W.P.Hwang
CS-CCS-NCTU

Lecture – Class and ADT

CDT and ADT

- A data type consists of a set of values of the same kind and a set of operations acting on the values.
- Concrete data type (CDT)
  1. The data representation is visible to the user of the data type.
- Abstract data type (ADT) (cf. procedural abstraction)
  1. Data abstraction
     One needs only know ***what*** operations on the data are available, and needs not know ***how*** the data are represented.
  2. Data encapsulation
     The data representation is hidden (encapsulated) and can be replaced by another representation without changing the external behavior of the operations.
- Example – Stack as concrete data type

  Stack implementation – Sequential array representation

```
struct stack {                    class stack {
   int top;              ≡        public:
   int stk[80];                      int top,stk[80];
};                                };
```

```
inline void push(stack& s,int n)
{  s.stk[++s.top]=n; }

inline void pop(stack& s) { s.top--; }

inline int& top(stack& s) { return s.stk[s.top]; }

inline const int& top(const stack& s)
{  return s.stk[s.top]; }

inline bool empty(const stack& s)
{  return s.top==-1; }
```

We shall for simplicity ignore stack full and stack empty here.

- Example (Cont'd)

  Having two overloaded **`top`** functions accommodates to STL-style stack.

```
const int& top(const stack&);    // A – peek stack top
int& top(stack&);                // B – modify stack top
```

1. Both use call- and return-by-reference, rather than by-value.
2. Version A is useful when a stack is passed by const reference e.g.
   `void p(const stack& s) { … top(s); … }`
3. Note that version B can't be declared as

   `int& top(const stack&)` // **`const int`** → **`int&`**

   because it cannot coexist with version A. Moreover, it needs **`const_cast`** and has unnatural properties: in functionality of parameter and modifiability of function value.

Stack application – Evaluation of postfix expressions

```
int main()
{
   const int sz=80;
   char exp[sz];
   cout << "Enter a postfix expression: ";
   while (cin.getline(exp,sz)) {
      cout << "Value = " << eval(exp) << endl;
      cout << "\nEnter a postfix expression: ";
   }
}
```

For simplicity, we assume that each line contains a syntactically correct integral postfix expression in which tokens are separated by spaces.

`cin.getline(exp,80)`

1. Read at most 79 characters plus **`'\n'`**
2. Store **`'\0'`**, instead of **`'\n'`**
3. On end-of-file, **`cin`** enters the eof state.
4. If more than 79 characters present, **`cin`** enters the fail state.

- Example (Cont'd)

`cin.getline(exp,6)`

input	exp	cin state
`2 3 +\n`	`2 3 +\0`	`good`
`23 45 +\n`	`23 45\0`	`fail`
`^z`	`-`	`eof`

A postfix expression may be evaluated with the help of a stack.

postfix expression	stack				
`2 3 4 * +`			4		
i.e. `2+3*4`		3	3	12	
	2	2	2	2	14
`2 3 + 4 *`		3		4	
i.e. `(2+3)*4`	2	2	5	5	20

```
int eval(char* exp)
{
   stack s={-1};
   char* p=strtok(exp," ");              ①
   while (p!=NULL) {
      if (strstr("+-*/",p)==NULL)        ②
         push(s,atoi(p));                ③
      else {
         int v=top(s);
         pop(s);
         switch (*p) {
         case '+': top(s)+=v; break;
         case '-': top(s)-=v; break;
         case '*': top(s)*=v; break;
         case '/': top(s)/=v; break;
         }
      }
      p=strtok(NULL," ");                ①
   }
   return top(s);
}
```

- Example (Cont'd)

① **`char* strtok(char* exp,const char* delimiters)`**

This function extracts successively the tokens in **`exp`** that are separated by characters in **`delimiters`**.

```
exp ───> |234  56   +\0|


exp ───> |234\056   +\0|        p=strtok(exp," ")
          ↗
         p
exp ───> |234\056\0 +\0|        p=strtok(NULL," ")
              ↗
             p
exp ───> |234\056\0 +\0|        p=strtok(NULL," ")
                  ↗
                 p
exp ───> |234\056\0 +\0|        p=strtok(NULL," ")
                    p ─┐
                       ═
```

After a first call to **`strtok`**, the function may be called with **`NULL`** as the 1st argument to extract the next token following by where the last call to **`strtok`** found a delimiter.

② **`const char* strstr(const char* s,const char* t);`**
`char* strstr(char* s,const char* t);` // C++ only

These two functions check if **`t`** is a substring of **`s`**.

```
strstr("+-*/","*")    ⇒ "*/"
strstr("+-*/","%")    ⇒ NULL
char s[]="+-*/";
strstr(s,"*")[0]='%'
```

③ **`atoi`**, **`atol`**, and **`atof`** convert strings to **`int`**, **`long`**, and **`double`**, respectively.

```
atoi("      777")     ⇒ 777
atoi("777bingo")      ⇒ 777
atoi("bingo777")      ⇒ 0
```

- Example (Cont'd)

  Disadvantages of CDT

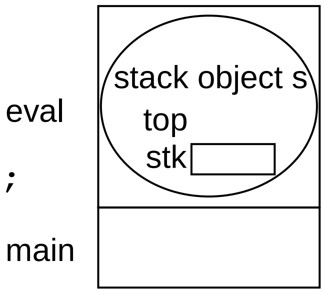
  1. On the application side, the user might mistakenly manipulate the stack, say, by **`s.top*=5`**, but the compiler cannot detect such errors.
  2. The application and implementation are not independent – the application code is mixed up with part of the implementation code.

  For the latter point, assume that the implementer decides to allocate the array dynamically and declares the stack type as follows:

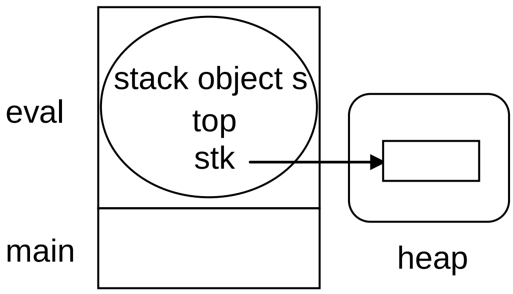
```
struct stack {
   int top;
   int* stk;
};
```

  The implementation of the stack operations remains unchanged. But, the application side has to be modified:

```
int eval(char* exp)
{
   stack s={-1};
   stack s={-1,new int[80]};
          :
   return top(s);
   int r=top(s);
   delete [] s.stk;
   return r;
}
```



runtime stack



runtime stack

- Example – Stack as abstract data type

  Stack implementation – Sequential array representation

```
class stack {
public:
   stack();
   void push(int);                    Interface
   void pop();
   int& top();                        // stack s;
   const int& top() const;            // s.top(); vs top(s);
   bool empty() const;
private:                          ..........................
   int _top;
   int stk[80];                       Implementation
};

inline stack::stack() : _top(-1) {}

inline void stack::push(int n) { stk[++_top]=n; }

inline void stack::pop() { _top--; }

inline int& stack::top() { return stk[_top]; }

inline const int& stack::top() const
{  return stk[_top]; }

inline bool stack::empty() const
{  return _top==-1; }
```

Remarks

1. Member functions defined inside the class definition are inline functions; member functions defined outside are non-inline functions, unless they are declared so.
   The **`inline`** specifier may appear in the declaration or the definition or both.

2. A const member function cannot modify data members.
   The **`const`** qualifier must appear in both the declaration and the definition, as it is a part of the function's type (whereas **`inline`** isn't.)

- Example (Cont'd)

  Stack application – Evaluation of postfix expressions

```
int eval(char* exp)
{
   stack s;                           // *
   char* p=strtok(exp," ");
   while (p!=NULL) {
      if (strstr("+-*/",p)==NULL)
         s.push(atoi(p));
      else {
         int v=s.top();
         s.pop();
         switch (*p) {
         case '+': s.top()+=v; break;
         case '-': s.top()-=v; break;
         case '*': s.top()*=v; break;
         case '/': s.top()/=v; break;
         }
      }
      p=strtok(NULL," ");
   }
   return s.top();
}
```

(cf. evaluate an expression; execute a statement)

The elaboration of the declaration in the starred line will call the default ctor (i.e. a ctor that can be called without an argument) tacitly to initialize the stack.

Q: Why are ctors (i.e. constructors) needed?

A: Since the data representation is hidden, one cannot use a brace-enclosed initializer-list* to initialize the stack as in

```
stack s={-1};
```

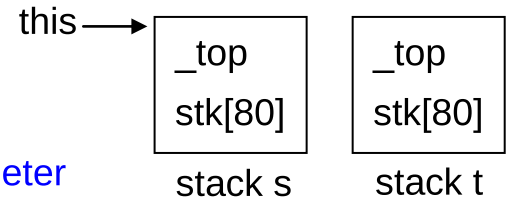
for it implies that the data members are visible.

* Brace-enclosed initializer-lists can only be used to initialize aggregates.
An aggregate is an array or a class with no user-declared ctors, no private or protected non-static data members, no base classes, and no virtual functions.

- Example (Cont'd)

CDT and compiled ADT

```
inline void stack::push(int n) // object-dependent code
{
   stk[++_top]=n;               // which stk? which _top?
}
s.push(atoi(p));
```



are compiled to

implicit object parameter

```
inline void push(stack* this,int n)
{
   this->stk[++this->_top]=n;
}
push(&s,atoi(p));
```

respectively, which are analogous to our earlier CDT operations:

```
inline void push(stack& s,int n)
{
   s.stk[++s.top]=n;
}
push(s,atoi(p));
```

The implicit object pointer may be cv-qualified:

```
inline bool stack::empty() const
{  return _top==-1; }
```

is compiled to

```
inline bool empty(const stack* this)
{  return this->_top==-1; }
```

which is analogous to our earlier CDT operation:

```
inline bool empty(const stack& s)
{  return s.top==-1; }
```

- Example (Cont'd)

  Advantages of ADT

  1 Data encapsulation
  2 The application and implementation are independent.
  3 Code reusability

  For the 2nd point, consider again allocating the array dynamically. The only changes that have to be made are given below.

```
class stack {
public:
   stack();
   ~stack();
   void push(int);
   void pop();
   int& top();
   const int& top() const;
   bool empty() const;
private:
   int _top;
   int* stk;
};
inline stack::stack()
: _top(-1),stk(new int[80])
{}
inline stack::~stack() { delete [] stk; }
```

  In particular, the application side remains the same.

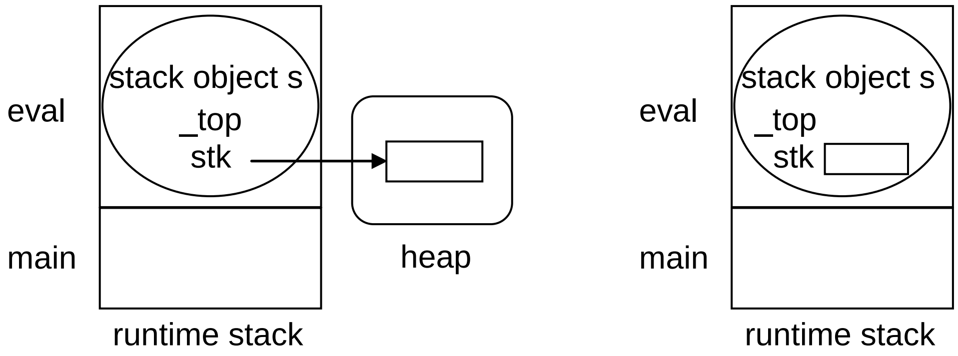
```
int eval(char* exp)
{
   stack s;          // call the ctor tacitly to construct the object
      :
   return s.top();
}                    // call the dtor tacitly to destruct the object
```

  Q: Why are dtors (destructors) needed?
  A: Again, it is because the data representation is hidden.

- Example (Cont'd)

A ctor is invoked when the lifetime of an object begins; the dtor is invoked when the object's lifetime ends.

Principle
Define a dtor for classes with dynamically allocated memory



Stack with dynamically-allocated array   Stack with statically-allocated array

Implicitly generated dtor

The fact that the dtor is invoked automatically when an object's lifetime ends implies that every class must have a dtor.

Instead of burdening the programmer with the task of defining a dtor for a class without dynamically allocated memory, the compiler will implicitly generate one.

For example, the implicitly generated dtor for the **`stack`** class implemented by statically-allocated array reads as

```
inline statck::~stack() {}
```

Notice that this is an inline function. Therefore, a call to it has no compiled code at all.

- Example (Cont'd)

  Now for code reusability.

  The **`stack`** class, once defined, may be kept in a user-defined library and reused in case of need. This is often accompanied with separate compilation.
  For illustration purpose, let's assume that member functions **`push`** and **`pop`** are non-inline.

  User-defined library

  File 1 – stack.h (class interface file)
  This file contains the definitions of the class and inline functions.

```
class stack {
public:
   stack() : _top(-1), stk(new int[80]) {}
   ~stack() { delete [] stk; }
   void push(int);
   void pop();
   int& top() { return stk[_top]; }
   const int& top() const { return stk[_top]; }
   bool empty() const { return _top==-1; }
private:
   int _top;
   int* stk;
};
```

  File 2 – stack.cpp (class implementation file)
  This file contains the definitions of non-inline functions.

```
#include "stack.h"
void stack::push(int n) { stk[++_top]=n; }
void stack::pop() { _top--; }
```

  One Definition Rule (ODR)

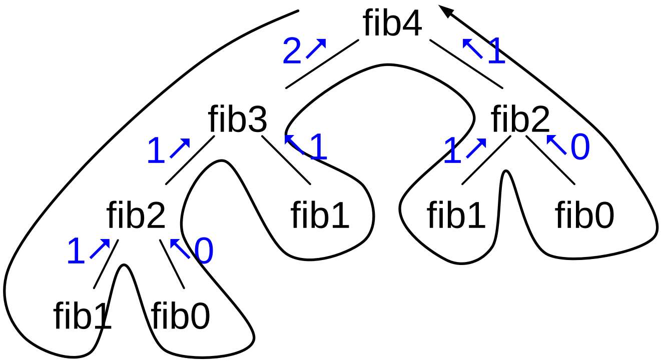
  1 Every program shall contain exactly one definition of every non-inline function or global object that is used in that program
  2 An inline function shall be defined in every translation unit in which it is used.

- Example (Cont'd)

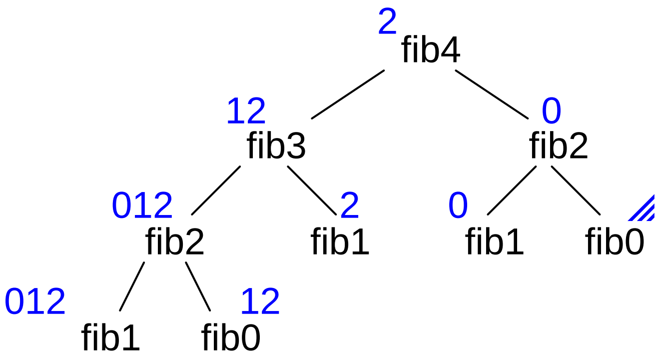
Applications that use the user-defined library

File 3 – fibback.cpp
This file contains an APP that uses backtracking to compute the nth Fibonacci number:

fib(n) $= n$, if $n \leq 1$; $=$ fib($n-1$) + fib($n-2$), otherwise



Iterative backtracking employs a stack.

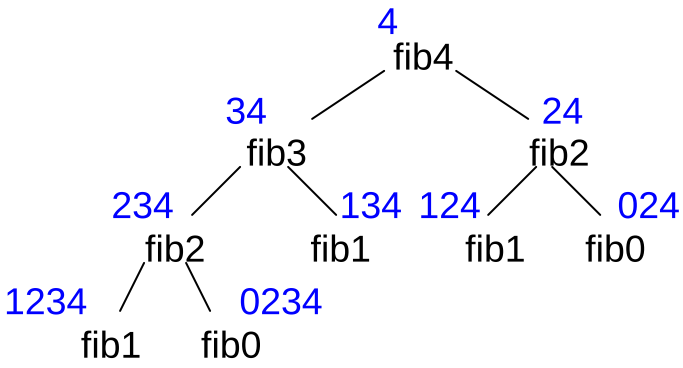



```
#include "stack.h"
int fib(int n)
{
   stack s;
   int r=0;
   do {
      while (n>1) { s.push(n-2); n--; }
      r+=n;
      if (!s.empty()) { n=s.top(); s.pop(); }
      else return r;
   } while (true);
}
```

- Example (Cont'd)

  File 4 – fibsimu.cpp

  This file contains an APP that simulates the runtime stack in the course of computing the nth Fibonacci number.




```
#include "stack.h"
extern stack s;              // declaration; external linkage
int fib()                    // definition; external linkage
{
   if (s.top()<=1) {
      int r=s.top(); s.pop(); return r;
   } else {
      s.push(s.top()-1); int a=fib();
      s.push(s.top()-2); int b=fib();
      s.pop();
      return a+b;
   }
}
```

Alternatively, **`fib`** may be coded as follows:

```
int fib()
{
   if (s.top()<=1) return s.top();
   else {
      s.push(s.top()-1); int a=fib(); s.pop();
      s.push(s.top()-2); int b=fib(); s.pop();
      return a+b;
   }
}
```

After this function returns, the caller shall pop off the stack top.

- Example (Cont'd)

File 5 – main.cpp

```
#include <iostream>
#include "stack.h"
using namespace std;
stack s;                    // definition; external linkage
int fib(),fib(int);         // declaration; external linkage
int main()
{
   int n;
   cout << "Enter an integer: ";
   while (cin >> n) {
      cout << "By backtracking: ";
      cout << "fib(" << n << ") = " << fib(n);
      s.push(n);
      cout << "By runtime stack simulation: ";
      cout << "fib(" << n << ") = " << fib();
      cout << "\n\nEnter an integer: ";
   }
}
```

On declarations and definitions of functions and objects

	Function	Object
Definition	with body	without **extern** or with initializer
Declaration	without body	with **extern** and without initializer

Definitions

```
stack s;
int x;
int x=2;
extern int x=2;
int f(int n) { return n; }          // define f and n
extern int f(int n) { return n; }
```

Declarations

```
extern stack s;
extern int x;
int f(int);
extern int f(int);
```

- Example (Cont'd)

  On linkage

  A name may have external linkage, internal linkage, or no linkage

  External linkage
  1. The name is visible in every translation unit of the program.
  2. e.g. functions and global objects (possibly declared **`extern`**)

  Internal linkage
  1. The name is visible only in the translation unit in which it is declared.
  2. e.g. functions and global objects declared **`static`**

  No linkage
  1. The name is visible only in the scope in which it is declared.
  2. e.g. local objects

  File 3 – fibback.cpp (revision 1) // File 4: **`extern stack s;`**
  // File 5: **`stack s;`**

```
#include "stack.h"
static stack s;           // definition; internal linkage
static void moveon(int& n)
{                         // definition; internal linkage (moveon)
    while (n>1) {
        s.push(n-2); n--;
    }
}
int fib(int n)            // definition; external linkage (fib)
{
    int r=0;              // definition; no linkage
    do {
        moveon(n);
        r+=n;
        if (!s.empty()) { n=s.top(); s.pop(); }
        else return r;
    } while (true);
}
```

- Example (Cont'd)

  Principle
  Things with distinct semantics shouldn't have similar syntax.

  The **`static`** specifier has two distinct meanings.
  1 Local static object – Related to lifetime
  2 Global static object/function – Related to scope (or linkage)

  C++ solution – Use unnamed namespaces instead of global static objects and functions.

  File 3 – fibback.cpp (revision 2)

```
#include "stack.h"
namespace {
   stack s;
   void moveon(int& n)
   {
      while (n>1) { s.push(n-2); n--; }
   }
}
int fib(int n)            // same as revision 1
{
   … moveon(n); … s.pop(); …
}
```

Properties of unnamed (or anonymous) namespace

- Members of an anonymous namespace are referred to without qualification.
  N.B. This is similar to names declared in the global namespace. In case of name conflict, members of an unnamed namespace are invisible.

```
namespace { int x=2; }
int x=3;
int main()
{
   cout << x;         // ambiguous!
   cout << ::x;       // global x
}
```

- Members of an unnamed namespace are visible only in the translation unit containing the unnamed namespace.

  Why? It is because an unnamed namespace

  ```
     namespace { body }
  ```

  is effectively compiled to

  ```
     namespace unique { body }
     using namespace unique;
  ```

  where *unique* is a compiler-generated identifier that differs from all other identifiers in the entire program.

  Example

  File X.cpp

  ```
  #include <iostream>
  using namespace std;
  namespace A { int f(); }
  int main() { cout << A::f(); }
  namespace A { int x=2; }
  ```

  File Y.cpp

  ```
  namespace A {
     extern int x;
     int f() { return x; }
  }
  ```

  Were the namespaces unnamed, they would be two distinct namespaces (one for each file), and the program is ill-formed.

