public:
     int    val;
     sllnode  *next;

     sllnode(int  x=-1, sllnode *p=NULL);

     ~sllnode();

};
```




**sll object**
**(A singly linear linked list is composed of a linked collection of zero/few sllnode objects)**


sll object in its minimal form; (sll class definition should specify this form)

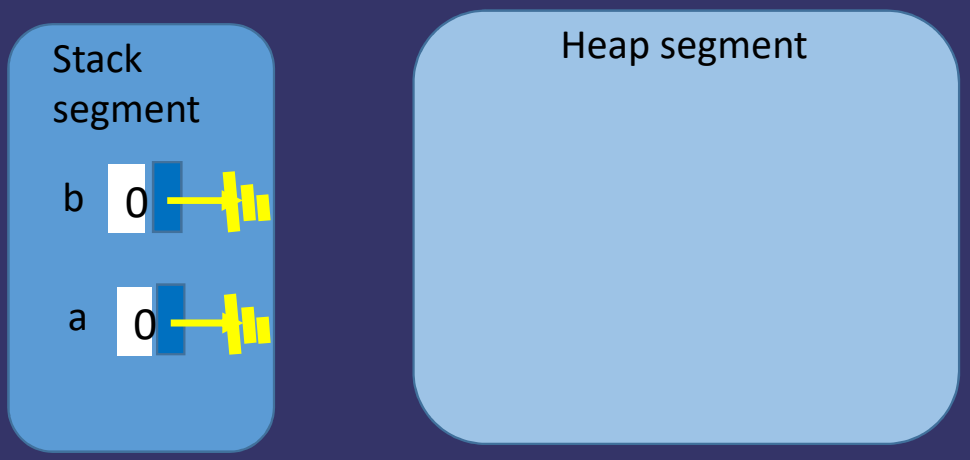
```
class sll
{
     sllnode *sllHead;
     public:     int  nodeCnt;

     sll(sllnode *p=NULL);
     ~sllnode();
     friend int  operator+(int x, sll  &list); // 3 + a
     friend int operator-(sll  &list); // - a
     friend ostream & operator<<(ostream &x, const sll &list); // cout<<a<<b
}
```

## Singly linear linked list.....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
}; // End of class sllnode definition
```

```
class sll
{
     sllnode *sllHead;
     public:     int  nodeCnt;

     sll(sllnode *p=NULL);
     ~sllnode();
     sll(const sll &x);
     friend int  operator+(int x, sll  &list); // 3 + a
     friend int operator-(sll  &list); // - a
     friend ostream & operator<<(ostream &x, const sll &list);
     sll & operator=(const sll &list);  //  a = b
    int operator[](int x);     //  a[3]
} ;   // End of class sll definitionv
```

```
int operator+(int x, sll  &list) // add int in the beginning
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)   // delete node from beginning
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
 {
    sll   a, b;
          3+a;
          4+a;
          -a;
          5+b;
          7+a;
 }
```

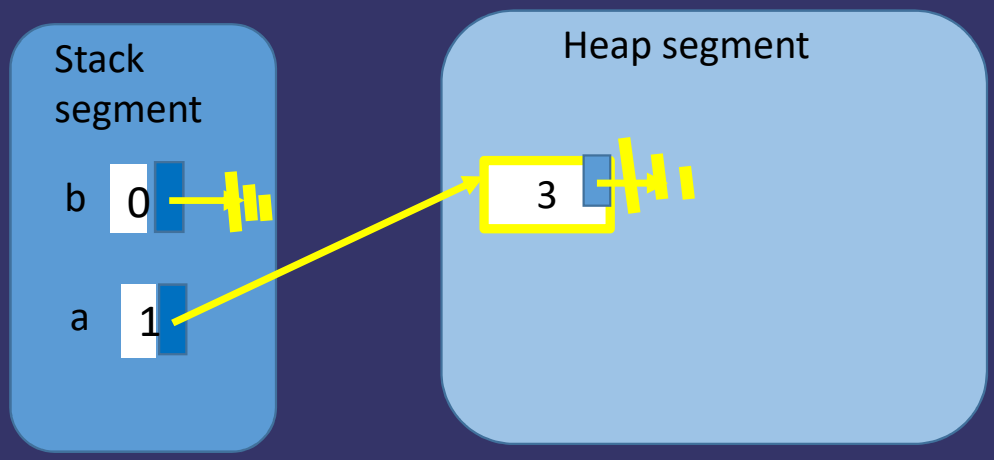


## Singly linear linked list.....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
}; // End of class sllnode definition
```

```
class sll
{
     sllnode *sllHead;
     public:     int  nodeCnt;

     sll(sllnode *p=NULL);
     ~sllnode();
     sll(const sll &x);
     friend int  operator+(int x, sll  &list); // 3 + a
     friend int operator-(sll  &list); // - a
     friend ostream & operator<<(ostream &x, const sll &list);
     sll & operator=(const sll &list);  //  a = b
    int operator[](int x);     //  a[3]
} ;   // End of class sll definitionv
```