  N.B. The definition of a namespace may be split over several parts in one or more compilation units.

Dynamic storage management (Part II)

- Dynamic storage allocation and deallocation for objects of non-POD types involve ctors and dtors, respectively.

- Single objects

  `new T(`*arguments*, *if any***`)`** where **`T`** is a non-POD type

  1 Call the appropriate ctor to create an object in the storage obtained from **`operator new`**.
  2 It is an error, if there is no appropriate constructor.
  3 In case of no arguments, the parentheses **`()`** may be omitted.

  `delete p` where **`p`** is **`T*`**, **`T`** is a non-POD type

  1 First, call **`T`**'s dtor to destroy the object pointed to by **`p`**
  2 Then, call **`operator delete`** to free the storage

- Array objects

  `new T[`n**`]`**

  1 Call the default ctor n times to create n objects in the storage obtained from **`operator new[]`**.
  2 It is an error, if there is no default constructor.

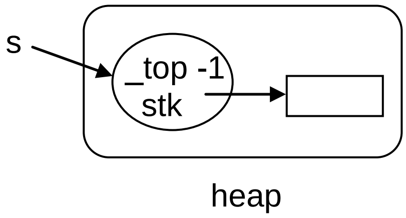
  `delete [] p` where **`p`** is **`T*`**, **`T`** is a non-POD type

  1 First, call **`T`**'s dtor as many times as there are objects in the storage pointed to by **`p`** to destroy them in the reverse order of their construction
  2 Then, call **`operator delete[]`** to free the storage

- Example

```
                  int n                            n
inline stack::stack() : _top(-1),stk(new int[80]) {}
inline stack::~stack() { delete [] stk; }

stack* s=new stack;
delete s;                80
          // or, stack()
```

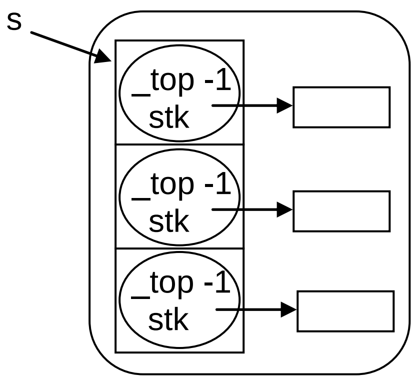
- Example (Cont'd)



or

```
stack* s=new (operator new(sizeof(stack))) stack;
s->~stack();
operator delete(s);                         // or, stack()
```

For array object:



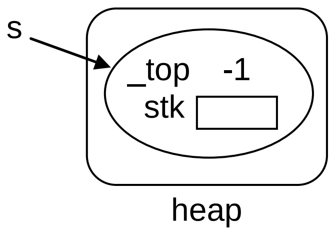

```
stack* s=new stack[3];
delete [] s;
```

or

```
void* buf=operator new[](3*sizeof(stack));
stack* s=static_cast<stack*>(buf);        80+i
for (int i=0;i<3;i++) new (s+i) stack();
for (int i=2;i>=0;i--) s[i].~stack();
operator delete[](s);
```

- Example

```
inline stack::stack() : _top(-1) {}
inline stack::~stack() {}
```




```
stack* s=new stack;
delete s;
```

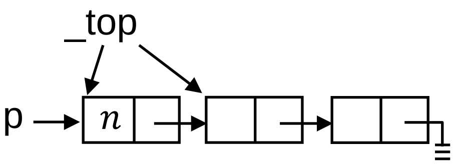
or

```
stack* s=new (operator new(sizeof(stack))) stack;
s->~stack();                // no compiled code indeed
operator delete(s);
```

- Example

Stack implementation – Linked list representation

```
class stack {
public:
   stack() : _top(NULL) {}
   ~stack() { while (!empty()) pop(); }
   void push(int);
   void pop();
   int& top() { return _top->datum; }
   const int& top() const { return _top->datum; }
   bool empty() const { return _top==NULL; }
private:
   struct node {
      node(int,node*);  // public members of class node
      int datum;        // but, visible only within the node
      node* succ;       // and stack classes
   };
   node* _top;
};

inline stack::node::node(int d,node* s)
:  datum(d),succ(s) {}

inline void stack::push(int n)
{
   _top=new node(n,_top);
}

inline void stack::pop()
{
   node* p=_top;
   _top=_top->succ;
   delete p;            //* invoke p->~node() tacitly
}                       // no compiled code indeed
```



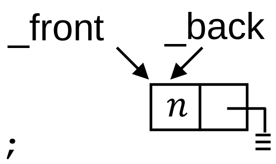
The starred line calls the implicitly generated dtor of class **node**

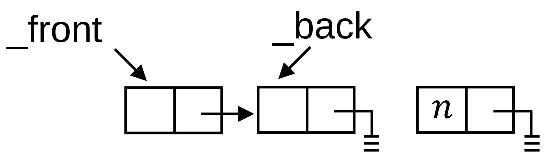
```
inline stack::node::~node() {}
```

- Example

  Queue implementation – Linked list representation

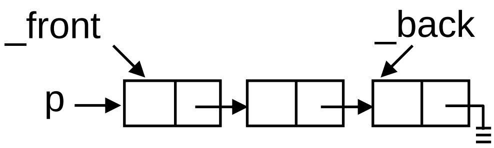
```
class queue {
public:
   queue() : _front(NULL) {}
   ~queue() { while (!empty()) pop(); }

   void push(int);           // enqueue
   void pop();               // dequeue

   int& front() { return _front->datum; }
   const int& front() const { return _front->datum; }
   bool empty() const { return _front==NULL; }
private:
   struct node;              // class declaration
   node *_front,*_back;
};

struct queue::node {         // class definition
   node(int,node*);
   int datum;
   node* succ;
};

inline queue::node::node(int d,node* s)
: datum(d),succ(s) {}

inline void queue::push(int n)
{
   if (empty())
      _front=_back=new node(n,NULL);
   else {
      _back->succ=new node(n,NULL);
      _back=_back->succ;
   }
}
```





- Example (Cont'd)

```
inline void queue::pop()
{
   node* p=_front;
   _front=_front->succ;
   delete p;
}
```



Queue application – Palindrome

```
bool palindrome(unsigned n)
{
   stack s; queue q;
   while (n>0) {
      int d=n%10;
      s.push(d); q.push(d);
      n/=10;
   }
   while (!s.empty())
      if (s.top()==q.front()) {
         s.pop(); q.pop();
      } else
         return false;
   return true;
}
```

Before return to the caller, call **`q.~queue()`** and **`s.~stack()`**, in that order.

For example, n = 12321
stack s   _top $\rightarrow$ 1 2 3 2 1
queue q   _front $\rightarrow$ 1 2 3 2 1 $\leftarrow$ back

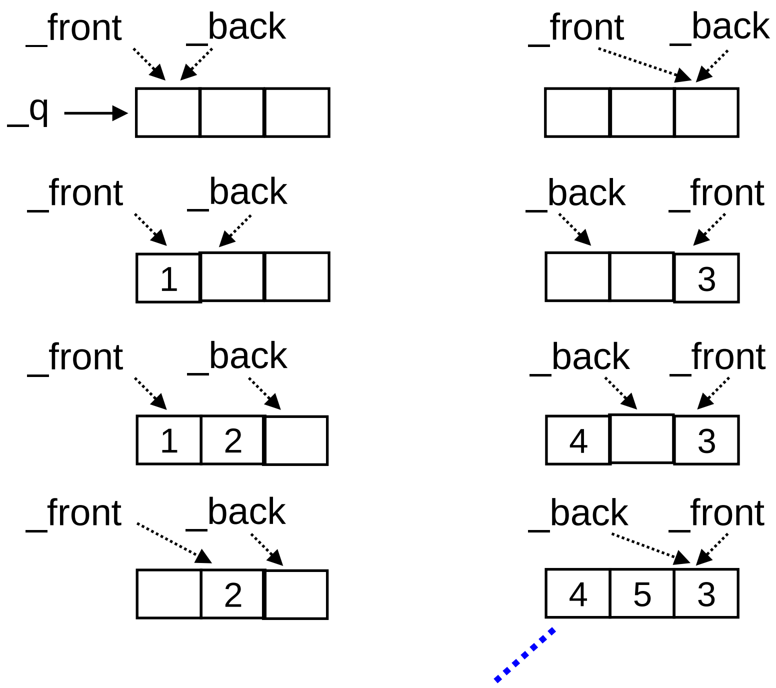
Comment
The loop may be terminated earlier, if the stack or queue size is known.

- Example

Queue implementation – Sequential array representation

```
class queue {
public:
   queue(int=80);            // default argument
   ~queue();
   void push(int);
   void pop();
   int& front();
   const int& front() const;
   bool empty() const;
private:
   int _front,_back,_capacity;
   int* _q;
};
```

A queue can store at most one less element than the array size.



No! Cannot distinguish between queue full and queue empty

N.B. A queue is full if `_front==(_back+1)%(_capacity+1)`.

- Example (Cont'd)

```
inline queue::queue(int n)
:  _front(0),_back(0),     // or, any number from 0 to n
   _capacity(n),
   _q(new int[n+1])
{}
```

Comments

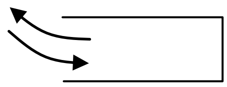
1 The default argument may appear in the declaration (common usage) or the definition of the member function, but not both.

2
```
queue q(10);          // call queue(10)
queue q;              // call queue(80)
queue q();            // error! look like a function signature
new queue(10)         // call queue(10)
new queue             // call queue(80)
new queue()           // call queue(80)
```

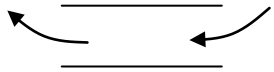
```
inline queue::~queue() { delete [] _q; }

inline void queue::push(int n)
{
   _q[_back]=n; _back=(_back+1)%(_capacity+1);
}

inline void queue::pop()
{
   _front=(_front+1)%(_capacity+1);
}

inline int& queue::front() { return _q[_front]; }

inline const int& queue::front() const
{
   return _q[_front];
}

inline bool queue::empty() const
{
   return _front==_back;
}
```

STL stack, queue, and deque

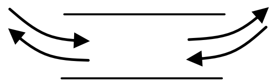
- Stack (FILO, First-In-Last-Out)
  Insert and delete at the same end

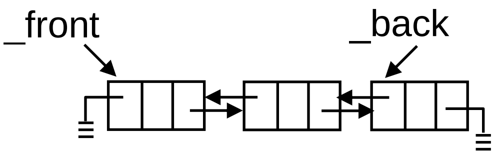


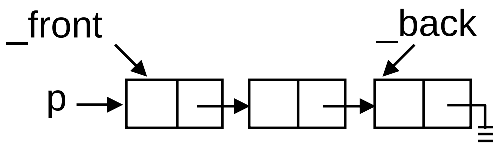
- Queue (FIFO, First-In-First-Out)
  Insert at one end and delete at the other end



- Deque (Doubly-ended queue)
  Insert and delete at both ends
  A deque may be implemented by an array or a doubly linked list.







pop-back is inefficient

- Operations

	stack	queue	deque
insert	push	push	push_front, push_back
delete	pop	pop	pop_front, pop_back
peek	top	front	front, back

- Example – stack and queue

```
#include <stack>
#include <queue>
bool palindrome(unsigned n)
{
   stack<int> s; queue<int> q;
   while (n>0) {
      int d=n%10; s.push(d); q.push(d); n/=10;
   }
   stack<int>::size_type c=s.size();
// queue<int>::size_type c=q.size();
   for (int i=1;i<=c/2;i++)
      if (s.top()==q.front()) { s.pop(); q.pop(); }
      else return false;
   return true;
}
```

- Example (Cont'd)

In STL, the containers (i.e. data structures) are required to define several public **typedef** names.
E.g. let **X** be a container containing objects of type **T**, then

```
X::size_type                // unsigned integral type
X::value_type               // T
```

```
template<typename T>        // T is a queue or deque class
void p(T& c)                // or, class
{
   typename T::value_type *p=&c.front(); ...
}
```

On dependent name

The blue-colored code has two interpretations:

1 **value_type** is a type defined in **T**
  And, **p** is a pointer

3 **value_type** is an enumerator or a static data member of **T**
  And, multiply it with **p**

T::value_type is called a dependent name.
A dependent name is parsed as a non-type, unless it is prefixed by the keyword **typename**.

- Example – deque

```
#include <deque>
bool palindrome(unsigned n)
{
   deque<int> d;
   while (n>0) { d.push_back(n%10); n/=10; }
   deque<int>::size_type c=d.size();
   for (int i=1;i<=c/2;i++)
      if (d.front()==d.back()) {
         d.pop_front(); d.pop_back();
      } else return false;
    return true;
}
```

- Example – class template **stack**; linked-list representation

```
template<typename T>
class stack {
public:
   typedef size_t size_type;
   typedef T value_type;
   stack();
   ~stack();
   void push(const value_type&);   // or, const T&
   void pop();
   value_type& top();
   const value_type& top() const;
   size_type size() const;
   bool empty() const;
private:
   struct node;
   node* _top;
   size_type sz;
};

template<typename T>
struct stack<T>::node {
   node(const T&,node*);   // or, const value_type&
   T datum;
   node* succ;
};

template<typename T>
stack<T>::node::node(const T& d,node* s)
: datum(d),succ(s) {}

template<typename T>
stack<T>::stack() : _top(NULL),sz(0) {}
```

Remark

For non-template classes, class name = type name

stack **stack s;**

For template classes, class name ≠ type name

stack **stack<int> s;**

- Example (Cont'd)

```
template<typename T>
stack<T>::~stack() { while (!empty()) pop(); }

template<typename T>
bool stack<T>::empty() const { return sz==0; }

template<typename T>
void stack<T>::push(const value_type& n)
{
    _top=new node(n,_top); sz++;   // within class scope
}

template<typename T>
void stack<T>::pop()
{
   node* p=_top; _top=_top->succ; delete p; sz--;
}

template<typename T>
typename stack<T>::value_type& stack<T>::top()
{
   return _top->datum;
}

template<typename T>
const typename stack<T>::value_type&
stack<T>::top() const
{
   return _top->datum;
}

template<typename T>
typename stack<T>::size_type stack<T>::size() const
{
   return sz;
}
```

// Outside the class scope, must be qualified
// Dependent name, must be prefixed by **typename**.

- Example (Cont'd)

  Implicit member template

  A member function of a class template is implictly a function template.
  The template arguments for a member function are determined by the template arguments of the type of the object for which the member function is called, rather than by the types of the function arguments.

```
template<typename T>
void stack<T>::push(const T&);
    T = int                          T = double? NO!
                                     double → int
stack<int> s; s.push(3.14);
```

cf.
```
template<typename T>
void p(const T&);
                 T = double
     p(3.14);
```

If you really wish **T = double** in this case, do this:

```
template<typename T>
class stack {
public:
   template<typename U> void push(const U&);
};

template<typename T>
template<typename U>
void stack<T>::push(const U& n)
{
   _top=new node(n,_top); sz++;
}
           T = int           U = double
stack<int> s; s.push(3.14);
                             double → int
template<typename T>
stack<T>::node::node(const T&,node*);
```

- Example – class template **`stack`**; sequential-array representation

```
template<typename T>
class stack {
public:
   typedef size_t size_type;
   typedef T value_type;
   stack(size_type=80);
   ~stack();
   void push(const value_type&);
   void pop();
   value_type& top() { return stk[_top]; }
   const value_type& top() const { return stk[_top]; }
   size_type size() const { return sz; }
   bool empty() const { return sz==0; }
private:
   T* stk;
   int _top;
   size_type sz;
};

template<typename T>
stack<T>::stack(size_type n)
:  stk((T*)operator new[](n*sizeof(T))),  //*
   _top(-1),
   sz(0)
{}
```

Q: Can we write **`new T[n]`** in the starred line?
A: No, as it would undesirably initialize each array element by **`T::T()`**, in case **`T`** is a class type.

```
template<typename T>
stack<T>::~stack()
{
   while (!empty()) pop();
   operator delete[](stk);
}
```

- Example (Cont'd)

```
template<typename T>
void stack<T>::push(const value_type& n)
{
   ++_top;
   new (stk+_top) T(n);        //* copy construction
   sz++;
}
```

Q: Can we write `stk[_top]=n;` in the starred line?

A: No, because the space `stk[_top]` is uninitialized. (It might be occupied and destroyed before.)
In case `T` is a class type, this would be a disaster. (To be explained later.)

```
template<typename T>
void stack<T>::pop()
{
   stk[_top].~T(); _top--;    //*
   sz--;
}
```

Pseudo destructors support generic programming.

1 If `T` isn't a class type, `~T()` is called a pseudo destructor.

2 The only effect of a pseudo destructor call
***exp*`.~T()`**
is the evaluation of ***exp***.
Thus, the two statements in the starred line may be combined into
`stk[_top--].~T();`

Nonstatic members

- Nonstatic members belong to each object of the class.
- They may be declared **`const`**, **`volatile`**, or **`const volatile`**.
- A cv-qualified nonstatic member function can be called on an object if the cv-qualification of the object is less than or equal to that of the member function. (cv-qualification is a partial order.)

  In other words, cv-qualified objects can only access a subset of the interface of the class.
- The type of the keyword **`this`** in a cv-qualified nonstatic member function of a class **`X`** is *cv-qualified* **`X*`**.
- Example (See Appendix)

```
                         string* this
string::reference
string::operator[](size_type pos)                ①
{
   return _data[pos];
}
                         const string* this
string::const_reference
string::operator[](size_type pos) const          ②
{
   return _data[pos];   // [] for built-in type
}

string a("snoopy");
const string b("snoopy");
a[0]='S';                // both are viable; ① is the best viable
cout << b[0];            // only ② is viable
```

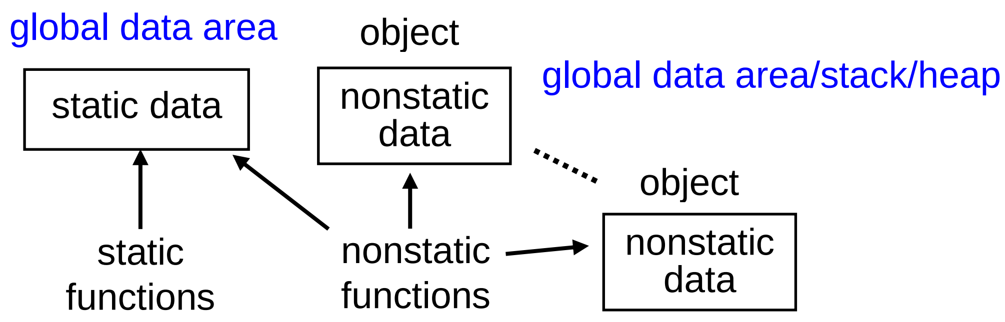
  where **`a[0]`** ≡ **`a.operator[](0)`** ⇢ **`operator[](&a,0)`**

- Ctors and dtors cannot be cv-qualified, but can be called for cv-qualified objects.
  A cv-qualified object is considered cv-qualified from the end of its construction to the beginning of its destruction.

Static members

- Static members belong to the whole class.
  In particular, a static data member isn't a part of an object of a class. There is only one copy of a static data member shared by all objects of the class.

- A static member function doesn't have a **`this`** pointer and can only access static members.
  A nonstatic member function has a **`this`** pointer and may access both static and nonstatic members.



- Static data members are not initialized by ctors; they have to be initialized outside the class body in the implementation file.
  Exception: Constant static data members of integral type may be initialized within the class body.

- Example (Hypothetical)

```
class string {          watch the space
public:
   string(const char* =_dstring);       ①
   static void dstring(const char*);    ②
private:
   static const int _dsize=16;*         ③
   static char _dstring[_dsize];
   char* _data;
};
```

---

* The current C++ standard treats this as a declaration and requires that it be defined outside the class body w/o the initializer:

```
const int string::_dsize;
```

This won't be required in the future, leaving an exception of one definition rule.

- Example (Cont'd)

```
char string::_dstring[_dsize]="";
void string::dstring(const char* s)
{
   strncpy(_dstring,s,_dsize-1);
}
string::dstring("snoopy");                    ④
string a;
a.dstring("pluto");                           ④
```

① Nonstatic members cannot be used as default parameters.

② The keyword **`static`** can only appear in the declaration, but not in the definition. Why?

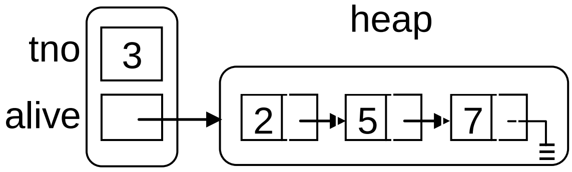
③ **`_dsize`** must be initialized within the class body so that the compiler can determine the array size.