```
int operator+(int x, sll  &list) // add int in the beginning
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)   // delete node from beginning
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
 {
         sll   a, b;
         3+a;
         4+a;
         -a;
         5+b;
         7+a;
 }
```

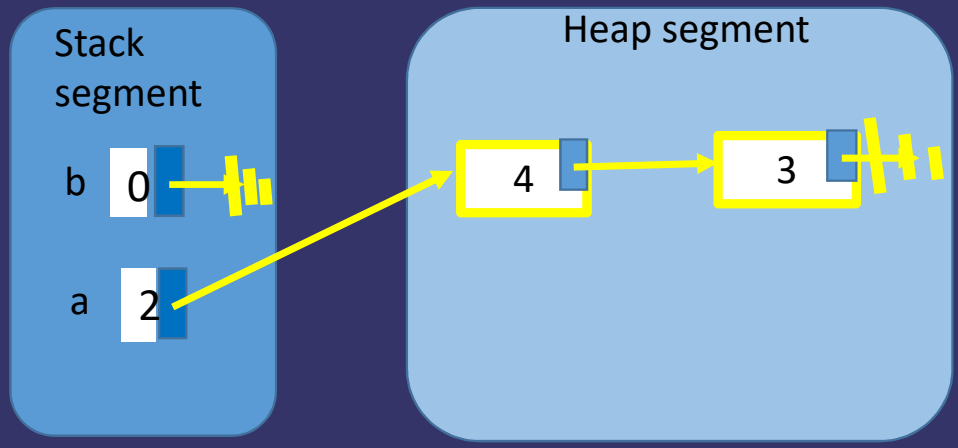


## Singly linear linked list.....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
}; // End of class sllnode definition
```

```
class sll
{
     sllnode *sllHead;
     public:     int  nodeCnt;

     sll(sllnode *p=NULL);
     ~sllnode();
     sll(const sll &x);
     friend int  operator+(int x, sll  &list); // 3 + a
     friend int operator-(sll  &list); // - a
     friend ostream & operator<<(ostream &x, const sll &list);
     sll & operator=(const sll &list);  //  a = b
    int operator[](int x);     //  a[3]
} ;   // End of class sll definitionv
```

```
int operator+(int x, sll  &list) // add int in the beginning
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)   // delete node from beginning
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
 {
         sll   a, b;
          3+a;
          4+a;
          -a;
          5+b;
          7+a;
 }
```

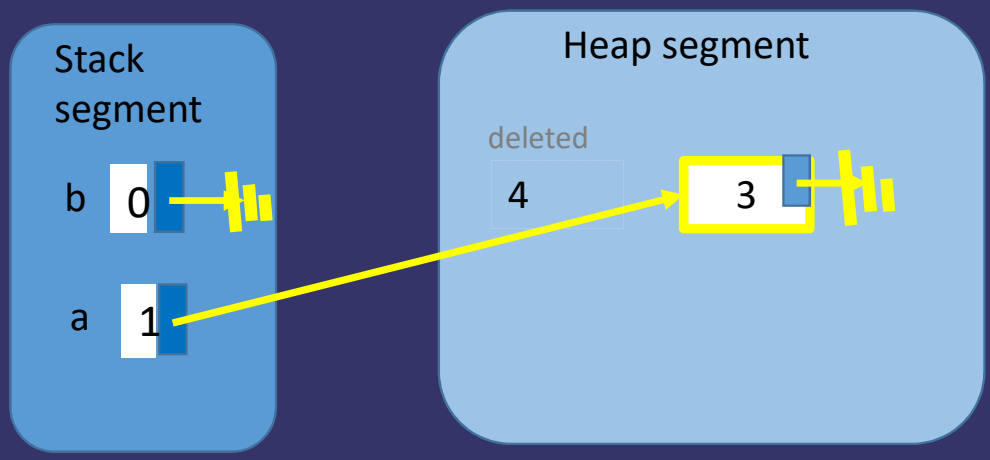


## Singly linear linked list.....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
}; // End of class sllnode definition
```

```
class sll
{
     sllnode *sllHead;
     public:     int  nodeCnt;

     sll(sllnode *p=NULL);
     ~sllnode();
     sll(const sll &x);
     friend int  operator+(int x, sll  &list); // 3 + a
     friend int operator-(sll  &list); // - a
     friend ostream & operator<<(ostream &x, const sll &list);
     sll & operator=(const sll &list);  //  a = b
    int operator[](int x);     //  a[3]
} ;   // End of class sll definitionv
```