④ Static members may be accessed in either way.
In *exp*`.`static-member or *exp*`->`static-member, the *exp* will be evaluated, e.g.
`(cout << a,a).dstring("pluto");`

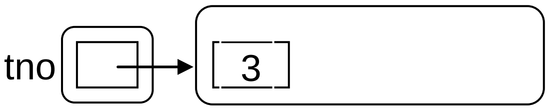
- Static data members are initialized and destroyed exactly like non-local (i.e. global) objects.

- Example

Suppose we have a **`tank`** class to record the # of tanks alive in the field and a list of alive tanks as static data members. The problem is how to free the list at the end of the program; but, for simplicity, we illustrate it with the # of alive tanks. That is,



is simplied to



- Example (Cont'd)

```
class tank {
public:
   tank(int tid) : tid(tid) { (*tno)++; }
   ~tank() { (*tno)--; }
   static int alive() { return *tno; }
private:
   int tid;
   static int* tno;
};
int *tank::tno=new int(0);

int main()
{
   cout << tank::alive() << endl;
   tank a(1);
   {
      tank b(2),c(3);
      cout << tank::alive() << endl;
   }
   cout << tank::alive() << endl;
}
```

Comments

1 The non-static ctor accesses both non-static and static data members. On the other hand, the static function **alive** only accesses static data member.

2 A dtor is defined even though no dynamic storage is allocated for **tank** objects.

3 **tank::tno** is initialized before **main** is called.

Now, consider the destruction of **tank::tno**.

Since it is a "raw pointer", the **int** object pointed to by it won't be automatically destroyed on returning from **main**.

Moreover, it is private and so we can't delete it before returning from **main**.

- Example (Cont'd)

Q: Ma we have a public static member function to delete **`tank::tno`** and call it *manually* before returning from **`main`**?
A: Bad idea – prefer automaic deletion

What we need is a "smart pointer", i.e. a pointer-like object whose dtor will delete the pointed-to object when it is destroyed.

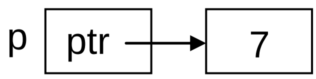
`auto_ptr`

1 a class template in STL, declared in **`<memory>`**
2 one of many kinds of smart pointers

A stripped-down version of **`auto_ptr`** is given below.

```
template<typename T>
class auto_ptr {
public:
   auto_ptr(T* p=0) : ptr(p) {}
   ~auto_ptr() { delete ptr; }
   T& operator*() const { return *ptr; }
private:
   T* ptr;
};
```

```
auto_ptr<int> p(new int(7));
(*p)++ ≡ p.operator*()++;
```



Here is a rewrite of the preceding example

```
#include <memory>
class tank {
public:
   tank(int tid) : tid(tid) { (*tno)++; }
   ~tank() { (*tno)--; }
   static int alive() { return *tno; }
private:
   int tid;
   static auto_ptr<int> tno;
};
auto_ptr<int> tank::tno(new int(0));
```

- Example (Cont'd)

  Remark
  Use **auto_ptr** objects instead of raw pointers, and you needn't bother to free the pointed-to objects by yourself.

  For example, we may rewrite

```
void p(int n)
{
   int* p=new int(n);
   cout << *p;
   delete p;
}
```

  as

```
void p(int n)
{
   auto_ptr<int> p(new int(n));
   cout << *p;
}
```

  auto_ptr shall be used with care.

```
int x;
auto_ptr<int> a(&x);              // no
auto_ptr<int> a(new int[7]);      // no
```

  Note that an **auto_ptr** object can't point to an array of objects, for its dtor uses the single-object **delete** operator.

  Finally, **auto_ptr** is a rare class that doesn't allocate dynamic storage, yet it has a dtor.

- A local class shall not have static data members.
- A static member function can't be cv-qualified or overloaded with a nonstatic member function.
- A ctor or dtor can't be a static member function.

Class scope

- A class defines a scope.

  `class X { };`

  return-type **X**'s-member-function ( ) { }

  X's-static-data-member [ ] = ;

  Inside the class scope, class members may be accessed by their names alone. Outside the class scope, class members must be accessed by the member access operators (`.` and `->`) or the scope resolution operator (`::`).

- Name resolution in class scope
  1. Process member declarations in order.
  2. Process member definitions (either inside or outside the class definition), including default arguments and ctor-initializers, in the completed class scope
  3. A name used in a class shall refer to the same declaration in its context and when re-evaluated in the completed scope of the class.

- Example (Hypothetical)

```
class string {
public:
   reference operator[](size_type);          1X
   typedef size_t size_type;
   typedef char&  reference;
   string(const char* s=_dstring)            2O
   : _data(...) {}                           2O
   static void dstring(const char* s)
   { strncpy(_dstring,s,_dsize-1); }         2O
private:
   static char _dstring[_dsize];             1X
   static const int _dsize=16;
   char* _data;
};
```

- Example (Cont'd)

```
char string::_dstring[_dsize]="";          2O
string::reference
string::operator[](size_type pos)          2O
{
    return _data[pos];                     2O
}
```

- Example

```
typedef int size_type;
typedef int& reference;
const int _dsize=32;
class string {
public:
    reference operator[](size_type);       3X
    typedef size_t size_type;              ↑
    typedef char& reference;         change meaning
private:                                   ↓
    static char _dstring[_dsize];          3X
    static const int _dsize=16;
};
```

- Example

```
int x=3;
class X {
public:
    X(int x) : x(x)
    {
        cout << x << this->x << X::x << ::x;
    }
private:                         this->X::x
    int x;
};
```

`X::x` means "look up the name `x` in the class `X`".
`::x` means "look up the name `x` in the global namespace".

Special member functions

- Default ctor, copy ctor, copy assignment operator, destructor, and address-of operator are special member functions
- If a class doesn't explicitly declare them, the implementation will
  1. implicitly declare them
  2. implicitly define them, if they are needed.
- Example

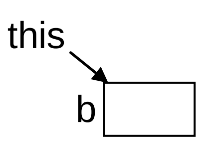
```
class X {};
```

is compiled to something like

```
class X {
public:
   X();
   ~X();
   X(const X&);
   X& operator=(const X&);
   X* operator&();
   const X* operator&() const;
};
```

These functions won't be defined unless they are needed.

```
X a;              // ctor and dtor
X b=a;            // copy ctor
b=a;              // copy assignment operator
X* c=&b;          // address-of operator
const X d;
const X* e=&d;    // const address-of operator
```



They are defined as

```
inline X::X() {}                          // &b = b.operator&()
inline X::~X() {}
inline X* X::operator&() { return this; }
inline const X* X::operator&() const { return this; }
inline X::X(const X&) : memberwise initialization {}
inline X& X::operator=(const X&){ memberwise assignment }
```

Constructors

- A constructor is used to initialize objects of its class type.
- A default ctor is a ctor that can be called without an argument.
- If a class does not declare a ctor (including copy ctor), a default ctor is declared implicitly.
- Member initializer list (ctor initializer)

```
X::X(...)
:  mem_id(exp), ...          phase 1 – Initialization
{
   statements, if any        phase 2 – Assignment
}
```

  If **exp** is omitted, **mem_id** is default-initialized (zero for non-class type); otherwise, it is direct-initialized.

  If a nonstatic data member is not named by a **mem_id** in the initializer list, then
  1 if the data member is of class type, it is default-initialized;
  2 otherwise, it is not initialized.

  The initialization phase initializes nonstatic data members in the declaration order – independent of the order of initializers.

  Const and reference data members must be initialized in the member initializer list.

- Example

```
class X {
public:
   X(int); const int& vx;
private:
   int x;
};
X::X(int x): vx(this->x),x(x) {}
X::X(int x): vx(this->x) { this->x=x; }
X::X(int x): x(x) { vx=this->x; } ✗
```

- Example

```
class string {
public:
   typedef size_t size_type;
   string(const char* ="");
private:
   size_type _cap(size_type);
   size_type _size,_capacity;
   char* _data;

};
string::string(const char* s)
:  _size(strlen(s)),
   _capacity(_cap(_size)),
   _data(strcpy(new char[_capacity+1],s))
{}
string::size_type string::_cap(size_type sz)
{
   size_type cap=15;
   while (cap<sz) (cap<<=1)++;
   return cap;
}
```

Notice that the member initializers may be listed in any order. But, as a good style,

List member initializers in the declaration order

so that what is really going on can be easily seen.

On the other hand, for this ctor to work, the declaration order of the data members has to be left as it is.

The following two definitions of the ctor are independent of the declaration order of the data members. The former is inefficient, and the latter uses assignments.

```
string::string(const char* s)
:  _size(strlen(s)),
   _capacity(_cap(strlen(s))),
   _data(strcpy(new char[_cap(strlen(s))+1],s))
{}
```

- Example (Cont'd)

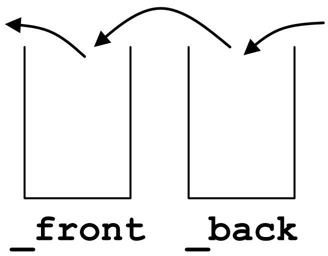
```
string::string(const char* s)
{
   _size=strlen(s);
   _capacity=15;
   while (_capacity<_size) (_capacity<<=1)++;
   _data=strcpy(new char[_capacity+1],s);
}
```

- Example – Queue as a pair of stacks

Invariant 1: queue = front ++ reverse back
Invariant 2: front is empty only if back is empty

Push takes $O(1)$ worst-case time, and pop takes $O(n)$ worst-case time. Both push and pop take $O(1)$ amortized time.

```
class queue {
public:
   void push(int);
   void pop();
   int& front();
   const int& front() const;
   bool empty() const;
private:
   void _check();
   stack _front,_back;
};
```




```
void queue::push(int n) { _back.push(n); _check(); }

void queue::pop() { _front.pop(); _check(); }

int& queue::front() { return _front.top(); }

const int& queue::front() const
{  return _front.top(); }

bool queue::empty() const { return _front.empty(); }
```

- Example (Cont'd)

```
void queue::_check()
{
   if (_front.empty())
      while (!_back.empty()) {
         _front.push(_back.top()); _back.pop();
      }
}
```

Q: Do we need to define a constructor for the **queue** class?

A: A **queue** object contains no data other than two **stack** sub-objects that can only be initialized by **stack**'s ctors. The only job of a **queue**'s ctor is to invoke an appropriate ctor of class **stack** for each **stack** subobject.

Case 1: The **stack** class has only the default constructor. For examples,

```
stack::stack() : _top(NULL) {}
stack::stack() : _top(-1),stk(new int[80]) {}
```

In this case, we simply use the implicit default constructor

```
queue::queue() {}
```

which is equivalent to

```
queue::queue() : _front(),_back() {}
```

Case 2: The **stack** class has only non-default constructors. For example,

```
stack::stack(int n)
: _top(-1),stk(new int[n]) {}
```

In this case, we have to define a constructor, say

```
queue::queue() : _front(80),_back(80) {}
```

Case 3: This is a combination of the preceding two cases. For example,

```
stack::stack(int n=80)
: _top(-1),stk(new int[n]) {}
```

One has to choose between case 1 and case 2.

Destructors

- A destructor is used to destroy objects of its class type.
- A class may have several ctors, but can only have one dtor. Moreover, the dtor must be parameterless.
- If a class does not declare a dtor, a dtor is declared implicitly.
- Before returning from the dtor, the destructors for nonstatic data members that are of class type are called in reverse order of their construction.
- Example (Queue cont'd)

  Q: Do we need to define a destructor for the **`queue`** class?
  A: Again, the two **`stack`** subobjects can only be destructed by **`stack`**'s dtor. The job of **`queue`**'s dtor is to invoke the **`stack`**'s dtor to destroy them.
  Moreover, the **`queue`** class doesn't allocate dynamic storage. (there are no other data at all). Thus, by the principle

  Define a dtor for classes with dynamically allocated memory

  we may simply use the implicit generated dtor

```
queue::~queue() {}
```

  which isn't the same as

```
queue::~queue() { _back.~stack();_front.~stack(); }
```

  since the latter destroys **`_front`** and **`_back`** twice.

- Principle

  Invoke the dtor by yourself only when you use placement new

1. `{ queue q; }` // automatic call **`q.~queue()`**
2. `queue* q=new queue;`
   `delete q;` // automatic call **`q->~queue()`**
3. `queue* q=new (operator new(sizeof(queue))) queue;`
   `q->~queue();` // manual call
   `operator delete(q);`

Copy constructors

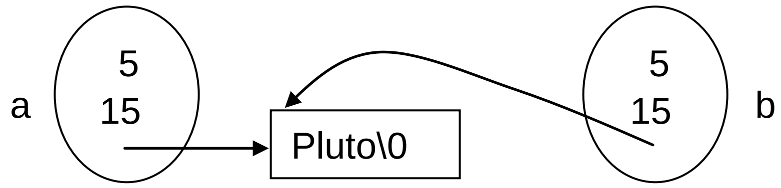
- A copy ctor is used to initialize one object by another object of the same class.
- Example

Assume that the **`string`** class adopts the implicit copy ctor that does memberwise initialization.

```
string::string(const string& rhs)
:  _size(rhs._size),
   _capacity(rhs._capacity),
   _data(rhs._data) {}
```

Notice: Objects of the same class may refer to the private data of each other.

```
int main()
{
   string a("Pluto");
   {
      string b(a);    // call the implicit copy ctor
      b[0]='p';
      cout << a;      // a was silently modified
   }                  // call b.~string()
   cout << a;         // "pluto" has already been freed!
}                     // call a.~string() to free it again!
```



Remark
Sharing is efficient, but suffers from the dangling pointer problem.

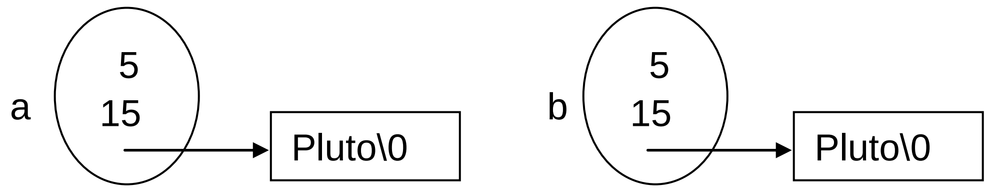
- Principle

Define a copy ctor for classes with dynamically allocated memory

```
string::string(const string& rhs)   // const
:   _size(rhs._size),
    _capacity(rhs._capacity),
    _data(strcpy(new char[_capacity+1],rhs._data))
{}
```

- Example (Cont'd)

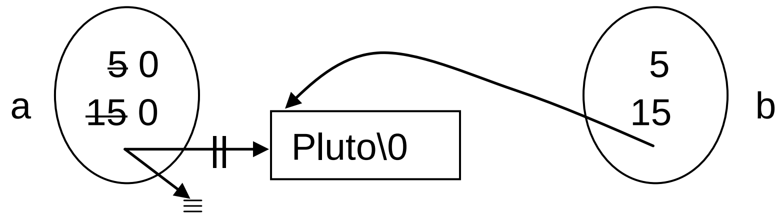
With this built-in copy ctor for the **string** class, we have



Remark – Copying is time and space consuming, but solves the dangling pointer problem.

Occasionally, we may adopt a "destructive copy" semantics. For example, we may wish to transfer the ownership by invalidating the source.

```
string::string(string& rhs)          // non-const
:   _size(rhs._size),
    _capacity(rhs._capacity),
    _data(rhs._data)
{
    rhs._size=rhs._capacity=0; rhs._data=NULL;
}
```

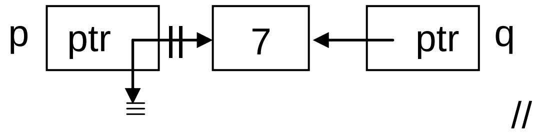


Remark
This is efficient and solves the dangling pointer problem, but is restricted to a single owner.

- Example

  auto_ptr has a semantics of strict ownership.

```
auto_ptr<int> p(new int(7));     // p is the owner
auto_ptr<int> q(p);              // q is the owner
cout << *q;                      // okay
cout << *p;                      // error
```




```
p=q;                             // p is the owner
cout << *p;                      // okay
cout << *q;                      // error
```

  Remark – Don't pass an **auto_ptr** object by value.

```
void r(auto_ptr<int> q) { cout << *q; }
auto_ptr<int> p(new int(7));
r(p);
cout << *p;                      // error
```

- A ctor for class **X** is a copy ctor if its first parameter is of type **X&** (possibly cv-qualified), and all the other parameters (if any) have default arguments.

- Example

```
string::string(string&);         // copy ctor
string::string(const string&);   // copy ctor
string::string(string&,int=1);   // copy ctor
```

  Each of them can be used to pass a **string** object by value, e.g.

```
void p(string b) {}
string a("Pluto"); p(a);
```

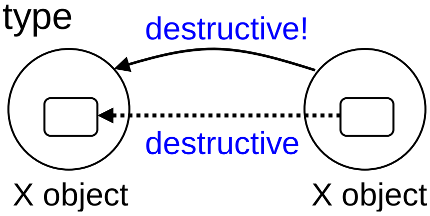
```
string::string(string&,int);     // ctor, but not copy ctor
string::string(string,int);      // ctor, but not copy ctor
```

  Each of them cannot be used to pass a **string** object by value, but can be used to construct a **string** object, e.g.

```
string b(a,3);
```

```
string::string(string);          // illegal
```

- If a class does not declare a copy ctor, one is declared implicitly. The implicit copy ctor for a class **`X`** is of type
  `X::X(const X&)`[*] or
  `X::X(X&)`[†]



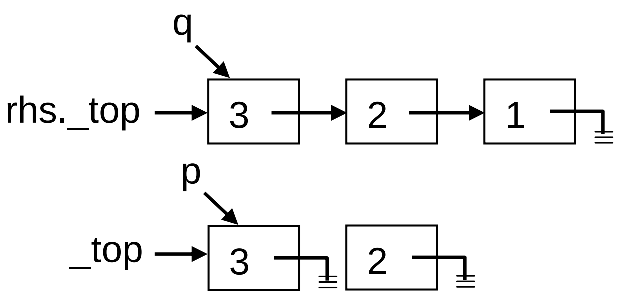
- Example – Stack

  Define a copy ctor for stacks represented by dynamic arrays (p9)

```
stack::stack(const stack& rhs)
: _top(rhs._top),stk(new int[80])
{
   for (int i=0;i<=rhs._top;i++)
      stk[i]=rhs.stk[i];
}
```

  Define a copy ctor for stacks represented by linked lists. The copy ctor below makes a deep copy of the source **`stack`** object.

```
stack::stack(const stack& rhs)
: _top(NULL)
{
   if (rhs._top!=NULL) {
      node* q=rhs._top;
      _top=new node(q->datum,NULL);
      node* p=_top;
      while ((q=q->succ)!=NULL) {
         p->succ=new node(q->datum,NULL);
         p=p->succ;
      }
   }
}
```



---

[*] If (1) each direct or virtual base class has a copy ctor whose first argument is cv-qualified, and (2) each nonstatic data member of class type has a copy ctor whose first argument is cv-qualified.

[†] Otherwise

- Example (Cont'd)

On the other hand, when stacks are represented by static arrays (p6), we may simply use the implicit copy ctor defined as

```
stack::stack(const stack& rhs)
: _top(rhs._top)
{
   for (int i=0;i<=rhs._top;++i)
      stk[i]=rhs.stk[i];
}
```

Note that it isn't defined as

```
stack::stack(const stack& rhs)
: _top(rhs._top),stk(rhs.stk)        // error
{}
```

as we cannot specify explicit initializer for arrays.

Member array initialization

Member arrays can only be default-initialized in ctor-initializers.

For example, the following ctor zero-initializes **stk[**i**]**, for all i.

```
stack::stack() : _top(-1),stk() {}
```

In general, if you want to copy-initialize arrays, you should use STL **vector** objects.

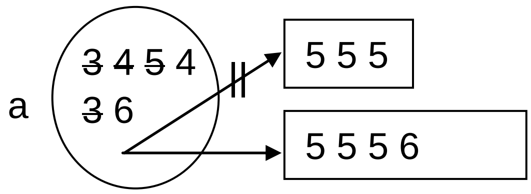
```
typedef int array[3];
array a={5,5,5};        // int a[3]={5,5,5};
array b(a);             // int b[3]=a; ✗
array &b(a);            // int (&b)[3]=a; ✓
int* b=a;               // ok

#include <vector>
vector<int> a(3,5);
vector<int> b(a);       // ok
```

- Example (Cont'd)

On vectors

vector objects and **string** objects have similar representations and copy ctors.
vector supports amortized constant time insert and erase at the end. Insert and erase in the middle take linear time.

```
vector<int> a(3,5);
a.push_back(6);
a.push_back(7);
a.pop_back();
```



Like the **string** class, the exact storage allocation strategy for the **vector** class is implementation-defined.

As another example, let's implement stacks by vectors

```
class stack {
public:
   void push(int n) { v.push_back(n); }
   void pop() { v.pop_back(); }
   int& top() { return v.back(); }
   const int& top() const { return v.back(); }
   bool empty() const { return v.empty(); }
private:
   vector<int> v;
};
```

Clearly, we needn't define ctor, dtor, and copy ctor for this **stack** class. The implicity generated ones will invoke those of the class **vector<int>**.

```
stack::stack()
: v()              // default-initialize v as an empty vector
{}
stack::stack(const stack& rhs)
: v(rhs.v)         // copy-initialize
{}
stack::~stack() {}
```

- Example – Queue

  Similarly, define a copy constructor when queues are represented by dynamic arrays (p24) or linked lists (p21).

  On the other hand, when queues are represented by two stacks (p40), we may simply use the implicit copy ctor defined as

```
queue::queue(const queue& rhs)
: _front(rhs._front),_back(rhs._back)
{}
```

- Example – Class template **pair** defined in **<utility>**

```
template<class T1,class T2>
struct pair {
   T1 first;      pair() {}
   T2 second;       ≠
   pair() : first(),second() {}
   pair(const T1& a,const T2& b)
   : first(a),second(b) {}
};
```

// Implicitly generated copy ctor

```
template<typename T1,typename T2>
pair<T1,T2>::pair(const pair<T1,T2>& p)
: first(p.first),second(p.second)
{}
```

```
pair<int,int> x(2,3);
pair<int,int> y(x);
```

This calls the implicitly generated copy ctor with **T1 = T2 = int**.

```
pair<long,long> z(x);
```

This is desirable, but cannot be done by the implicitly generated copy ctor, for the lack of the conversion

pair<int,int> → **pair<long,long>**

- Example (Cont'd)

To this end, the STL **pair** has a converting ctor:

```
template<typename T1,typename T2>
template<typename U1,typename U2>
pair<T1,T2>::pair(const pair<U1,U2>& p)
: first(p.first),second(p.second)
{}
```

T1＝T2＝long **U1＝U2＝int**

```
pair<long,long> z(x);
```

Observe that this converting ctor effects the conversion

pair<U1,U2> → **pair<T1,T2>**

provided that the conversion **U1** → **T1** and **U2** → **T2** exist.

Q: Which ctor is selected for the initialization in the starred line?

```
pair<int,int> x(2,3);
pair<int,int> y(x);      //*
```

A: The copy ctor is selected, because it is more specialized than the converting ctor.

Note on class template **X**

Within the class scope
1 Use **X<***template-parameters***>** for class type
2 Use **X** for class name, e.g. ctor, dtor

```
template<typename T1,typename T2>
pair<T1,T2>::pair(const pair<T1,T2>&)
```

Outside class scope
1 Always use **X<***template-parameters***>**

```
int main()
{
    pair<int,int> a(2,3);               // type
    cout << pair<int,int>(2,3).first;   // ctor call
}
```

create an anonymous temporary object

Initialization

- Direct initialization

  `X a(b);`

  1 new — `new X(b)`
  2 cast — `X(b)`, `(X)b`, `static_cast<X>(b)`
  3 member initializer — `T::T() : a(b){}`

  Copy initialization

  `X a=b;` — Arrays can only be copy-initialized.

  1 parameter passing
  2 function return
  3 throwing/handling exception
  4 `{}`_enclosed initializers — `X a[3]={b,b,b};`

- Direct initialization = copy initialization, If `X` is an arithmetic, enumeration, pointer, or reference type.

- Direct initialization
  Call the appropriate constructor, if possible

- Copy initialization where `b`'s type = `X`
  Call the appropriate copy constructor, if possible

- Copy initialization where `b`'s type ≠ `X`

  1 a temporary `X` object is created (may be eliminated by optimization)
  2 the initializer `b` is converted to the temporary `X` object by the appropriate user-defined conversion, if possible
  3 the object `a` being initialized is then direct-initialized from the temporary `X` object

- Notes on temporary objects

  Temporary objects are created in various contexts.
  C++ allows compilers to optimize temporary objects out of existence, if possible.
  However, the semantics must be respected as if the temporary objects were created.

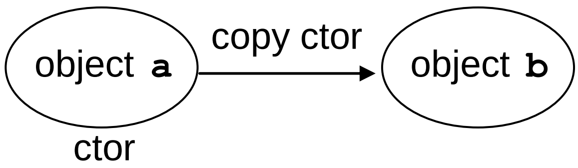
- Example

Recall the ctor and copy ctor of the **`string`** class

```
// const char* → string
string::string(const char*);
string::string(const string&);
```

Direct initialization

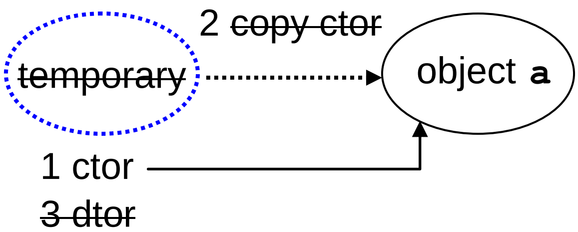
```
string a("pluto");
string b(a);
```



Copy initialization

```
string b=a;
```

```
string a="pluto";
```



1. **`"pluto"`** isn't a **`string`** object and so is implicitly converted to a temporary **`string`** object using the ctor, which is then direct-constructed to **`a`**.
   Making the conversion explicit obtains equivalent code:
   `string a=string("pluto");`

2. With return value optimization, the temporary is eliminated – the object is directly constructed in **`a`**.

3. Although the copy ctor and dtor weren't called, the semantics must be respected:
   a) since the temporary object is an rvalue, the copy ctor has to be of type

      `string::string(const string&)` [*]

   b) the copy ctor and dtor must be accessible

   N.B. VC++↓, GNU C++↑

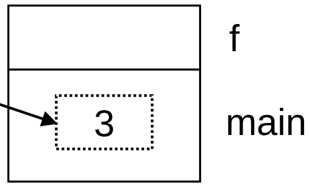
---

[*] Recall that references may refer to const rvalues.

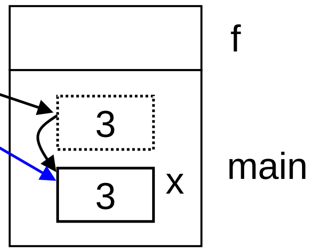
- Example (Cont'd)

On return value optimization

```
int f() { return 3; }
int main()
{
   cout << f();
}
```



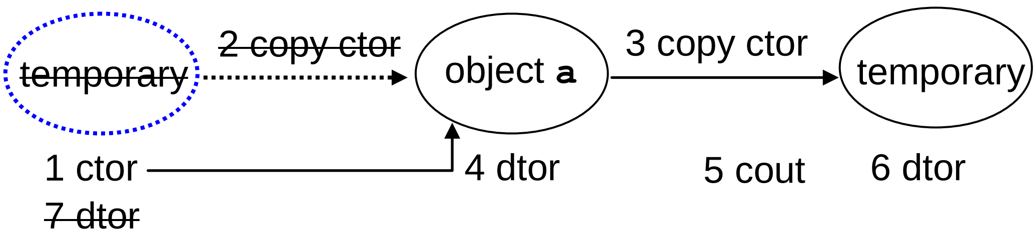

```
int f() { return 3; }
int main()
{
   int x=f();
}
```



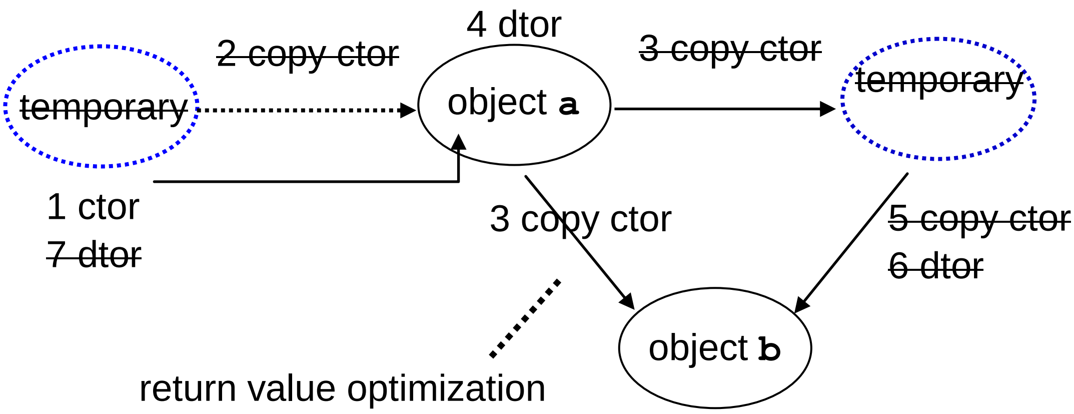
- Example

```
string f(string a) { return a; }
```

```
cout << f("pluto");
```




```
string b=f("pluto");    // or, string b(f("pluto"));
```



N.B. The temporaries are destroyed at the end of evaluating the expression in which they are created and in the reverse order of their construction.

- Example (Cont'd)

  Note that even if `f("pluto")` is an rvalue*, it can be modified:

  ```
  f("pluto")="snoopy";
  ```

  However, this is meaningless as it modifies a temporary object. "Return by const value" remedies this problem:

  ```
  const string f(string a) { return a; }
  ```

  N.B. `string("pluto")="snoopy";` can't be remedied.

  On modifiable expressions

  An rvalue expression of class type is modifiable by means of the class's member functions. There is nothing special. The call

  `f("pluto").`*mf*`(…)` where *mf* is a member function

  ought to be allowed. The assignment

  ```
  f("pluto")="snoopy";
  ```

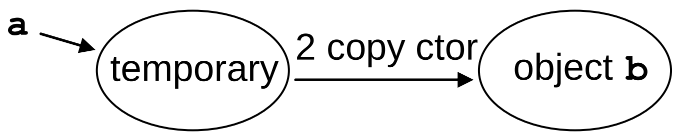
  is nothing but

  ```
  f("pluto").operator=("snoopy");
  ```

- Example

  Copy ctors are primarily used in call- and return-by-value.
  Call- and return-by-reference never use copy ctors.

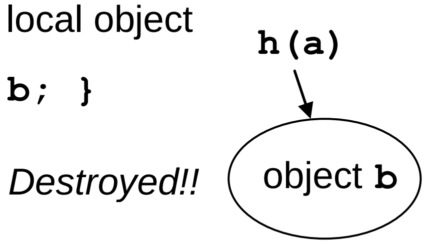
  ```
  const string& g(const string& a) { return a; }
  string b=g("snoopy")†;
  ```



1 ctor 3 dtor

  Warning: Never return a reference to a local object

  ```
  string& h(string b) { return b; }
  string a("snoopy");
  cout << h(a);
  ```



---

* The result of calling a function that doesn't return a reference is an rvalue

† References may refer to const objects (treated as rvalues) of different types, provided that the required conversions exist.

Explicit constructors

- An explicit constructor can only be invoked in direct initialization (especially, in casts).
- Example

```
class string {
public:
   explicit string(const char* ="");     // 2
   explicit string(const string&);       // 3
};
string f(string);
const string& g(const string&);
```

Comments

1. The specifier **`explicit`** can only appear in the declaration, but not in the definition.

2. Explicit constructors will not be used for implicit conversions.

	explicit	non-explicit
`string a("snoopy");`	✓	✓
`string a="snoopy";`	×	✓
`f("snoopy")`	×	✓
`g("snoopy")`	×	✓

   Instead, explicit conversions are needed, e.g.

```
f(string("snoopy"))
f((string)"snoopy")
f(static_cast<string>("snoopy"))
```

3. An explicit copy ctor prevents objects from being passed to or returned from functions.
   Thus, the function **`f`** is ill-formed due to the explicit copy ctor.

	explicit	non-explicit
`string b(a);`	✓	✓
`string b=a;`	×	✓
`g(a)` (don't care)	✓	✓

- Explicit constructors with multiple arguments are effectless, since they cannot take part in implicit conversions.
For example, the explicit declaration below is redundant.

// construct a **string** object with **n** copies of **s**
```
explicit string(const char* s,int n);
```

```
string a("snoopy",3);       // ok
string a=("snoopy",3);      // implicit conversion?
```

Note that the explicit declaration below is fine, since the ctor can be called with one argument.

```
explicit string(const char*,int=1);
string a("snoopy");         // ok
string a="snoopy";          // error
```

- Principle

Make the ctor
X::X(T)
explicit when **T** and **X** are unrelated (so that the conversion
T → **X**
is unnatural).

Example

```
string::string(const char*);
```

This constructor ought to be non-explicit, allowing the reasonable implicit conversion **const char*** → **string** so that the function

```
void p(const string&);
```

can be invoked by the call

```
p("snoopy");
```

```
stack::stack(int n) : _top(-1),stk(new int[n]) {}
```

This ctor ought to be explicit, disallowing the unreasonable implicit conversion **int** → **stack** so that the function

```
void p(const stack&);
```

can't be invoked by the call

```
p(7);         // had better enforce the call p(stack(7))
```

User-defined conversions

- User-defined conversions

  1 Converting ctor

  `X::X(T)` // **`T`** → **`X`**

  A converting ctor is a ctor declared without the function specifier **`explicit`** that can be called with a single parameter.

  2 Type conversion operator (conversion function)

  `X::operator T()` // **`X`** → **`T`**

  a) **`T`** can't be (cv-qualified) **`X`**, **`X&`**, **`void`**, array/function type
  b) Neither parameter types nor return type can be specified.

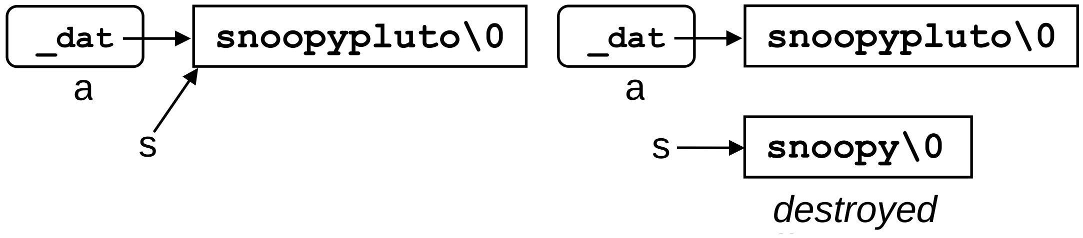
- Example

```
string::string(const char*);
string::operator const char*() const;  // hypothetical

string a("snoopy");
const char* s=a;     // static_cast<const char*>(a)
                     // a.operator const char*()
```

Version A – Sharing

```
string::operator const char*() const;
{
   return _data;
}
a+="pluto";
cout << s;           // snoopypluto or dangling pointer
```

Case1: enough storage Case 2: shy of storage



- Example (Cont'd)

  Version B – Copying

```
string::operator const char*() const;
{
   return strcpy(new char[_size+1],_data)
}
a+="pluto";
cout << s;       // ok, snoopy
delete [] s;
```

  The client has to deallocate the storage allocated by the server!

  Remark

  The conversion **string** → **const char*** isn't as natural as the conversion **const char*** → **string**. Thus, the **string** class doesn't support the implicit conversion; instead, it provides the explicit conversion by **c_str()**.

  c_str() must be used with care – the returned pointer is invalid if the related string object is modified subsequently.

```
const char* string::c_str() const
{
   return _data;
}
string a("snoopy");
const char* s=a.c_str();
a+="pluto";
cout << s;       // undefined
```

- Principle

  1) Implicit **X** → **T** conversion is desired
     Non-explicit **T::T(X);**
     X::operator T();

  2) Explicit **X** → **T** conversion is desired
     Explicit **T::T(X);**
     X::toT();
     where **toT** is a member function performing the conversion.

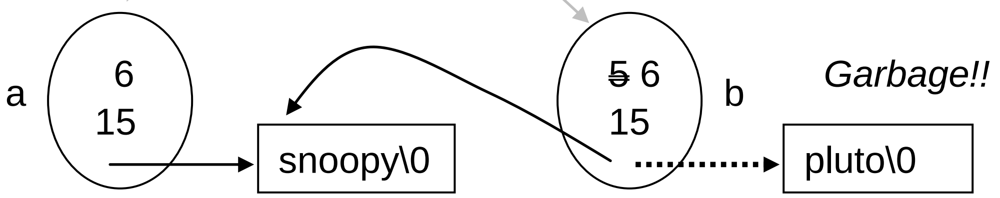
Copy assignment operator

- A copy assignment operator is used to assign one object to another object of the same class.

- Example

  Assume that the **`string`** class adopts the implicit copy assignment operator that does memberwise assignment.

```
string& string::operator=(const string& rhs)
{
   if (this!=&rhs) {      // self-assignment, for efficiency
      _size=rhs._size; _capacity=rhs._capacity;
      _data=rhs._data;
   }
   return *this;
}
string a("snoopy");
string b("pluto");
b=a;                        // b.operator=(a);
```



- Principle

  Define a copy assignment operator for classes with dynamically allocated memory.

```
string& string::operator=(const string& rhs)
{
   if (this!=&rhs) {      // self-assignment, for correctness
      delete [] _data;
      _size=rhs._size; _capacity=rhs._capacity;
      _data=strcpy(new char[_capacity+1],rhs._data);
   }
   return *this;
}
```

- Self-assignment

  The statement **`a = b;`** is a self-assignment if objects **`a`** and **`b`** are "the same".

  Object identity (Address equality)

  1. Two objects occupy the same memory location
  2. Check by **`&a==&b`**
     or **`this==&rhs`** inside a copy assignment operator
  3. Class independent, easy and efficiency

  Object equality (Value equality)

  1. Two objects have the same content
  2. Check by **`a==b`**
     or **`*this==rhs`** inside a copy assignment operator
  3. Class dependent, probably hard and inefficiency

  Object identity $\Rightarrow$ object equality, but not *vice versa*

- Example

```
string a("snoopy");      // a=b; a=*c;
string& b=a;             // a=d;
string* c=&a;            // object identity  a, b, *c
string d("snoopy");      // object equality  a, b, *c, d
d.reserve(31);
a==d                     // object equality?
a=="snoopy"              // object equality?
```

  Note: The **`string`** class defines the last two cases to be equal.

- On the parameter type of copy assignment operator

```
string& string::operator=(const string&);
```

  Q: Why is the object passed by reference, rather than by value?
  A: First of all, it can be passed by value.
     However, passing by value has to invoke the copy ctor.
     Moreover, it requires object equality be used for detecting self assignment.

- On the parameter type of copy assignment operator (Cont'd)

  Q: Why does it refer to a const object?
  A: Our semantics isn't destructive.
  If destructive copy assignment is desired, we may write

```
string& string::operator=(string& rhs)
{
   if (this!=&rhs) {     // self-assignment, for correctness
      delete [] _data;
      _size=rhs._size; _capacity=rhs._capacity;
      _data=rhs._data;
      rhs._size=rhs._capacity=0; rhs._data=NULL
   }
   return *this;
}
```

- On the return type of copy assignment operator

```
string& string::operator=(const string&);
```

  Q: Why does it return by reference, rather than by value?
  A: To avoid invoking the copy ctor.
  Returning by value is semantically incorrect in **`(a=b)=c`**.

  Q: Why doesn't it return a const object?
  A: To allow **`(a=b)=c`**
  so that the semantics is consistent with built-in types.

  Q: Why is the return type not **`void`**?
  Why does it return **`*this`**, rather than **`rhs`**?
  A: Again, to make the semantics consistent with built-in types.

- A copy assignment operator **`X::operator=`** is a member function with exactly one parameter of type **`X`** or (possibly cv-qualified) **`X&`**

- If a class does not declare a copy assignment operator, one is declared implicitly.
  The implicit copy assignment operator for a class **`X`** is of type
  `X& X::operator=(const X&)`
  or
  `X& X::operator=(X&)`

- Example – Overloading operator=

```
string& string::operator=(const char* rhs)
{
   if (_data!=rhs) {
      delete [] _data; _size=strlen(rhs);
      _capacity=15;
      while (_capacity<_size) (_capacity<<=1)++;
      _data=strcpy(new char[_capacity+1],rhs);
   }
   return *this;
}
```

This isn't a copy assignment operator – if it is absent, none will be declared. Without it, the starred line below can still be done by the copy assignment operator, except that an extra temporary object will be created.

```
a="pluto";                    // *
a=a.c_str();                  // self-assignment
```

- Example – Stack and Queue

For stack and queue, the copy assignment operator is hardly useful. To avoid unintentional use, one should

1 declare it as private

   Thus, the compiler won't generate one. And, if a client of the class attempts to use it, a compile error will occur.