```
int operator+(int x, sll  &list) // add int in the beginning
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)   // delete node from beginning
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
 {
         sll   a, b;
          3+a;
          4+a;
          -a;
          5+b;
          7+a;
 }
```

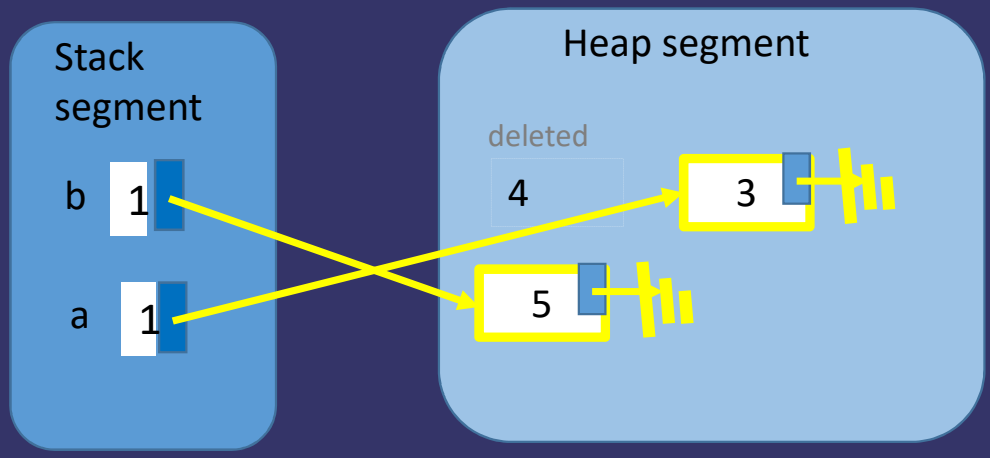


## Singly linear linked list.....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
}; // End of class sllnode definition
```

```
class sll
{
     sllnode *sllHead;
     public:     int  nodeCnt;

     sll(sllnode *p=NULL);
     ~sllnode();
     sll(const sll &x);
     friend int  operator+(int x, sll  &list); // 3 + a
     friend int operator-(sll  &list); // - a
     friend ostream & operator<<(ostream &x, const sll &list);
     sll & operator=(const sll &list);  //  a = b
    int operator[](int x);     //  a[3]
} ;   // End of class sll definitionv
```

```
int operator+(int x, sll  &list) // add int in the beginning
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)   // delete node from beginning
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
 {
          sll   a, b;
           3+a;
           4+a;
           -a;
           5+b;
           7+a;
 }
```

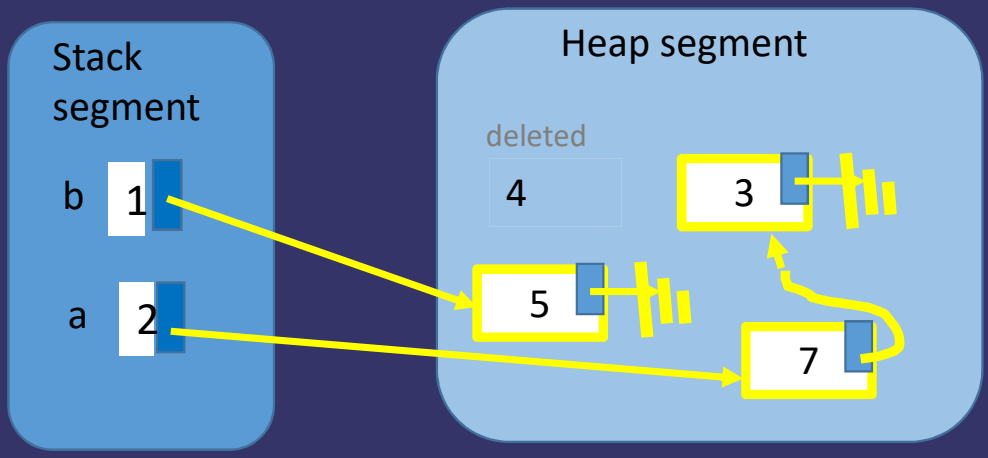


## Singly linear linked list.....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
}; // End of class sllnode definition
```

```
class sll
{
     sllnode *sllHead;
     public:     int  nodeCnt;

     sll(sllnode *p=NULL);
     ~sllnode();
     sll(const sll &x);
     friend int  operator+(int x, sll  &list); // 3 + a
     friend int operator-(sll  &list); // - a
     friend ostream & operator<<(ostream &x, const sll &list);
     sll & operator=(const sll &list);  //  a = b
    int operator[](int x);     //  a[3]
} ;   // End of class sll definitionv
```