2 provide no implementation

   Thus, if a friend of the class or the class itself attempts to use it, a linking error will occur.

```
class X {
public:
   void p() { X a,b; a=b; }        // linking error
private:
   X& operator=(const X&);
};
void q() { X a,b; a=b; }           // compile error
```

Copy construction and copy assignment

- Both copy construction and copy assignment have to copy the source object to the destination. Often, they have the same copying process. Thus, we may define a common private member function for the copying process to share code between them.

- Example

```
string::string(const string& rhs) { copy(rhs); }

string& string::operator=(const string& rhs)
{
   if (this!=&rhs) {
      delete [] _data;
      copy(rhs);
   }
   return *this;
}
```

where **copy** is a private member function defined as

```
void string::copy(const string& rhs)
{
   _size=rhs._size;
   _capacity=rhs._capacity;
   _data=strcpy(new char[_capacity+1],rhs._data);
}
```

A less efficient method is to implement copy construction in terms of copy assignment.

```
string::string(const string& rhs)
:  _data()                  // default construction   (weakest)
{
   *this=rhs;               // copy assignment
}                           // operator=(rhs);
                            // this->operator=(rhs);
```

Clearly, this is inefficient and arguable, for one-step construction is replaced by default construction plus copy assignment.

- Example (Cont'd)

  In general, albeit inefficient and arguable, one may write

```
T::T(const T& rhs)
: ...                    // default construction
{
   *this=rhs;            // copy assignment
}
```

Reversely, one might think of implementing copy assignment in terms of copy construction.

```
string& string::operator=(const string& rhs)
{
   if (this!=&rhs) {
      this->~string();             // destruction
      new (this) string(rhs);      // reconstruction
   }
   return *this;
}
```

Don't do this. In general, one shouldn't write

```
T& T::operator=(const T& rhs)
{
   if (this!=&rhs) {
      this->~T();                  // destruction
      new (this) T(rhs);           // reconstruction
   }
   return *this;
}
```

This idiom has lots of problems. To mention only one – it changes normal object lifetime. This will cause errors if the constructor or destructor has *side effects*.

In comparison, observe that the previous idiom doesn't change object's lifetime.

- Example – Heap array initialization

```
template<typename T>
class vector {
public:
   typedef size_t sise_type;
   vector();
   explicit vector(size_type,const T& =T());
   ~vector();
private:
   size_type _size,_capacity;
   T* _data;
};

template<typename T>
vector<T>::vector()
: _size(0),_capacity(0),_data(NULL)
{}

template<typename T>
vector<T>::~vector()
{
   for (int i=n-1;i>=0;i--) _data[i].~T();
   operator delete[](_data);
}
```

Heap array initialization

// Version A – the solution

```
template<typename T>
vector<T>::vector(size_type n,const T& val)
:  _size(n),
   _capacity(n)
   _data((T*)operator new[](n*sizeof(T)))
{
   for (int i=0;i<n;i++)
      new (_data+i) T(val);   // copy construction
}
```

- Example (Cont'd)

```
// Version B – inefficient and arguable.
template<typename T>
vector<T>::vector(size_type n,const T& val)
:  _size(n),_capacity(n),
   _data(new T[n])           // default construction
{
   for (int i=0;i<n;i++)
      _data[i]=val           // copy assignment
}
```

```
// Version C – Warning
template<typename T>
vector<T>::vector(size_type n,const T& val)
:  _size(n),_capacity(n),
   _data(new T[n])           // default construction
{
   for (int i=0;i<n;i++)
      new (_data+i) T(val);  // copy construction
}
```

This version initializes each **`T`** object twice. In general, it will not crash, but may result in memory leak, e.g.
`vector<string> v(7,"Snoopy");`

```
// Version D – Incorrect
template<typename T>
vector<T>::vector(size_type n,const T& val)
:  _size(n),_capacity(n),
   _data((T*)operator new[](n*sizeof(T)))
{
   for (int i=0;i<n;i++)
      _data[i]=val           // copy assignment
}
```

This version modifies uninitialized **`T`** objects. It will often crash on attempting to delete nonexistent old data, e.g.
`vector<string> v(7,"Snoopy");`

Operator overloading

- An operator function shall either be
  1) a non-static member function, or be
  2) a non-member function having at least one parameter whose type is (a reference to) a class or an enumeration.
- operator**=**, operator**()**, operator**[]**, and operator**->** must be non-static member functions.
- The precedence, associativity, and arity of an operator cannot be altered.
- **:: ?: . .*** cannot be overloaded
- Example – As member functions

  Operator+= is often a member function and similar in signature to operator=.

```
string& string::operator+=(const string& rhs)
{
   size_type new_size=_size+rhs._size;
   if (_capacity<new_size) {
      while (_capacity<new_size) (_capacity<<=1)++;
      char* old_data=_data;
      _data=strcpy(new char[_capacity+1],old_data);
      delete [] old_data;
   }
// if (this!=&rhs) strcat(_data,rhs._data); else
   {
   for (size_type i=0;i<rhs._size;i++)
      _data[i+_size]=rhs._data[i];
   _data[new_size]='\0';
   }
   _size=new_size;      // *
   return *this;
}
```

  Adjusting the size in the starred line is more than conceptually correct – it cannot be done too earlier for self-appending to work.

- Example (Cont'd)

  ```
  string& string::operator+=(const char*);
  ```

  This can be defined in a like manner.

- Example – As non-member functions

  operator+ as a member function

  ```
  string string::operator+(const string& rhs) const
  {
     string s(*this); s+=rhs; return s;
  }
  string a("snoopy"),b("pluto");
  cout << a+b;
  cout << a+"pluto";              ① ✓
  cout << "snoopy"+b;             ② ×
  ```

  ① **a.operator+("pluto")**
  Ok, **"pluto"** is converted to a temporary **string** object.

  ② **"snoopy".operator+(b)**
  No, **"snoopy"** won't be converted to a **string** object.

  Because operator+ is supposed to be commutative, it should not be a member function in this case.

  Remark: In other cases, operator+ may be a member function, eg

  ```
  class matrix {
  public:
     matrix(int m,int n);
     matrix operator+(const matrix&) const;
  };
  ```

  operator+ as a non-member function   // **a+b** ≡ **operator+(a,b)**

  ```
  string operator+(const string& lhs,const string& rhs)
  {
     string s(lhs); s+=rhs; return s;
  }

  string operator+(const string&,const char*);
  string operator+(const char*,const string&);
  ```

- Example (Cont'd)

```
string operator+(const string&,const string&); ①
string operator+(const string&,const char*);   ②
string operator+(const char*,const string&);   ③
```

```
string a("snoopy"),b("pluto");
string c=a+b;
```

① identity/identity (best viable)
② not a viable function
③ not a viable function

```
string c=a+"pluto";
```

① identity/user-defined conversion
② identity/array-to-pointer (best viable)
③ not a viable function

```
string c="snoopy"+b;
```

Similarly, this invokes ③

```
string c="snoopy"+"pluto";
```

Error! If no operand of an operator has (a reference to) a class or enumeration type, the operator is assumed to be built-in.

```
string c=operator+("snoopy","pluto");
```

① user-defined conversion/user-defined conversion
② user-defined conversion/array-to-pointer
③ array-to-pointer/user-defined conversion

Ambiguous! ② and ③ are better than ①, but they are no better than each other.

Q: Why not return a reference to an object?
A: You cannot.
Remember – Never return a reference to a local object

Q: Why not return a const object?
A: It should be, but the standard `string` class fails to recognize this. Returning a non-const object allows this strange code
`a+b=a;`
which is inconsistent with built-in types.

- Example – As non-member functions

operator<< and operator>> as member functions

```
ostream& string::operator<<(ostream& os) const
{
   return os << _data;
}
istream& string::operator>>(istream& is)
{
   char buf[255];
   is >> buf;
   *this=buf;              // operator=(buf);
   return is;
}
string a;
a >> cin;                  // a.operator>>(cin);
a << cout;                 // a.operator<<(cout);
```

Because **cin** and **cout** used to be the left operand, operator<< and operator>> should not be member functions.

operator<< and operator>> as non-member functions

```
ostream& operator<<(ostream& os,const string& s)
{
   return os << s.c_str();
}
istream& operator>>(istream& is,string& s)
{
   char buf[255];
   is >> buf;
   s=buf;
   return is;
}
cin >> a;                  // operator>>(cin,a);
cout << a;                 // operator<<(cout,a);
```

Friend

- A friend of a class is a function or class that isn't a member of the class but is permitted to use its private and protected members.
- A friend declaration may appear anywhere in a class.
- Example

  operator<< as a non-friend of class **stack**

```
ostream& operator<<(ostream& os,const stack& s)
{
   stack t(s);
   while (!t.empty()) {
      os << t.top(); t.pop();
   }
   return os;
}
```

  operator<< as a friend of class **stack** (Linked list representation)

```
class stack {
friend ostream& operator<<(ostream&,const stack&);
...
};
```

```
ostream& operator<<(ostream& os,const stack& s)
{
   stack::node* p=s._top; // qualified, not in class scope
   while (p!=NULL) {
      os << p->datum; p=p->succ;
   }
   return os;
}
```

- Principle

  1 Don't have too many friends.
  2 Be a friend only when it can improve inefficiency or save code

- Example

  Stack/Queue implementation (Revisited)

  Version 1 – For illustration purpose only

```
class stack {
public: ...
private:
   class node {
      friend class stack;
      node(int,node*);
      int datum;
      node* succ;
   };
   node *_top;
};

class queue {
public: ...
private:
   class node {
      friend class queue;
      node(int,node*);
      int datum;
      node* succ;
   };
   node *_front,*_back;
};
```

  Version 2

```
class node {
   friend class stack;       // friend saves code
   friend class queue;
   node(int,node*);          // private ctor
   int datum;
   node* succ;
};
```

- Example (Cont'd)

```
class stack {
public: …
private:
   node* _top;
};
```

```
class queue {
public: …
private:
   node *_front,*_back;
};
```

Version 3

```
class node;                    // class declaration

class stack {
public:
   stack() : _top(NULL) {}
   ~stack() { while (!empty()) pop(); }
   void push(int);
   void pop();
   int& top()
   const int& top() const;
   bool empty() const { return _top==NULL; }
private:
   node* _top;
};

class node {                   // class definition
   friend void stack::push(int);
   friend void stack::pop();
   friend int& stack::top();
   friend const int& stack::top() const;
   node(int d,node* s) : datum(d), succ(s) {}
   int datum;
   node* succ;
};
```

- Example (Cont'd)

```
void stack::push(int n)
{  _top=new node(n,_top); }

void stack::pop()
{  node* p=_top; _top=_top->succ; delete p; }

int& stack::top() { return _top->datum; }

const int& stack::top() const
{  return _top->datum; }
```

- A friend function can be defined inside a class if the class is non-local and the function name is unqualified.

  A friend function defined inside a class is in the scope of the class in which it is defined.

```
class X {
   friend void p() {}
public:
   static void p() {}
};
int main()
{
   p();          // call friend
   X::p();       // call static member
}
```

Exception handling

Motivation

- An exception is an abnormal situation such as index out of range, stack overflow, divide by zero, etc, that occurs during program execution.
- A language with exception handling mechanism provides a way to raise and handle exceptions.
- Exception handling supports a clean separation between the normal execution code and the exception handling code.
- Example

  Multiply a sequence of numbers subject to the requirements:
  1) If there is a zero in the sequence, don't do any multiplication
  2) the sequence can only be scanned at most once

  Version 0 – Fail to meet the requirements

```
int mul(int* begin,int* end)
{
   if (begin==end) return 1;
   else if (*begin==0) return 0;
   else return *begin*mul(begin+1,end);
}
```

  Version 1 – Without exception handling

```
int mul(int* begin,int* end)
{
   if (begin==end) return 1;
   else if (*begin==0) return 0;
   else {
      int r=mul(begin+1,end);
      return r==0? r: r**begin;
   }
}
```

  Drawback: The code is somewhat messy.

- Example (Cont'd)

Version 2 – With exception handling

```
int mul(int* begin,int* end)
{
   if (begin==end) return 1;
   else if (*begin==0) throw 0;  // raise an exception
   else return *begin*mul(begin+1,end);
}
int main()
{
   int a[7]={4,2,3,0,1,7,5};
   try {                         // try block
      cout << mul(a,a+7);
   }
   catch (int r) {               // exception handler
      cout << r;
   }
}
```

Remarks

1 A throw-expression (of type **void**) raises an exception.
2 A try-block is a statement that provides one or more exception handlers.
3 This idiom of exception handling allows one to easily escape from deep recursion.

- Example

```
#include <stdexcept>
string::reference string::at(size_type pos)
{
   if (pos<size())
      return operator[](pos);
   else
      throw out_of_range("invalid string position");
}
```

- Example (Cont'd)

```
string::const_reference
string::at(size_type pos) const;
```

This is defined in exactly the same way.

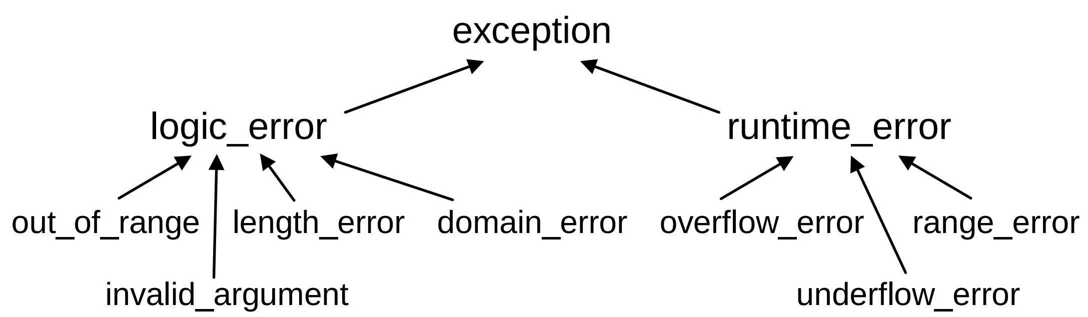
```
int main()
{
   string a("snoopy");
   try { cout << a.at(9); }
   catch (out_of_range& expt) {
      cout << expt.what();
   }
}
```

cf.

```
int main()
{
   string a("snoopy");
   cout << a[9];              // undefined behavior
}
```

- Exception hierarchy

C++ library provides a hierarchy of exception classes that defines an "isa" relationship among the classes. The major portion of the hierarchy is depicted below.



Each class **`T`**, except for **`exception`**, in the hierarchy has a ctor **`T::T(const string&)`**.

Each class **`T`** offers a member function
`const char* what();`
to retrieve the message carried on a **`T`** object.

Throwing and handling exceptions

- Throwing/handling an exception involves copy-initialization[*].
- Throwing an exception
  1. A throw-expression copy-initializes a temporary object, called an exception object, which is allocated in an unspecified way.
  2. The exception object persists as long as there is a handler being executed for that exception.
     In particular, if the handler being executed rethrows the exception, the exception object remains. (See next point)
  3. A throw-expression with no operand rethrows the exception being handled without copying it.
  4. When an exception is thrown, the control is transferred to the nearest handler with a matching type.
- Handling an exception
  1. The handlers of a try block are tried in order of appearance. The search for a matching handler continues in a dynamically surrounding try block.
  2. If the handler has a named (or unnamed) by-value parameter, the named (or, a temporary, which may be eliminated) object is copy-constructed with the exception object.
  3. If the handler has a by-reference parameter, it refers directly to the exception object.
  4. A ... handler is a catch-all handler.
     If present, it shall be the last handler for its try block.
  5. An uncaught exception calls **`terminate()`**, which by default calls **`abort()`**.

```
void terminate();     // declared in <exception>
void abort();         // declared in <ctdlib>
```

[*] The discussion in this section applies equally well to class objects or values of built-in types. But, of course, no (copy) ctors will be invoked for built-in types.

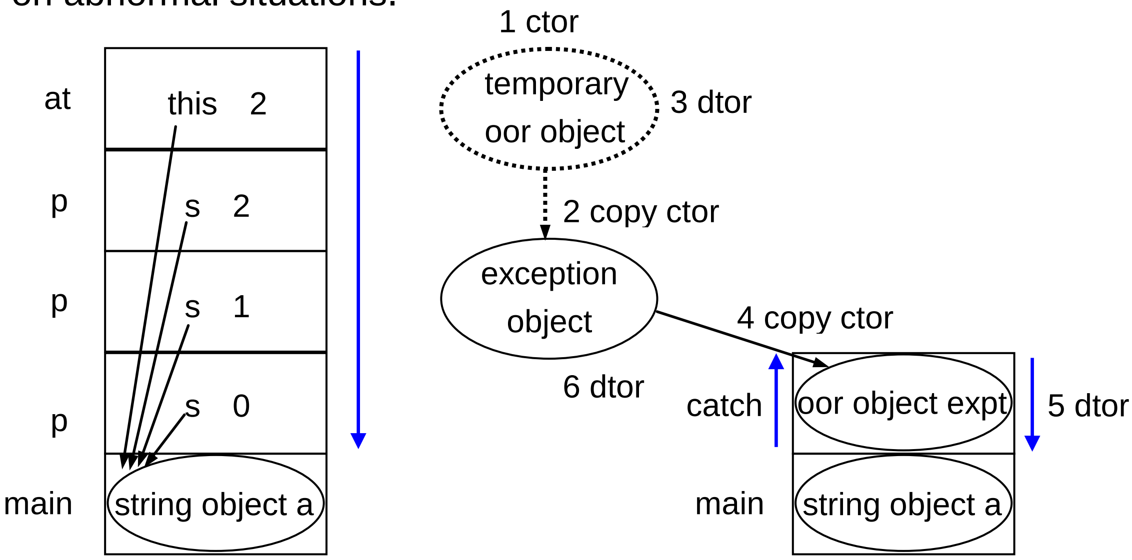
- Example

```
void p(const string& s,string::size_type pos)
{
   if (pos>=s.size()) { cout << endl; return; }
   cout << s.at(pos); p(s,pos+1);
}
int main() { string a("hi"); p(a,0); }
```

Rewrite it by removing the termination condition:

```
#include <stdexcept>
void p(const string& s,string::size_type pos)
{
   cout << s.at(pos); p(s,pos+1);
}
int main()
{
   string a("hi");
   try { p(a,0); }
   catch (out_of_range expt) {      // catch by value
      cout  << endl;
   }
}
```

Caution – This example is for illustration purpose only. Exception handling is quite expensive and should be used only on abnormal situations.

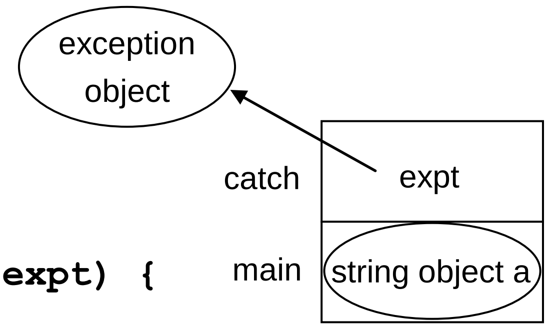


- Example (Cont'd)

  1 The temporary oor object is constructed by the call
  `out_of_range("invalid string position")`
  which may be optimized out of existence
  2 Throw an exception
  4 Handling an exception
  Note: The parameter **`expt`** isn't used inside the catcher and so may be removed. If so, a temporary object may or may not be generated.
  6 The exception object ceases to exist at this point.

Prefer catch-by-reference to catch-by-value

```
int main()
{
   string a("hi");
   try {
      p(a,0);
   }
   catch (out_of_range& expt) {
      cout << endl;
   }
}
```



Note: If the parameter name **`expt`** is removed, there is clearly no reference to the exception object.

Another example

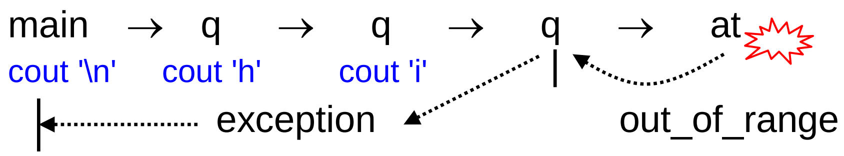
```
void q(const string& s,string::size_type pos)
{
   if (pos>=s.size()) return;
   cout << s.at(pos);
   q(s,pos+1);
   cout << s.at(pos);
}
int main()
{
   string a("hi"); q(a,0); cout << endl;
}
```

- Example (Cont'd)

  Again, we may rewrite it by removing the termination condition.

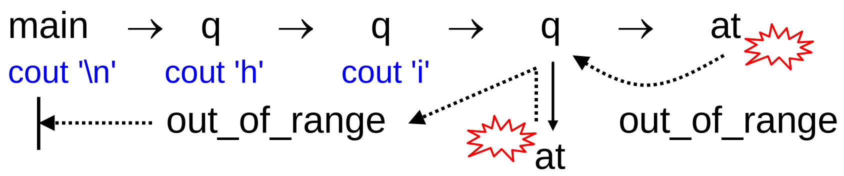
  Approah 1 – Each caller in the calling sequence responds partially to the exception

```
#include <stdexcept>
void q(const string& s,string::size_type pos)
{
   try {
      cout << s.at(pos); q(s,pos+1);
   }
①  catch (out_of_range&) { throw exception(); }
②  catch (exception&) { cout << s.at(pos); throw; }
}
int main()
{
   string a("hi");
   try { q(a,0); }
   catch (exception&) { cout << endl; }
}
```



Note that an exception is finished when the corresponding catcher exits.

① The order of these two catchers shall be left as it is, because an **`out_of_range`** is also an **`exception`**, but not *vice versa*. Were the order reversed, the **`out_of_range`** exception would be raised twice.



- Example (Cont'd)

② **`catch (exception& e) { … throw; }`**
`catch (exception& e) { … throw e; }`

The former rethrows the current exception, whereas the latter throws a new *copy* of the current exception (which is destroyed thereafter).

The former is better, because 1) it is more efficient, and 2) it doesn't change the type of the exception being propagated.

For instance, if the current exception referred to by **`e`** is of type **`out_of_range`**, the former will propagate that **`out_of_range`** exception, but the latter will propagate an **`exception`** exception.

Approah 2 – Only the last caller responds to the exception

```
void q(const string& s,string::size_type pos)
{
   try {
      cout << s.at(pos); q(s,pos+1);
      cout << s.at(pos);
   }
   catch (out_of_range&) {}
}
int main() { string a("hi"); q(a,0); cout << endl; }
```

main ⇄ q ⇄ q ⇄ q → at
cout '\n'   cout 'h'   cout 'i'

Alternative code

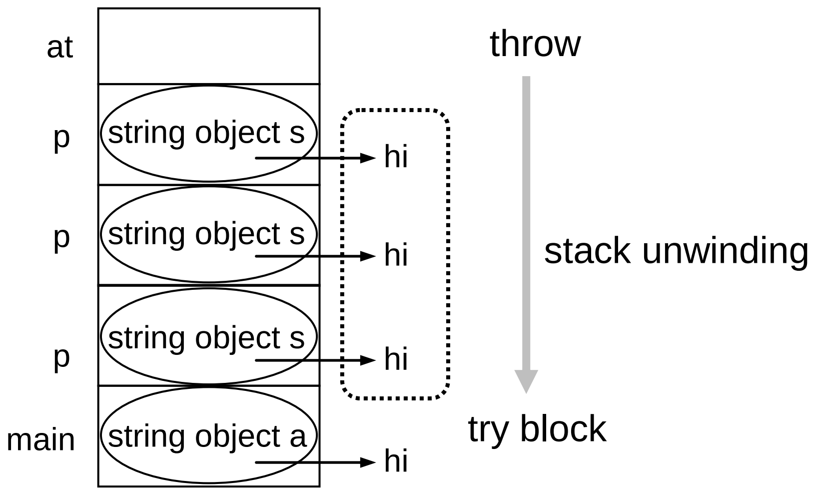
```
void q(const string& s,string::size_type pos)
{
   try {
      cout << s.at(pos); q(s,pos+1);
   }
   catch (out_of_range&) { return; }
   cout << s.at(pos);
}
```

- Stack unwinding

The process of calling dtors for class objects constructed on the path from a try block to a throw-expression.

Example

```
void p(string s,string::size_type pos)
{
   cout << s.at(pos); p(s,pos+1);
}
int main()
{
   string a("hi");
   try { p(a,0); } catch (out_of_range&) {}
}
```



If a destructor throws *out* an exception during stack unwinding, C++ calls **terminate()**, as it is facing a dilemma:

1) Should it ignore the original exception (and stop unwinding the stack)?
2) Should it ignore the current exception (and stop cleaning up what the dtor is supposed to clean) ?

This is no good answer – either choice loses information.

Note: An exception may still be thrown during the execution of a dtor as long as it doesn't propagate out of the dtor.

Principle
Keep exceptions from propagating out of destructors

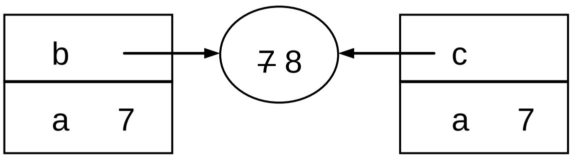
- Function invocations and exception handling

1. Callee look-up adopts static scoping.
   Catcher look-up adopts dynamic scoping.
   (This must be the case, for the callers should be responsible to the exception.)

2. The callee returns to the caller. (The call site is alive.)
   The catcher does not return to the thrower. (The throw site is dead.)

3. The caller's arguments are directly passed to the callee.
   The thrower's exception is copied, and it is the copy that is propagated to the catcher. (This must be the case, since the throw site is dead.)

```
int main()
{
   int a(7);
   try {
      try { throw a; }
      catch (int& b) { b++; throw; }
   }
   catch (int& c) { cout << c; }
   cout << a << endl;
}
```



   Note: Catch-by-reference isn't the same as call-by-reference.

4. The type match between caller and calle is loose – many type conversions are applicable.
   The type match between thrower and catcher is strict – only two kinds of type conversions are applicable:

   1 convert class **`A`** to class **`B`**, where **`A`** isa **`B`**
   2 convert a pointer to **`void*`** (possibly cv-qualified)

```
void p(float) {}        // callable for any type that can
                        // be converted to float
try {}
catch (float) {}        // catch float exception only
```

Function try block

- A function try block associates handlers with the ctor-initializer, if present, and the function body.
- Function try blocks support the cleanest separation between the normal execution code and the exception handling code.
- Example (See p82)

```
void q(const string& s,string::size_type pos)
try
{
   cout << s.at(pos); q(s,pos+1);
}
catch (out_of_range&) { throw exception(); }
catch (exception&) { cout << s.at(pos); throw; }

int main()
try
{
   string a("hi");
   q(a,0);
}
catch (exception&) { cout << endl; }
```

Note that the parameters of the function are visible in the catcher, but the local variables declared inside the function body aren't.

- Function try blocks are particularly useful with constructors.

Constructor failure

- When a constructor throws out an exception, the construction of the object fails and everything that has already been done must be undone. Thus, when a constructor fails,

  1 all of the fully constructed subobjects are destroyed
  2 the exception continues to propagate

  Furthermore, since the construction has failed, the object never existed, the destructor will never be called, as there is nothing to destroy.

- Example

```
class foo {
public:
   foo(const string&);
private:
   string s;
};

foo::foo(const string& s)
: s(s)                              // destroyed!
{
   throw "Bomb!";
}

int main()
try
{
   string s("Snoopy");              // destroyed! (stack unwinding)
   foo bar(s);                      // never call ~foo()
}
catch (const char* msg) { cout << msg; }
```

- Example

```
class foo {
public:
   foo(const string&);
   ~foo() { delete [] t; }
private:
   string s; char* t;
};
```

Incorrect version

```
foo::foo(const string& s)
: s(s),t(strcpy(new char[s.size()+1],s.c_str()))
{                                  // *t won't be freed
   throw "Bomb!";
}
```

Correct version

```
foo::foo(const string& s)
try
: s(s),t(strcpy(new char[s.size()+1],s.c_str())
{
   throw "Bomb!";
}
catch (const char*) { delete [] t; }

int main()                         // automatically rethrown
try
{
   string s("Snoopy");
   foo bar(s);
}
catch (const char* msg) { cout << msg; }
```

Semantics

1. Before entering the handler of a function try block of a ctor or dtor, the fully constructed subobjects are destroyed.
2. After executing a handler of a function try block of a ctor or dtor, the exception being handled is rethrown.

- Example (Cont'd)

```
int main()
try
{
   string s("Snoopy");
   foo* bar=new foo(s);     // storage freed
}
catch (const char* msg) { cout << msg; }
```

If the ctor was initiated by a non-placement new expression, the storage in which the object was being constructed is freed.

- Example (Cont'd; Advanced)

```
int main()
try
{
   string s("Snoopy");
   void* buf=operator new(sizeof(foo));
   foo* bar=new (buf) foo(s);     // storage not freed
}
catch (const char* msg) { cout << msg; }
```

If the ctor was initiated by a placement new expression, then

1 if the invoked placement operator new has a matching placement operator delete, call it
2 otherwise, do nothing

First of all, the language predefines a matching pair of placement operator new and placement operator delete.
They are said to be matched because all parameter types except the first are identical.

```
void* operator new(size_t,void* p)
{
   return p;
}

void operator delete(void*,void*) {}
```

- Example (Cont'd; Advanced)

  Next, recall that

  `new (buf) foo(s)`

  requests storage by the call

  `operate new(sizeof(foo),buf)`

  Since this placement operator new has a matching placement operator delete, the latter will be invoked on ctor failure as follows

  `operate delete(`*`buf`*`,buf)`

  where ***buf*** is the pointer returned from the earlier call

  `operate new(sizeof(foo),buf)`

  Remarks

  1 The placement operator delete is always passed the same arguments (except the first) as were passed to the placement operator new.
  The first argument passed is the returned value of the call to the placement operator new.

  2 Since the predefined placement operator delete has no idea what to do, it does nothing. Hence, the storage isn't freed.

  Put together, what we need is an overloaded pair of placement operator new and placement operator delete

```
void* operator new(size_t,foo* p)
{
   return p;
}
void operator delete(void* q,foo* p)
{
   operator delete(p);   // or, operator delete(q), for p=q
}
```

  and a forced call to our placement operator new

```
foo* bar=new ((foo*)buf) foo(s);
```

Exception specifications

- An exception specification specifies the exceptions that a function might throw out.

- A function with no exception specification allows all exceptions. A function with an empty exception specification, **`throw()`**, does not allow any exception.

- An unexpected exception calls **`unexpected()`**, which by default calls **`terminate()`**.
  `void unexpected();` // declared in **`<exception>`**

- Example

```
class foo {
public:
   foo(const string&) throw(const char*);
   ~foo() throw() { delete [] t; }
private:
   string s; char* t;
};
```

```
foo::foo(const string& s) throw(const char*)
try
: s(s),t(strcpy(new char[s.size()+1],s.c_str())
{
   throw "Bomb!";
}
catch (const char*) { delete [] t; }
```

Note that the same exception specification must appear in both the declaration and the definition.

Examples

- Example 1 – Balanced parentheses

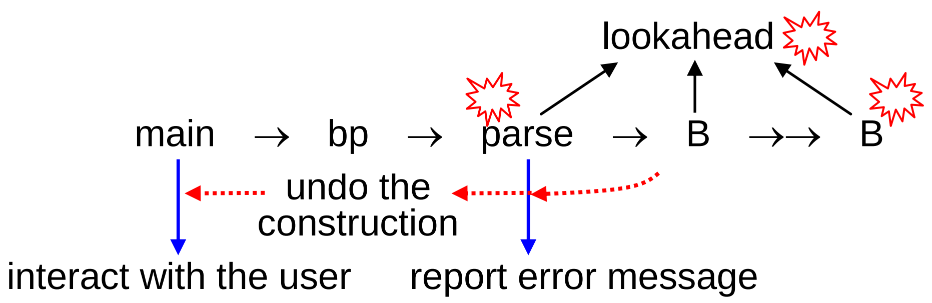
  Context-free grammar

  $B \to \varepsilon$
  $B \to (\ B\ )\ B$

  Attribute grammar

  $B \to \varepsilon \qquad B.pairs = 0$
  $B_1 \to (\ B_2\ )\ B_3 \qquad B_1.pairs = B_2.pairs + B_3.pairs + 1$

```
class bp {
public:
   bp(const string&) throw (bad_bp);
   int pairs() const { return _pairs; }
private:
   string paren;
   int _pairs;
   mutable string::size_type next;
   void parse() throw (bad_bp);
   void B(int&) throw (bad_bp);
   void lookahead() const throw (bad_bp);
};
```



- Example 1 (Cont'd)

```
int main()
{
   string s;
   cout << "Enter a string of parantheses: ";
   while (getline(cin,s)) {
      try {
         bp t(s);
         cout << t.pairs() << "-pair bp\n";
      }
      catch(...) { cout << "Try again!\n"; }
      cout << "Enter a string of parantheses: ";
   }
}

class bad_bp : public exception {  //* isa
public:
   bad_bp(const string& msg) : msg(msg) {}
   ~bad_bp() throw() {}
   const char* what() const throw()
   {
      return msg.c_str() ;
   }
private:
   string msg;
};
```

Remarks

1. The inheritance in the starred line may be removed without affecting the behavior of this program.
2. We have to define a dtor for **bad_bp**, because it must have the same exception specification as that of **exception** (the implicit generated dtor has no exception specification).

- Example 1 (Cont'd)

Recursive descent parser

```
bp::bp(const string& s) throw (bad_bp)
: paren(s)        // undo on pasring error
{
   parse();       // initialize _pairs on a successful parse
}

void bp::parse() throw (bad_bp)
try
{
   next=string::npos;        ①
   lookahead();
   B(_pairs);
   if (next<paren.size())   ②
      throw
      bad_bp(string("Extra char ")+paren[next]);
}
catch (bad_bp& e) {
   cout << e.what() << endl;
   throw;
}
```

① **string::npos** is a static data member defined by
const string::size_type string::npos=-1;
Note that the value of **npos** is indeed the maximum integer of the unsigned type **string::size_type**.

②

```
                 next                 next
                  ↓                    ↓
parens   |(())())\0| no       |(())()\0| ok
```

Don't write the condition as **paren[next]!='\0'**.
If **next==paren.size()**, **paren[next]** is undefined.
(Usually, it is the null character; but it is indeed undefined!)

However, if **paren** is a const string, **paren[next]** is defined to be the null character.

- Example 1 (Cont'd)

```
void bp::lookahead() const throw (bad_bp)   ②
{
   do
      next++;
   while (next<paren.size()&&paren[next]==' ');
   if (next<paren.size()&&                    ①
      string("()").find(paren[next])==string::npos)
   throw bad_bp(string("Illegal char  ")+paren[next]);
}
```

① **`find(`*c*`)`** returns the lowest position that contains character *c* if any; otherwise, returns **`npos`**.

	next ↓		next ↓		next ↓	
parens	`(()x()\0`	no	`(())()\0`	ok	`(())()\0`	ok

② Consider the three data members

		modified?		
	part of a **bp** object?	**parse**	**B**	**lookahead**
paren	O	X	X	X
_pairs	O	O	O	X
next	X	X	X	O

`next` is just an indexing variable and conceptually not a part of a **`bp`** object.
`lookahead()` is declared const, because it doesn't modify **`paren`** and **`_pairs`**.
Declaring **`next`** as mutable allows **`lookahead()`** to modify it

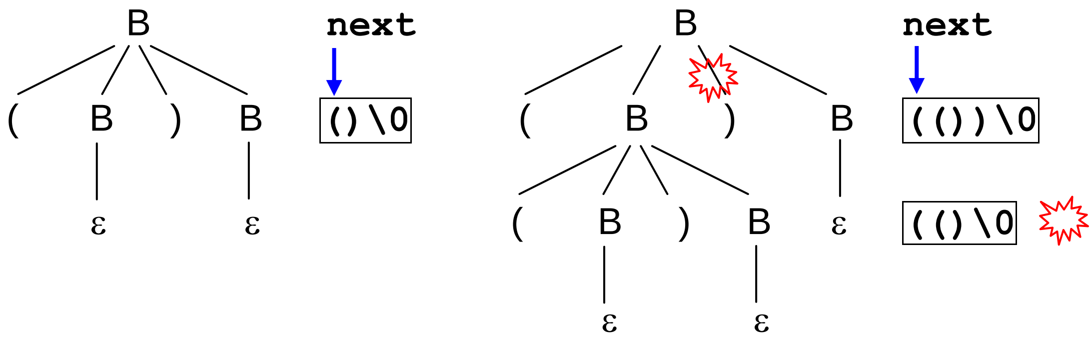
Note:
Mutable data members are modifiable inside const member functions.

- Example 1 (Cont'd)

$B \to \varepsilon$ $\qquad B.pairs = 0$

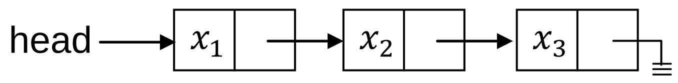
$B_1 \to (\, B_2 \,)\, B_3$ $\qquad B_1.pairs = B_2.pairs + B_3.pairs + 1$

```
void bp::B(int& pairs) throw (bad_bp)
{
   if (next<paren.size()&&paren[next]=='(') {
      lookahead();
      B(pairs);
      if (next<paren.size()&&paren[next]==')') {
         lookahead();
         int pairs2; B(pairs2);
         pairs+=pairs2+1;
      } else
         throw bad_bp(") expected at the end");
   } else
      pairs=0;
}
```



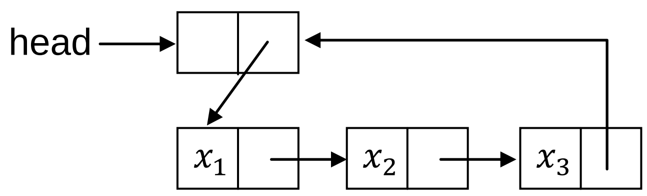
- Example 2 – Singly linked list with a header node

Singly linked list with no header node



Disadvantage **–** Have to distinguish
1) between the deletion of node x_1 and other node, and
2) between the insertion before and after node x_1.

Singly linked list with a header node

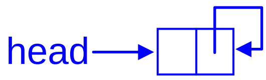


Advantage **–** Once established, the head pointer always points to the header node.

```
class list {
public:
// types
   class iterator;
   class const_iterator;
// ctor/dtor/copy assignment
   list();
   list(const list&);
   list& operator=(const list&);
   ~list();
// modifiers
   iterator insert(iterator,int);
   iterator erase(iterator);
   void push_front(int);
   void push_back(int);
   void pop_front();
   void pop_back();
```

- Example 2 (Cont'd)

```
// iterators
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
// capacity
    bool empty() const;
private:
    struct node;
    node* head;
};

struct list::node {
    node(int d,node* s) : datum(d),succ(s) {}
    int datum;
    node* succ;
};
```




```
// ctor
list::list()
: head((node*)operator new(sizeof(node)))
{
    head->succ=head;
}
```

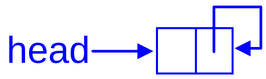
Since the header node contains no datum, it shall not be created by **new** operator.

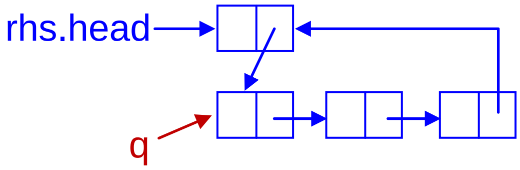
```
// dtor
list::~list()
{
    while (!empty()) pop_front();
    operator delete(head);
}
```

- Example 2 (Cont'd)

// copy ctor

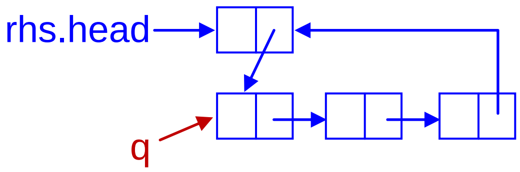
```
list::list(const list& rhs)
: head((node*)operator new(sizeof(node)))
{
   head->succ=head;
   node* q=rhs.head->succ;
   while (q!=rhs.head) {
      push_back(q->datum); q=q->succ;
   }
}
```

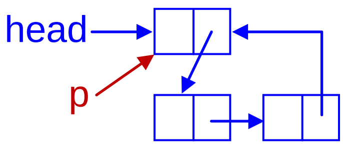


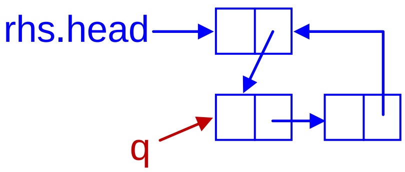


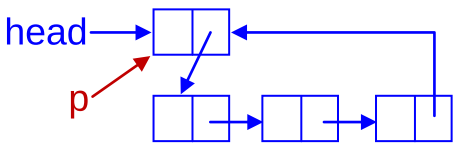
// copy assignment operator

```
list& list::operator=(const list& rhs)
{
   if (this!=&rhs) {
      node *p=head,*q=rhs.head->succ;
      while (p->succ!=head&&q!=rhs.head) {
         p->succ->datum=q->datum;
         p=p->succ; q=q->succ;
      }
      if (p->succ==head)
         while (q!=rhs.head) {
            push_back(q->datum); q=q->succ;
         }
      else
         while (p->succ!=head) pop_back();
   }
   return *this;
}
```





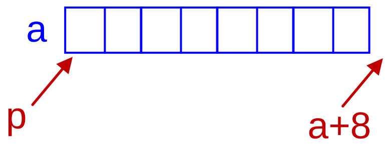




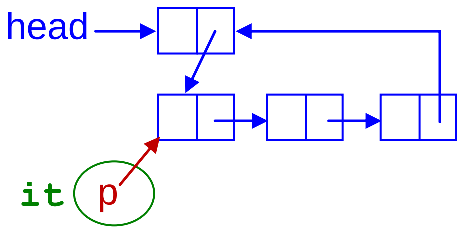
- Example 2 (Cont'd)

Motivation for iterators

```
int a[8];
for (int* p=a;p!=a+8;++p)
   cout << *p;
```




```
for (node* p=head->succ;p!=head;p=p->succ)
   cout << p->datum;
```




```
list b;
for (list::iterator it=b.begin();it!=b.end();++it)
   cout << *it;
```

Generic algorithm

```
template<typename iterator>
void print(iterator first,iterator last)
{
   for (iterator it=first;it!=last;++it)
      cout << *it;
}
print(a,a+8);                  // iterator = int*
print(b.begin(),b.end());      // iterator = list::iterator
```

Iterators

An iterator is a pointer or a pointer-like object that provides a general method of iterating over the elements within a data structure (or, a container, i.e. an object that contains other objects).