```
int operator+(int x, sll  &list) // add int in the beginning
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)   // delete node from beginning
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
 {
          sll   a, b;
           3+a;
           4+a;
           -a;
           5+b;
           7+a;
 }
```




- Every time new sllnode is called the node space is requested from heap via Operating System (OS) memory manager module (time consuming)
- Could have been done in a faster way using avail list which saves (allocation/dealocation) time

# Singly linear linked list (with overloaded new/delete using avail list)

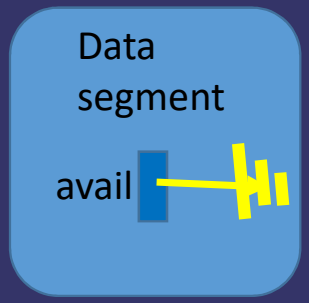
**Solution:**

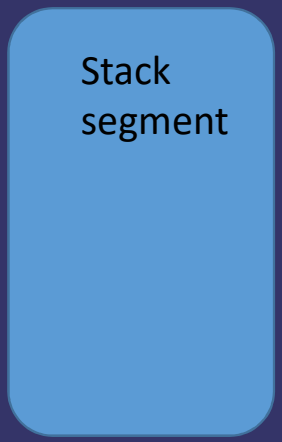
- Maintain a avail list (list of deleted nodes)
- Where & how? In sllnode class create a static pointer to node member avail; declare and initialize it globally outside the class
- Overload new (in sllnode) such that it if avail list is non-empty gets a node from avail list instead of calling OS else call OS
- Overload delete (in sllnode) such that it inserts the node to be deleted in the beginning of avail list instead of calling OS to deallocate it

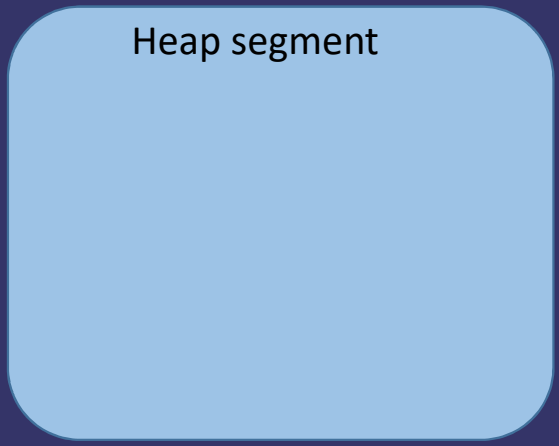
## Singly linear linked list (using avail list).....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     static  sllnode  *avail;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
void *operator new(size_t sz)
{   void *p;
     if(sllnode::avail==NULL)  // if empty avail list
        p=::operator new(sz);  /// get from heap
     else  // get from avail
     {   p=avail; sllnode::avail=sllnode::avail->next;
      }
    return p;
}

void operator delete(void *p)
{
   sllnode *t=(sllnode*)p;
   t->next=sllnode::avail;
   t->val=0;  //  reset node contents
   sllnode::avail=t;  // add the node to avail list
   return;
}
}; // End of class sllnode definition
sllnode * sllnode::avail=NULL;
class sll
{.....};
```

```
int operator+(int x, sll  &list)
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
 {
         sll   a, b;
          3+a;
          4+a;
          -a;
          5+b;
          7+a;
 }
```



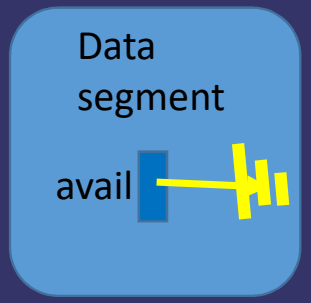



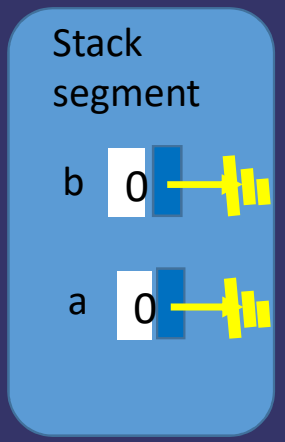



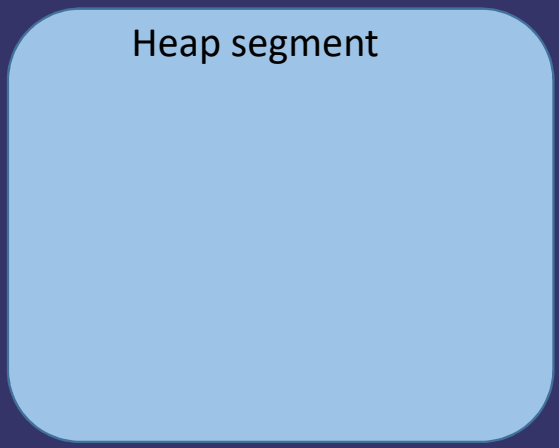
## Singly linear linked list (using avail list).....

```
class  sllnode
{  public:
    int   val;   sllnode  *next;
    static  sllnode  *avail;
    sllnode(int  x=-1, sllnode *p=NULL);
    ~sllnode();
void *operator new(size_t sz)
{  void *p;
    if(sllnode::avail==NULL)  // if empty avail list
       p=::operator new(sz);  /// get from heap
    else  // get from avail
    {  p=avail; sllnode::avail=sllnode::avail->next;
     }
   return p;
}

void operator delete(void *p)
{
   sllnode *t=(sllnode*)p;
   t->next=sllnode::avail;
   t->val=0;  //  reset node contents
   sllnode::avail=t;  // add the node to avail list
   return;
}
}; // End of class sllnode definition
sllnode  * sllnode::avail=NULL;
class sll
{.....};
```

```
int operator+(int x, sll  &list)
{   cout<<"opeartor binary + overload "<<endl;
    list.sllHead=new sllnode(x, list.sllHead);
    list.nodeCnt++;  return 1;
}
int operator-(sll  &list)
{  cout<<"opeartor unary - overload "<<endl;
   sllnode  *temp=list.sllHead;
   int  info=temp->val;
   list.sllHead=list.sllHead->next;
   list.nodeCnt--;
   delete temp;
   return info;
}
 main(void)
{
        sll   a, b;
         3+a;
         4+a;
         -a;
         5+b;
         7+a;
}
```