- Example 2 (Cont'd)

Iterator categories

input ← forward ← bidirectional ← random access
output ← forward

	output	input	forward
Read		=*i	=*i
Access		->	->
Write	*i=		*i=
Iteration	++	++	++
Comparison		== !=	== !=

	bidirectional	random-access
Read	=*i	=*i
Access	->	-> []
Write	*i=	*i=
Iteration	++ --	++ -- + - += -=
Comparison	== !=	== != < <= > >=

Example

Singly linked lists support forward iterators.
Doubly linked lists support bidirectional iterators.
Pointers to array elements are random access iterators.

STL supports iterators.

typdedef T* iterator;

typedef char* iterator;

For examples,
Bidirectional list
Random access vector, deque, string

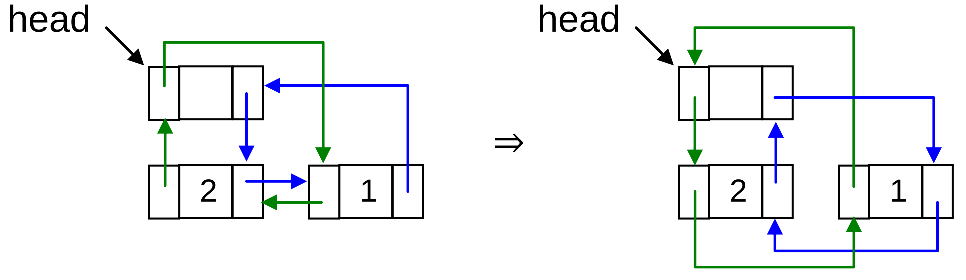
STL has 3 components.

1 Containers
2 Generic algorithms
3 Iterators

Iterators are the glue that holds containers and generic algorithms together.

- Example 2 (Cont'd)

```
                                                        class Bfn
template<class InputIterator,class T>
T accumulate(InputIterator first,InputIterator
                                             last,T init)
{                                                   Bfn f
   T result=init;
   for (InputIterator it=first;it!=last;++it)
      result=result+*it;  // result=f(result,*it);
   return r;
}

template<class RandomAccessIterator>
void sort(RandomAccessIterator,RandomAccessIterator);

#include <iostream>
#include <string>
#include <list>                // doubly-linked list
#include <algorithm>           // for sort
#include <numeric>             // for accumulate
using namespace std;
int main()
{
   string a("pluto");
   sort(a.begin(),a.end());
   cout << a;                  // loptu
   list<int> b;
   for (int i=1;i<=7;i++) b.push_back(i);
   cout << accumulate(b.begin(),b.end(),0); // 28
   b.sort();
}
```



- Example 2 (Cont'd)

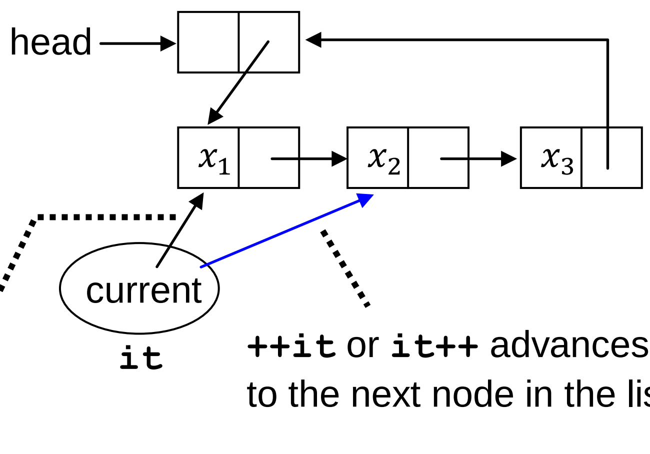
Forward iterators supported by singly linked lists

Recall that forward iterators support the following operations:

Read	`=*it`
Access	`->`
Write	`*it=`
Iteration	`++`
Comparison	`== !=`

Iterator type pointing to `int` (analogous to `int*`)

```
class list::iterator {
public:
   iterator(node* cur=0) : current(cur) {}
   int& operator*() const;
   int* operator->() const;
   iterator& operator++();
   const iterator operator++(int);
   bool operator==(iterator it) const   // call by value
   { return current==it.current; }
   bool operator!=(iterator it) const
   { return current!=it.current; }
private:
   node* current;
};
```



`++it` or `it++` advances the pointer to the next node in the list

`*it` accesses the datum x_1 contained in the pointed-to node.

- Example 2 (Cont'd)

Required precondition: the iterator is *dereferenceable*
Note: An iterator **`it`** is *dereferenceable*, if **`*it`** is defined. That is, if **`it.current`** doesn't point to a header node.

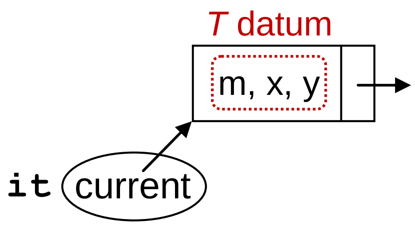
```
int& list::iterator::operator*() const
{
   return current->datum;
}
```

Remark – The return type must be **`int&`** (rather than **`int`**) so as to allow **`=*it`** and **`*it=`**.

```
int* list::iterator::operator->() const
{
   return &operator*();      // or, &**this
}                            // or, &current->datum;
```

Remarks

1 For an object **`it`** of class type in which operator**`->`** is defined,
`it->m`
is interpreted as
`it.operator->()->m` = **`(*it.operator->()).m`**
rather than
`it.operator->(m)`
Q: What is the type of **`m`**?



2 It follows that the return type of **`operator->()`** is *T**.
Note that **`it->m`** is meaningful only if *T* is a class type.
For our example,
`it->datum` // no
`*it.operator->()` // ok

3 **`&operator*()`**
`= &(this->operator*())`
`= &((*this).operator*())`
`= &**this`

- Example 2 (Cont'd)

Required precondition: the iterator is *dereferenceable*.

```
list::iterator& list::iterator::operator++()
{
    current=current->succ;            // prefix++
    return *this;                     // postfix++
}

const list::iterator list::iterator::operator++(int)
{
    iterator old=*this;  // default copy ctor; sharing is ok here
    ++*this;             // or, current=current->succ;
    return old;          // or, iterator old(current);
}                        //     node* old=current;
```

The postfix **++** operator must have a dummy parameter of type **int**; the default argument passed to it is zero, i.e.

```
it++ = it.operator++(0)
```

An explicit call may pass any integral value.

The **++** operator may be a non-member function with one parameter of class or enumeration type. In this case, the 2nd parameter of the postfix **++** operator shall be of type **int**.

To be consistent with built-in types, the return types of the prefix and postfix ++ operators shall be as presented. For example, as with built-in types,

++++it is allowed

(∴ the prefix ++ operator, like assignment operator, returns a reference to **it**, i.e. ***this**)

but

it++++ is disallowed   It is allowed in STL!!

(∴ the return type of the postfix ++ operator should not be **iterator**)

Remark
Prefer **++it;** to **it++;** because the former is less expensive.

- Example 2 (Cont'd)

  Iterator type pointing to **`const int`** (analogous to **`const int*`**)

```
class list::const_iterator {
public:
   const_iterator(const node* cur=0)
   : current(cur) {}
   const int& operator*() const
   { return current->datum; }
   const int* operator->() const
   { return &operator*(); }
   const_iterator& operator++();
   const const_iterator operator++(int);
   bool operator==(const_iterator it) const
   { return current==it.current; }
   bool operator!=(const_iterator it) const
   { return current!=it.current; }
private:
   const node* current;
};
```

```
list::const_iterator&
list::const_iterator::operator++()
{
   current=current->succ;
   return *this;
}
```

```
const list::const_iterator
list::const_iterator::operator++(int)
{
   const_iterator old=*this;
   ++*this;
   return old;
}
```

- Example 2 (Cont'd)

Implicit **`iterator`** → **`const_iterator`** conversion
(analogous to **`int*`** → **`const int*`**)

e.g.

```
list b;
list::const_iterator it=b.begin();
```

Method A – Type conversion operator

```
class list::const_iterator { … };   // define first

class list::iterator {
public:
   operator const_iterator() const { return current; }
   // other members remain unchanged
};
```

Since the type conversion operator returns a **`const_iterator`** object by value, it must be defined after the **`const_iterator`** class.

Method B – Friend + Converting ctor

```
class list::iterator {                // define first
   friend class const_iterator;
   // members remain unchanged
};

class list::const_iterator {
public:
   const_iterator(iterator it)
   : current(it.current) {}
   // other members remain unchanged
};
```

Since the converting ctor accesses the private member **`current`** of **`iterator`** class, it must be a friend of and defined after the **`iterator`** class.

- Example 2 (Cont'd)

  Explicit `const_iterator` → `iterator` conversion
  (analogous to `const int*` → `int*`)

  e.g. // or, `static_cast<list::iterator>(b.begin())`

```
const list b;
list::iterator it=list::iterator(b.begin()); // A
or
list::iterator it=b.begin().toIterator();    // B
```

  Note: STL list doesn't support explicit conversion.

  Method A – Friend + Explicit ctor

```
class list::const_iterator {        // define first
    friend class iterator;
    // members remain unchanged
};

class list::iterator {
public:
    explicit iterator(const_iterator it)
    : current(const_cast<node*>(it.current))
    {}
    // other members remain unchanged
};
```

  Method B – Ordinary member function

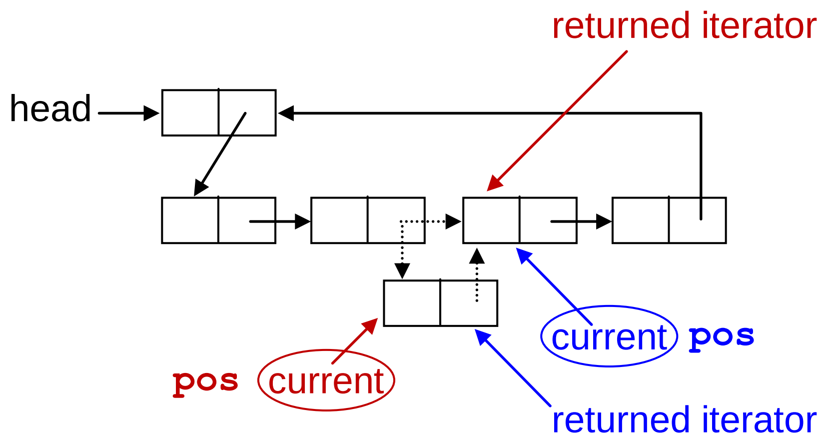
```
class list::iterator { … };          // define first

class list::const_iterator {
public:
    iterator toIterator()
    {
        return const_cast<node*>(current);
    }
    // other members remain unchanged
}
```

- Example 2 (Cont'd)

Modifiers – insert and erase

`list::iterator list::insert(iterator pos,int d);`

1 insert a node containing the datum **`d`** before the node pointed to by the iterator **`pos`**, and
2 return an iterator pointing to the node just inserted



`list::iterator list::erase(iterator pos);`

Required precondition: the iterator **`pos`** is *dereferenceable*
Postcondition: After the erasion, the iterator **`pos`** isn't *dereference-able* – the node it points to was just erased!

1 erase the node pointed to by the iterator **`pos`**
2 return an iterator pointing to the node immediately following the node just erased

Observe that both **`insert`** and **`erase`** of the **`list`** class have to access the private member **`current`** of the **`iterator`** class.

Method A – Friend

```
class list::iterator {
   friend class list;
   // members remain unchanged
};
```

- Example 2 (Cont'd)

```
list::iterator list::insert(iterator pos,int d)
{
   node* p=new node(d,pos.current);
   node* q=head;
   while (q->succ!=pos.current) q=q->succ;
   q->succ=p;
   return p;
}
```

```
list::iterator list::erase(iterator pos)
{
   node* q=head;
   while (q->succ!=pos.current) q=q->succ;
   q->succ=pos.current->succ;   //*
   delete pos.current;
   return q->succ;
}
```

Method B – Type conversion operator

```
class list::iterator {
public:
   operator node*() const { return current; }
   // other members remain unchanged
};
```

Next, replace every occurrence of **pos.current** in Method A by **pos**, except for that occurs in the starred line.

Finally, rewrite the starred line as
q->succ=((node*)pos)->succ;

Q: Why can't it be written as **q->succ=pos->succ;**

Modifiers – insert and erase at two ends

```
void list::push_front(int d) { insert(begin(),d); }
void list::push_back(int d) { insert(end(),d); }
void list::pop_front() { erase(begin()); }
```

- Example 2 (Cont'd)

```
void list::pop_back()
{
   iterator pos;
   for (iterator it=begin();it!=end();pos=it++);
   erase(pos);
}
```

Iterators and capacity

```
list::iterator list::begin() { return head->succ; }
```

```
list::const_iterator
list::begin() const { return head->succ; }
```

```
list::iterator list::end() { return head; }
```

```
list::const_iterator
list::end() const { return head; }
```

Notice that all of them return by value, rather than by const value. This is sometimes useful. For example,

`*++begin()` // access the 2nd element of the list

```
bool list::empty() const { return begin()==end(); }
```

Generic algorithm and user interface // defined in <algorithm>

```
template<class InputIterator,class T>
InputIterator find(InputIterator first,
              InputIterator last,const T& value)
{
   InputIterator it;
   for (it=first;it!=last&&*it!=value;++it);
   return it;
}
```

```
ostream& operator<<(ostream& os,const list& a)
{
   list::const_iterator it=a.begin();
   while (it!=a.end()) os << *it++ << " ";
   return os << endl;
}
```

- Example 2 (Cont'd)

```
int main()
{
  list b; char c;
  while (cout << "Command: ",cin >> c)
   switch (c) {
   int d;
   case 'i': cin >> d; b.push_front(d); break;
   case 'j': int e; cin >> d >> e;
             b.insert(find(b.begin(),b.end(),e),d);
             break;
   case 'k': cin >> d; b.push_back(d); break;
   case 'd': cin >> d; {
             list::iterator it
                     =find(b.begin(),b.end(),d);
             if (it!=b.end()) b.erase(it); }
             break;
   case 'f': cin >> d;
             cout << (find(b.begin(),b.end(),d)
                             !=b.end()) << endl;
             break;
   case 'o': cout << "List b: " << b; break;
   case 'c': { list c(b); cout << "List c: " << c; }
             break;
   case 'a': { static list a; a=b;
               cout << "List a: " << a; }
             break;
   default:  cout << "Invalid command\n";
   }
}
```

Commands

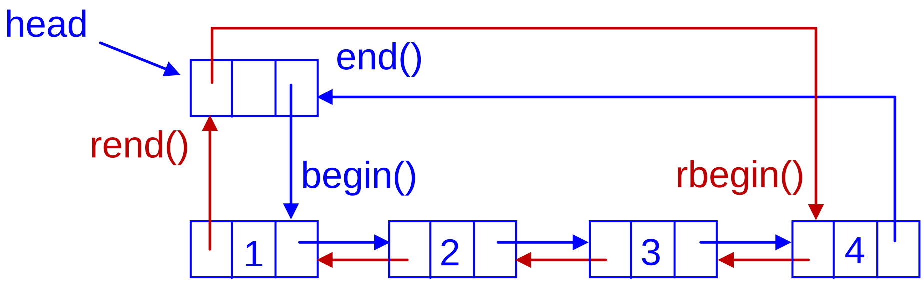
i	*d*	insert *d* at the begin
j	*d e*	insert *d* before the 1st *e* or at the end, if *e* isn't found
k	*d*	insert *d* at the end
d	*d*	erase the 1st *d* or do nothing, if *d* isn't found

- Example 2 (Cont'd)

  Reverse iterators

  1 only for bidirectional and random-access iterators
  2 iterate in the opposite direction

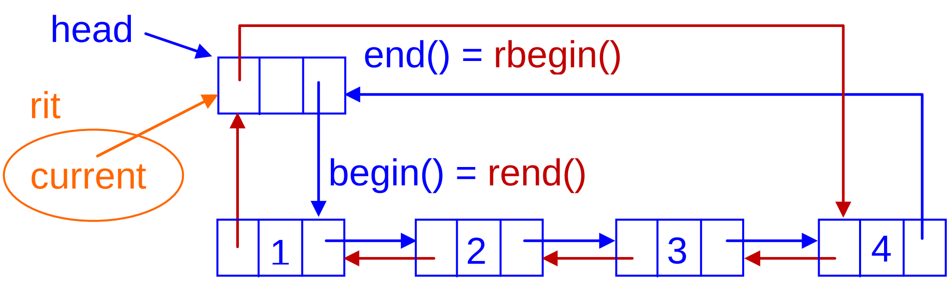
	forward	backward
iterator	`++`	`--`
reverse_iterator	`--`	`++`




```
list<int> a;
for (int i=1;i<=4;i++) a.push_back(i);
list<int>::iterator it;
list<int>::reverse_iterator rit;
for (it=a.begin();it!=a.end();++it)        // 1234
   cout << *it;
for (rit=a.rbegin();rit!=a.rend();++rit)
   cout << *rit;                           // 4321
for (it=a.end();it!=a.begin();)            // 4321
   cout << *--it;
for (rit=a.rend();rit!=a.rbegin();)        // 1234
   cout << *--rit;
```

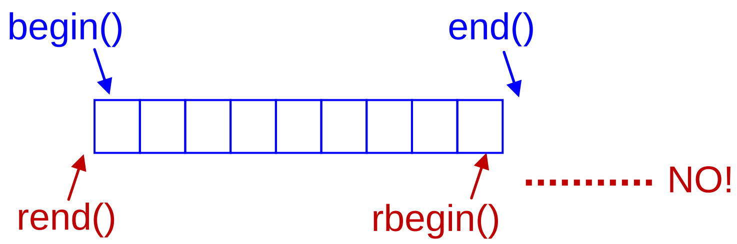
The preceding diagram for **rbegin()** and **rend()** is imaginary.

Reality



- Example 2 (Cont'd)

  Reason: To be consistent with pointers to arrays
  Recall that there is a valid pointer past the end of an array.
  But, there is no valid pointer before the beginning of an array.



How does it work?

```
for (rit=a.rbegin();rit!=a.rend();++rit)
   cout << *rit;
```

`*rit` returns **`currnt->pred->datum;`**

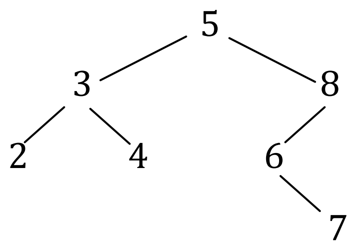
where **`pred`** points to the predecessor node.

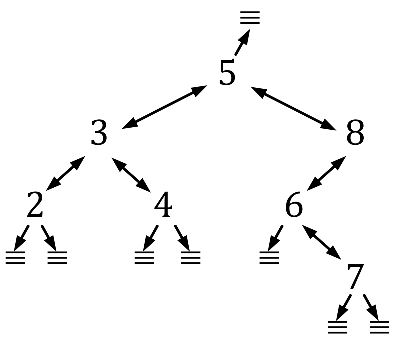
- Example 3 – Binary search tree

A binary search tree is a binary tree that satisfies

$$y\text{'s datum} < x\text{'s datum} \leq z\text{'s datum}$$

where x is a node in the tree, y is a node in the left subtree of x, and z is a node in the right subtree of x.