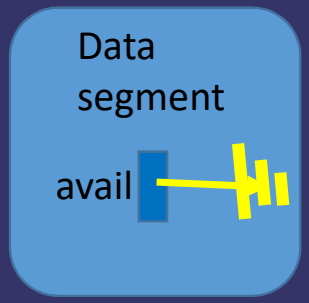



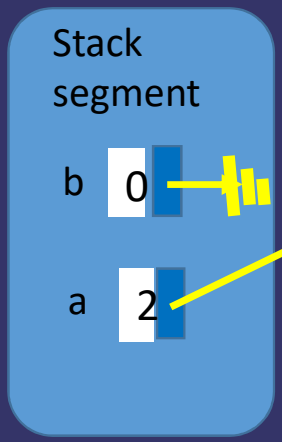



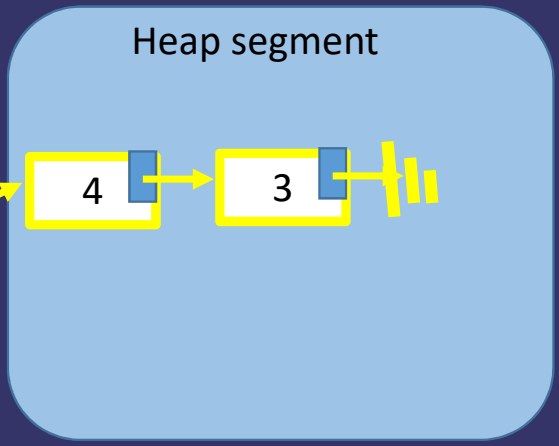
## Singly linear linked list (using avail list).....

```
class  sllnode
{  public:
    int   val;   sllnode  *next;
    static  sllnode  *avail;
    sllnode(int  x=-1, sllnode *p=NULL);
    ~sllnode();
void *operator new(size_t sz)
{  void *p;
    if(sllnode::avail==NULL)  // if empty avail list
     p=::operator new(sz);  //get from heap
    else  // get from avail
    {  p=avail; sllnode::avail=sllnode::avail->next;
     }
   return p;
}

void operator delete(void *p)
{
  sllnode *t=(sllnode*)p;
  t->next=sllnode::avail;
  t->val=0;  //  reset node contents
  sllnode::avail=t;  // add the node to avail list
  return;
}
}; // End of class sllnode definition
sllnode  * sllnode::avail=NULL;
class sll
{.....};
```

```
int operator+(int x, sll  &list)
{   cout<<"opeartor binary + overload "<<endl;
    list.sllHead=new sllnode(x, list.sllHead);
    list.nodeCnt++;  return 1;
}
int operator-(sll  &list)
{ cout<<"opeartor unary - overload "<<endl;
  sllnode  *temp=list.sllHead;
  int  info=temp->val;
  list.sllHead=list.sllHead->next;
  list.nodeCnt--;
  delete temp;
  return info;
}
main(void)
{
       sll   a, b;
        3+a;
        4+a;
        -a;
        5+b;
        7+a;
}
```



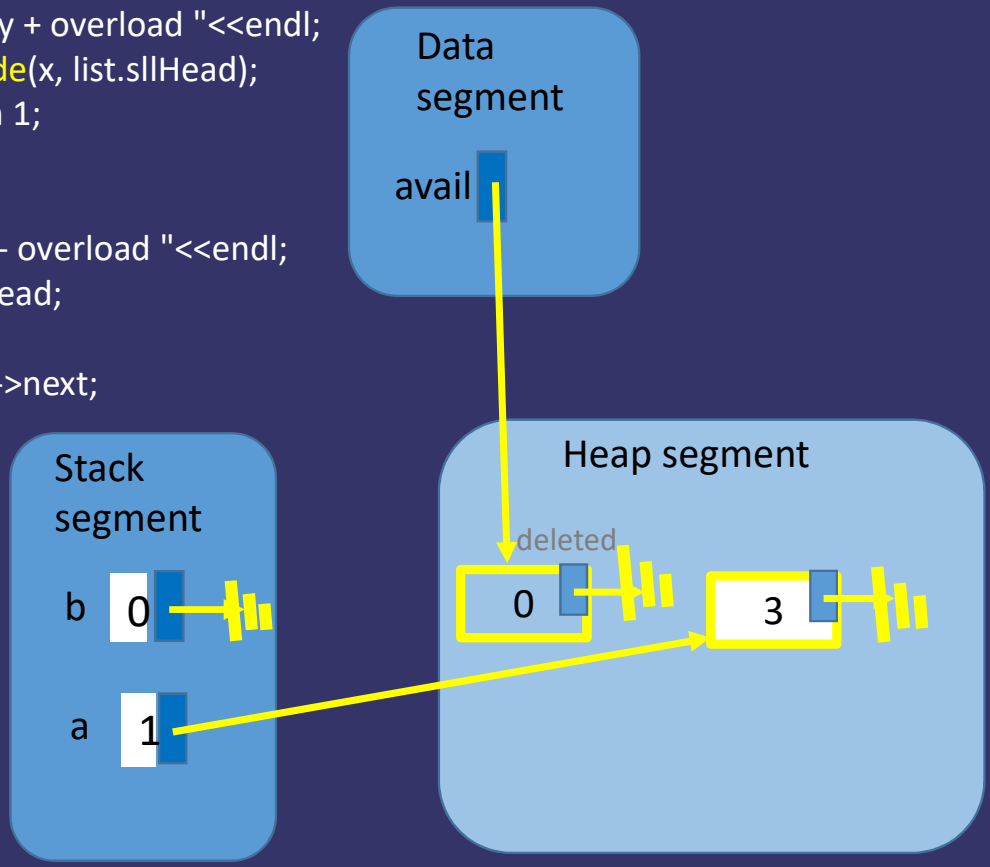



## Singly linear linked list (using avail list).....

```
class  sllnode
{  public:
    int   val;   sllnode  *next;
    static  sllnode  *avail;
    sllnode(int  x=-1, sllnode *p=NULL);
    ~sllnode();
void *operator new(size_t sz)
{  void *p;
    if(sllnode::avail==NULL)  // if empty avail list
       p=::operator new(sz);  /// get from heap
    else  // get from avail
    {  p=avail; sllnode::avail=sllnode::avail->next;
     }
   return p;
}

void operator delete(void *p)
{
  sllnode *t=(sllnode*)p;
  t->next=sllnode::avail;
  t->val=0;  //  reset node contents
  sllnode::avail=t;  // add the node to avail list
  return;
}
}; // End of class sllnode definition
sllnode  * sllnode::avail=NULL;
class sll
{.....};
```

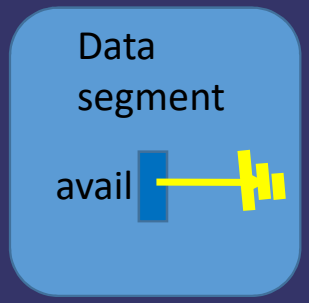
```
int operator+(int x, sll  &list)
{   cout<<"opeartor binary + overload "<<endl;
    list.sllHead=new sllnode(x, list.sllHead);
    list.nodeCnt++;  return 1;
}
int operator-(sll  &list)
{  cout<<"opeartor unary - overload "<<endl;
   sllnode  *temp=list.sllHead;
   int  info=temp->val;
   list.sllHead=list.sllHead->next;
   list.nodeCnt--;
   delete temp;
   return info;
}
main(void)
{
        sll   a, b;
         3+a;
         4+a;
         -a;
         5+b;
         7+a;
}
```

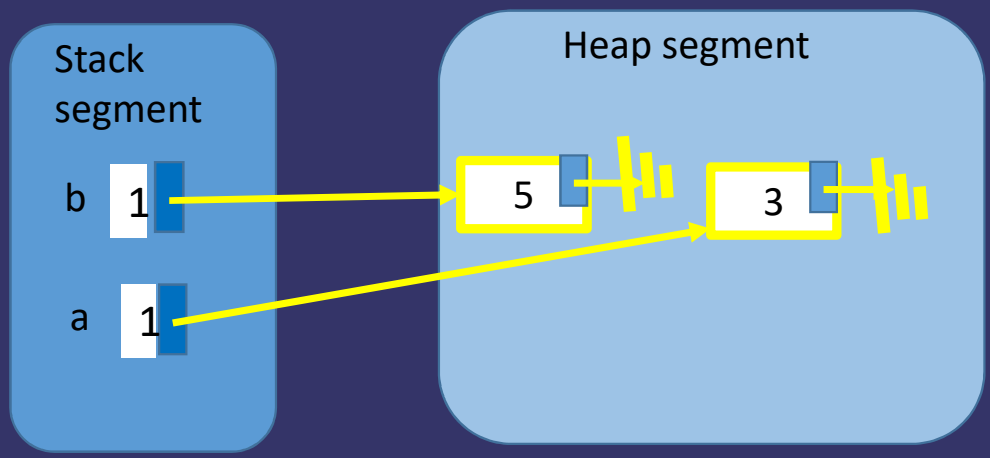


## Singly linear linked list (using avail list).....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     static  sllnode  *avail;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
void *operator new(size_t sz)
{   void *p;
     if(sllnode::avail==NULL)  // if empty avail list
        p=::operator new(sz);  /// get from heap
     else  // get from avail
     {  p=avail; sllnode::avail=sllnode::avail->next;
     }
     return p;
}

void operator delete(void *p)
{
   sllnode *t=(sllnode*)p;
   t->next=sllnode::avail;
   t->val=0;  //  reset node contents
   sllnode::avail=t;  // add the node to avail list
   return;
}
}; // End of class sllnode definition
sllnode  * sllnode::avail=NULL;
class sll
{.....};
```

```
int operator+(int x, sll  &list)
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
 {
         sll   a, b;
          3+a;
          4+a;
          -a;
          5+b;
          7+a;
 }
```

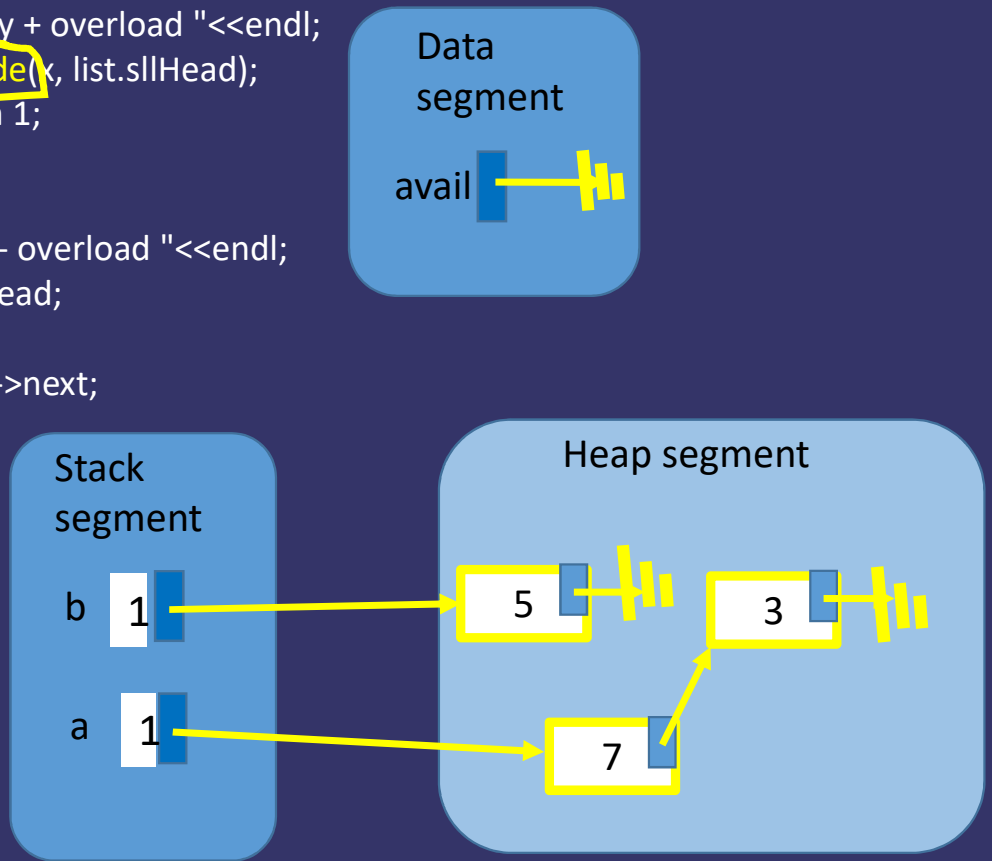


## Singly linear linked list (using avail list).....

```
class  sllnode
{  public:
    int   val;   sllnode  *next;
    static  sllnode  *avail;
    sllnode(int  x=-1, sllnode *p=NULL);
    ~sllnode();
void *operator new(size_t sz)
{  void *p;
    if(sllnode::avail==NULL)  // if empty avail list
      p=::operator new(sz);   // get from heap
    else  // get from avail
    {  p=avail; sllnode::avail=sllnode::avail->next;
     }
   return p;
}

void operator delete(void *p)
{
   sllnode *t=(sllnode*)p;
   t->next=sllnode::avail;
   t->val=0;  //  reset node contents
   sllnode::avail=t;  // add the node to avail list
   return;
}
}; // End of class sllnode definition
sllnode  * sllnode::avail=NULL;
class sll
{.....};
```

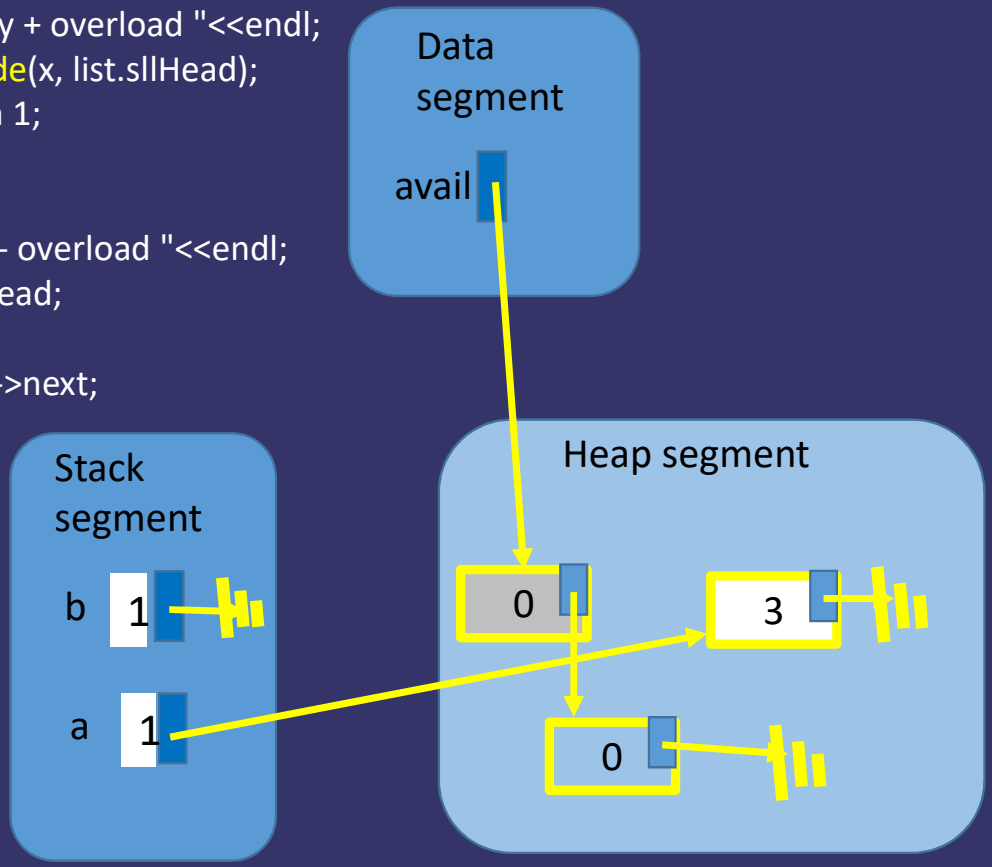
```
int operator+(int x, sll  &list)
{   cout<<"opeartor binary + overload "<<endl;
    list.sllHead=new sllnode(x, list.sllHead);
    list.nodeCnt++;  return 1;
}
int operator-(sll  &list)
{  cout<<"opeartor unary - overload "<<endl;
   sllnode  *temp=list.sllHead;
   int  info=temp->val;
   list.sllHead=list.sllHead->next;
   list.nodeCnt--;
   delete temp;
   return info;
}
 main(void)
{
        sll   a, b;
         3+a;
         4+a;
         -a;
         5+b;
         7+a;
}
```



## Singly linear linked list (using avail list).....

```
class   sllnode
{   public:
     int    val;    sllnode  *next;
     static  sllnode  *avail;
     sllnode(int  x=-1, sllnode *p=NULL);
     ~sllnode();
void *operator new(size_t sz)
{   void *p;
     if(sllnode::avail==NULL)  // if empty avail list
        p=::operator new(sz);   // get from heap
     else  // get from avail
     {   p=avail; sllnode::avail=sllnode::avail->next;
      }
    return p;
}

void operator delete(void *p)
{
   sllnode *t=(sllnode*)p;
   t->next=sllnode::avail;
   t->val=0;  //  reset node contents
   sllnode::avail=t;  // add the node to avail list
   return;
}
}; // End of class sllnode definition
sllnode  * sllnode::avail=NULL;
class sll
{.....};
```

```
int operator+(int x, sll  &list)
{    cout<<"opeartor binary + overload "<<endl;
      list.sllHead=new sllnode(x, list.sllHead);
      list.nodeCnt++;  return 1;
}
int operator-(sll  &list)
{  cout<<"opeartor unary - overload "<<endl;
    sllnode  *temp=list.sllHead;
    int  info=temp->val;
    list.sllHead=list.sllHead->next;
    list.nodeCnt--;
    delete temp;
    return info;
}
 main(void)
{
        sll   a, b;
         3+a;
         4+a;
         -a;
         5+b;
         7+a;
         -a;
         -b;
}
```



# Assignments (Operator Overloading)….

- Complex class
  Add (+), Subtract (-), Multiply (*), Divide (/), Conjugate (!), Compare (==, !=), Copy (=),
  Subscript ([ ]) – returns real for [0] and img for [1]; Input-Output (>>,<<)

- Fraction class
  Add (+), Subtract (-), Multiply (*), Divide (/), Normalize ( unary *), Compare (==, !=, <, >), Copy (=),
  Subscript ([ ]) – returns numerator for [0] and denominator for [1]; Input-Output (>>,<<)

- Matrix class (Don't forget to add copy constructor)
  Add (+), Subtract (-), Multiply (*), Divide (/), Invert ( ! ), Compare (==), **Copy (=), Subscript ([ ]) – check and display message for out of bound access,** Allocation/Deallocation (new, delete), Input-Output (>>,<<)

- Set class Don't forget to add copy constructor)
  Union (+, Difference (-), Intersection (*), Subset (<, <=), Superset (>, >=), Compare (==, !=), Input-Output (>>,<<)

- Linked List class Don't forget to add copy constructor)
  Concatenate (+), Reverse ( ! ), Compare (==), **Copy (=), Subscript ([ ]) – check and display message when index is more than the size of the list, Allocation/Deallocation (new, delete) – using avail list,** Input-Output (>>,<<)

# Summary (Points to note....)

- Operator overloading enables us to develop complete algebra for user-defined types much in same way as it is available for built-in data types

- Enhances readibility:
  assignComplex(a,b.multComplex(addComplex(c,d))) may be expressed as a=b*(c+d)

- Input arguments : Ordinary Arithmetic and Booleans operators will not change their arguments, so pass by const reference instead value; Operators like +=, = ++, --, etc.

- Output arguments: If the effect of the operator is to produce a new value (e.g. +, -, etc.), generate a new static object as the return its reference the return value for all of the assignment operators should be a non-const reference to the lvalue to allow (such as (a=b).func( ) )

- Murray (Rob Murray, "C++ Strategies & Tactics") guidelines for choosing between member/non-member (friend):

| Operator | Recommended use |
|---|---|
| All unary operators | member |
| = ( ) [ ] -> ->* | *must* be member |
| += -= /= *= ^= &= \|= %= >>= <<= | member |
| All other binary operators | non-member |


subrata.nandi@cse.nitdgp.ac.in