In addition to the lchild and rchild pointers, each node in the tree also contains a parent pointer. The root node is the only node whose parent pointer is NULL.

Tree traversal

preorder	root – left subtree – right subtree
inorder	left subtree – root – right subtree
postorder	left subtree – right subtree – root
reverse preorder	root – right subtree – left subtree
reverse inorder	right subtree – root – left subtree
reverse postorder	right subtree – left subtree – root
level order	

For the preceding binary tree, we have

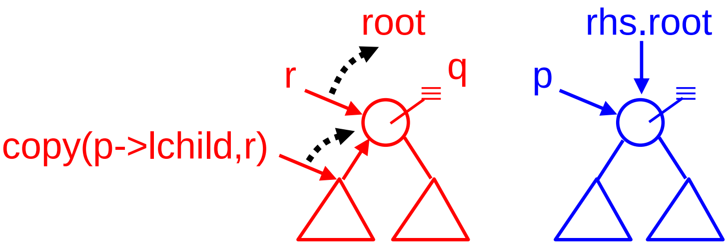
preorder	5 3 2 4 8 6 7
inorder	2 3 4 5 6 7 8
postorder	2 4 3 7 6 8 5
reverse preorder	5 8 6 7 3 4 2
reverse inorder	8 7 6 5 4 3 2
reverse postorder	7 6 8 4 2 3 5
level order	5 3 8 2 4 6 7

- Example 3 (Cont'd)

```
int main()
{
   bst b; char c;
   while (cout << "Command: ",cin >> c)
      switch (c) {
      int d;
      case 'i': cin >> d; b.insert(d); break;
      case 'd': cin >> d; b.erase(d); break;
      case 'f': cin >> d; cout << b.find(d) << endl;
                break;
      case 'o': cout << "BST b: "; b.inorder();
                break;
      case 'c': { bst c(b); cout << "BST c: ";
                  c.inorder(); }
                break;
      default:  cout << "invalid command\n";
      }
}

class bst {
public:
   bst();
   bst(const bst&);
   void inorder() const;
   ~bst();
   void insert(int);
   bool find(int) const;
   void erase(int);
private:
   struct node;
   node* root;
   void destroy(node*);
   node* copy(node*,node*);
   void inorder(node*) const;
   node* search(int) const;
   node* succ(node*) const;
};
```

- Example 3 (Cont'd)

```
struct bst::node {
   node(int,node*,node*,node*);
   int datum;
   node *lchild,*parent,*rchild;
};

bst::node::node(int d,node* l,node* p,node* r)
: datum(d),lchild(l),parent(p),rchild(r)
{}

bst::bst() : root(NULL) {}

bst::~bst() { destroy(root); }

void bst::destroy(node* p)
{
   if (p!=NULL) {          // reverse postorder traversal
      destroy(p->rchild);
      destroy(p->lchild);
      delete p;
   }
}

bst::bst(const bst& rhs)
: root(copy(rhs.root,NULL))
{}

bst::node* bst::copy(node* p,node* q)
{
   if (p==NULL) return NULL;     // preorder traversal
   else {
      node* r=new node(p->datum,NULL,q,NULL);
      r->lchild=copy(p->lchild,r);
      r->rchild=copy(p->rchild,r);
      return r;
   }
}
```



- Example 3 (Cont'd)

```
void bst::inorder() const
{
   inorder(root);  cout << endl;
}
void bst::inorder(node* p) const
{
   if (p!=NULL) {                    // inorder traversal
      inorder(p->lchild);
      cout << p->datum << " ";
      inorder(p->rchild);
   }
}
```

Q: Why do we need the private recursive function
`void bst::inorder(node*) const;`

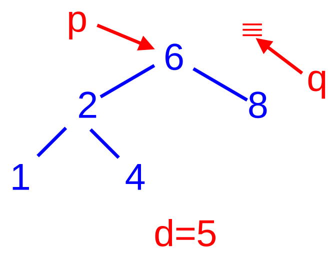
Can we make
`void bst::inorder() const;`
itself recursive as follows?

```
class bst {
private:
   bst(node* r) : root(r) {}  // add a private ctor
};

void bst::inorder() const
{
   if (root!=NULL) {
      bst(root->lchild).inorder();
      cout << root->datum << " ";
      bst(root->rchild).inorder();
   }
}
```

A: No

- Example 3 (Cont'd)

```
bool bst::find(int d) const
{
   return search(d)!=NULL;
}

bst::node* bst::search(int d) const
{
   node* p=root;
   while (p!=NULL)
      if (d==p->datum) break;
      else if (d<p->datum) p=p->lchild;
      else p=p->rchild;
   return p;
}
```

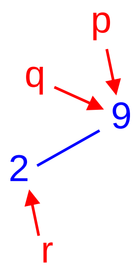


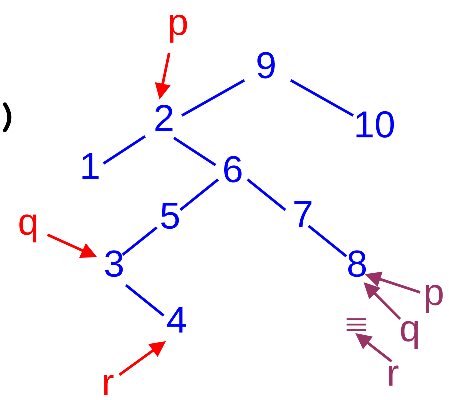

```
void bst::insert(int d)
{
   if (root==NULL)
      root=new node(d,NULL,NULL,NULL);
   else {
      node *p=root,*q=NULL;
      while (p!=NULL) {
         q=p;
         p=d<p->datum? p->lchild: p->rchild;
      }
      (d<q->datum? q->lchild: q->rchild)    // *
                             =new node(d,NULL,q,NULL);
   }
}
```

Recall that the starred line is for C++ only. It is equivalent to

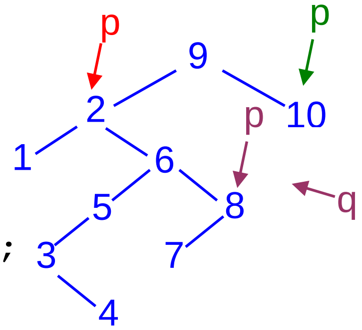
```
if (d<q->datum)
   q->lchild=new node(d,NULL,q,NULL);
else
   q->rchild=new node(d,NULL,q,NULL);
```

- Example 3 (Cont'd)

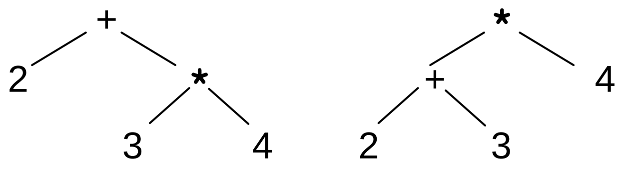





```
void bst::erase(int d)
{
   node* p=search(d);
   if (p==NULL) return;  // do nothing if d isn't in the tree
   node* q                // q is the node to splice out
   =  p->lchild==NULL||p->rchild==NULL?
      p:
      succ(p);
   node* r=q->lchild==NULL? q->rchild: q->lchild;
   if (r!=NULL) r->parent=q->parent;
   if (q->parent==NULL)
      root=r;
   else if (q==q->parent->lchild)
      q->parent->lchild=r;
   else
      q->parent->rchild=r;
   if (q!=p) p->datum=q->datum;
   delete q;
}

bst::node* bst::succ(node* p) const
{
   if (p==NULL) return p;     // retun null, if no successor
   if (p->rchild!=NULL) {
      p=p->rchild;
      while (p->lchild!=NULL) p=p->lchild;
   } else {
      node* q;
      do {
         q=p; p=p->parent;
      }
      while (p!=NULL&&q==p->rchild);
   }
   return p;
}
```



- Example 4 – Expression tree



Prefix expression	+ 2 * 3 4	* + 2 3 4
Infix expression	2 + 3 * 4	2 + 3 * 4
Postfix expression	2 3 4 * +	2 3 + 4 *

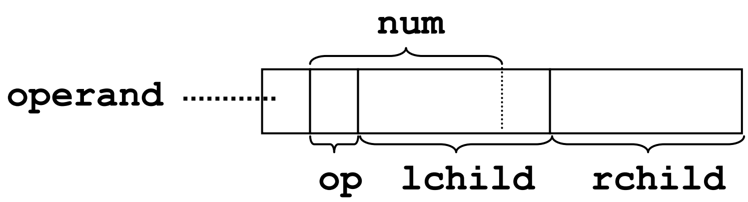
This examples takes a prefix expression, parses it and constructs an expression tree, displays the corrsponding full-parenthesized infix expression, and evaluates the infix expression.

```
int main()
{
   string s;
   while (cout << "Enter: ",getline(cin,s)) {
      if (s.empty()) continue;
      try {
         exp e(s);
         e.inorder();
         cout << " = " << e.eval() << endl;
      }
      catch(bad_exp&) { cout << "Try again\n"; }
   }
}

class bad_exp : public exception {
public:
   bad_exp(const string& msg) : msg(msg) {}
   ~bad_exp() throw() {}
   const char* what() const throw()
   { return msg.data(); }
private:
    string msg;
};
```

- Example 4 (Cont'd)

  Version A

```
class exp {
public:
   exp(const string&);
   ~exp();
   void inorder() const;
   int eval() const;
private:
   struct node;
   node* root;
   node* parse(istringstream&);
   void destroy(node*);
   void inorder(const node*) const;
   int eval(const node*) const;
};

struct exp::node {
   node(char,node*,node*);
   node(int);
   bool operand;          // discrimination tag
   union {                // anonymous union
      int num;
      struct { char op; node *lchild,*rchild; };
   };                     // anonymous struct
};

exp::node::node(char op,node* l,node* r)
: operand(false),op(op),lchild(l),rchild(r)
{}

exp::node::node(int n) : operand(true),num(n)
{}
```



- Example 4 (Cont'd)

  Anonymous union and anonymous struct

  An anonymous union (or struct) defines an unnamed object of unnamed type.

  The members of an anonymous union (or struct) are treated as defined in the scope in which the anonymous union (or struct) is declared, e.g.

```
node* t=new node('+',new node(2),new node(3));
t->operand
t->op
t->lchild->num
```

  Anonymous structs are nonstandard and extensions of most C++ compilers. Without anonymous structs, we have to write

```
union {
   int num;
   struct { char op; node *lchild,*rchild; } oper;
};
```

  Notice that the struct type must be unnamed, because an anonymous union can only have nonstatic data members, i.e. it shan't have member types and member functions.

```
exp::node::node(char op,node* l,node* r)
: operand(false)
{
   oper.op=op; oper.lchild=l; oper.rchild=r;
}
node* t=new node('+',new node(2),new node(3));
t->operand
t->oper.op
t->oper.lchild->num;
```

- Example 4 (Cont'd)

  String streams

  String streams support I/O operations on string objects.

```
#include <sstream>
int main()
{
   string s("2 3 4");
   istringstream sin(s);
   ostringstream sout;
   int n;
   while (sin >> n) sout << n << ' ';
   cout << sout.str() << sout.str().size();
}                            // 2 3 4 6
```

  Below is an example on reading mixed-type data from a string.

```
istringstream sin("* 3 / 4 2");
ostringstream sout;
union { int n; char c; };
while (!sin.eof()) {         // while(sin>>n) ×
   sin >> n;
   if (sin.fail())
      if (sin.eof())         // " "    failbit=1, eofbit=1
         break;              // "+"    failbit=1, eofbit=1
      else {                 // "*"    failbit=1, eofbit=0
         sin.clear();        // reset
         sin >> c;
         sout << c << ' ';
      }
   else
      sout << n << ' ';      // "2"    failbit=0, eofbit=1
}                            // "2 "   failbit=0, eofbit=0
cout << sout.str();
```

  This doesn't work when the string contains characters + or -, as they are treated as plus or minus signs on reading a number.

- Example 4 (Cont'd)

To correct it, read characters instead.

```
istringstream sin("+ 2 - 3 4");
ostringstream sout;
union { int n; char c; };
while (sin >> c) {              //*
   if (isdigit(c)) {
      sin.unget();
      sin >> n;
      sout << n << ' ';
   } else sout << c << ' ';
}
cout << sout.str();
```

The starred line skips white-space characters.
After reading each non-whice-space character, failbit=0, eofbit=0.
At last, failbit=1, eofbit=1.

Parsing a mixed type expression

We now use this technique to parse a mixed-type expression:

```
exp::exp(const string& s)
try
: root(NULL)
{
   istringstream sin(s);
   root=parse(sin);
   char c;
   if (sin >> c)
      throw bad_exp("Extra op/number/char");
}
catch (bad_exp& e) {
   if (root!=NULL) destroy(root);
   cout << e.what() << endl;
}                               // automatically rethrow
```

- Example 4 (Cont'd)

```
exp::node* exp::parse(istringstream& sin)
{
   union { int num; char op; };
   if (sin >> op)
      if (isdigit(op)) {
         sin.unget(); sin >> num;
         return new node(num);
      } else if (string("+-*/").find(op)
                                  !=string::npos) {
         node *l=NULL,*r;
         try {
            l=parse(sin); r=parse(sin);
         }
         catch (bad_exp&) {
            if (l!=NULL) destroy(l);
            throw;
         }
         return new node(op,l,r);
      } else throw bad_exp("Illegal char");
   else throw bad_exp("Missing operand");
}
```

Since the tree has been constructed in postorder, it shall be destroyed in reverse preorder.

```
exp::~exp() { destroy(root); }
void exp::destroy(node* t)
{
   if(t->operand) delete t;
   else {
      node *l=t->lchild,*r=t->rchild;
      delete t;
      destroy(r);
      destroy(l);
   }
}
```

- Example 4 (Cont'd)

```
void exp::inorder() const { inorder(root); }
void exp::inorder(const node* t) const
{
   if (t->operand) cout << t->num;
   else {
      cout << '(';
      inorder(t->lchild);
      cout << t->op;
      inorder(t->rchild);
      cout << ')';
   }
}
```

```
int exp::eval() const { return eval(root); }
int exp::eval(const node* t) const
{
   if (t->operand) return t->num;
   else
   switch (t->op) {
   case '+': return eval(t->lchild)+eval(t->rchild);
   case '-': return eval(t->lchild)-eval(t->rchild);
   case '*': return eval(t->lchild)*eval(t->rchild);
   case '/': return eval(t->lchild)/eval(t->rchild);
   }
}
```

Version B

In this version, we shall build a map and use it to speed up the evaluation of an expression.

`'+'`	0	→ function **+**	
`'-'`	1	→ function **-**	dic
`'*'`	2	→ function *****	
`'-'`	3	→ function **/**	

It would be convenient if **`t->op`** can be used to index the array: **`dic[t->op]`**

- Example 4 (Cont'd)

  Map (or dictionary)

  A map is an associative container that supports unique keys and fast retrieval of data based on the keys (the entries of a map are ordered by the keys).

  Map supports bidirectional iterators.

```
#include <map>
#include <utility>
int main()
{
   typedef pair<string,int> entry;
   entry d[4]={entry("pluto",5),
               entry("snoopy",6),
               entry("doraamon",8),
               entry("garfield",8)};
   map<string,int> dic(d,d+4);
   map<string,int>::iterator it;
   for (it=dic.begin();it!=dic.end();++it)
      cout << it->first << ' ' << it->second;
   cout << dic["snoopy"];
}
```

  Pointers/references to static member functions

  Let the function below be a *static* private member of class **`exp`**

```
int exp::plus(int x,int y) { return x+y; }
```

  Just like non-member functions, this static member function is of the function type

```
int(int,int);                    // function type
```

  which is independent of the class in which they resides.

  E.g. inside a member function of class **`exp`**, we may write

```
int (*pf)(int,int)=plus;         // or, &plus
int (&rf)(int,int)=plus;
cout << pf(2,3) << rf(2,3);
```

- Example 4 (Cont'd)

```
class exp {
// add the following private members
private:
   static int plus(int x,int y) { return x+y; }
   static int minus(int x,int y) { return x-y; }
   static int multiplies(int x,int y) { return x*y; }
   static int divides(int x,int y) { return x/y; }
   typedef int (*bop)(int,int);
   typedef pair<char,bop> entry;
   static entry d[4];
   static map<char,bop> dic;
};
exp::entry exp::d[4]={entry('+',plus),
                      entry('-',minus),
                      entry('*',multiplies),
                      entry('/',divides)};
map<char,exp::bop> exp::dic(d,d+4);

int exp::eval() const { return eval(root); }
int exp::eval(const node* t) const
{
   if (t->operand)
      return t->num;
   else
      return
      dic[t->op](eval(t->lchild),eval(t->rchild));
}
```

Version C

Suppose we want to

1) store the value of each subexpression in the root node of its representation tree, and
2) compute e1 $\oplus$ e2 using the stored values of e1 and e2

In order to do so, $\oplus$ has to be a non-static member function of class **exp::node**

- Example 4 (Cont'd)

First of all, let's redefine **`exp::node`** as

```
struct exp::node {
   node(char,node*,node*);
   node(int);
   int plus() { return lchild->num+rchild->num; }
   int minus() { return lchild->num-rchild->num; }
   int multiplies() { return lchild->num*rchild->num; }
   int divides() { return lchild->num/rchild->num; }
   bool operand;
   int num;
   char op;
   node *lchild,*rchild;
};
```

Observe that anonymous union doesn't help here, because an operand node is a subset of an operator node.

Pointers to on-static member functions (and data)

The language provides the following declarator and operators:

`::*`	pointer to member declarator
`->*` `.*`	pointer to member operators

1 **`->*`** and **`.*`** are tokens.
  `::` and **`*`** may be separated by spaces.

2 "pointer to member" is a little bit misleading – it actually means "pointer to non-static member".

3 There are no references to non-static members.

Inside a member function of class **`exp`**, we may write

```
node* r=new node('+',new node(2),new node(3));
int (node::*pmf)()=&node::plus;
int node::*pmd=&node::num;          // or, exp::node::*
r->*pmd=(r->*pmf)();
*r.*pmd=(*r.*pmf)();
```

Note: A pointer to member can only be formed by **&**qualified-id.

- Example 4 (Cont'd)

  It can't be written as

  &(qualified-id) // cannot be enclosed in parentheses
  qualified-id // no function to pointer conversion
  &unqualified-id // even within the scope of unqualified-id's class

  E.g. Inside a member function of class **node**, we may write

  ```
  int (node::*pmf)()=&node::plus;
  ```

  where the underlined part can't be omitted.

  Finally, we have to modify class **exp** as follows:

```
class exp {
// remove and add the following private members
private:
//  int eval(const node*) const;     // remove
    void eval(node*) const;
    typedef int (node::*bop)();
    typedef pair<char,bop> entry;
    static entry d[4];
    static map<char,bop> dic;
};
exp::entry exp::d[4]
= { entry('+',&node::plus),entry('-',&node::minus),
    entry('*',&node::multiplies),
    entry('/',&node::divides)};
map<char,exp::bop> exp::dic(d,d+4);
int exp::eval() const
{
    eval(root); return root->num;
}
void exp::eval(node* t) const
{
    if (!t->operand) {
        eval(t->lchild); eval(t->rchild);
        t->num=(t->*dic[t->op])();
    }
}
```

Appendix

The **string** class runs in this lecture.

```
class string {
public:

// types
   typedef size_t size_type;
   typedef char&  reference;
   typedef const char& const_reference;

// ctor/copy/dtor
   string(const char* ="");
   string(const string&);
   ~string();
   string& operator=(const string&);
   string& operator=(const char*);

// string operations
   const char* c_str() const;

// modifiers
   string& operator+=(const string&);
   string& operator+=(const char*);

// element access
   reference operator[](size_type);
   const_reference operator[](size_type) const;
   reference at(size_type)
   const_reference at(size_type) const;

// capacity
   size_type capacity() const { return _capacity; }
   size_type size() const { return _size; }
   bool empty() const { return _size==0; }
private:
   size_type _cap(size_type);
   size_type _size,_capacity;
   char* _data;
};
```

// non-member functions

```
string operator+(const string&,const string&);
string operator+(const string&,const char*);
string operator+(const char*,const string&);
ostream& operator<<(ostream&,const string&);
istream& operator>>(istream&,string&);
```

Example

```
#include <string>
int main()
{
   string s("snoopy"),t("Pluto");
   s[0]='S';
   cout << s+t;                          // SnoopyPluto
   cout << s.size() << s.capacity();     // 6 15   (6 6)
   s+=t;
   cout << s;                            // SnoopyPluto
   cout << s.size() << s.capacity();     // 11 15  (11 12)
   s+=s;
   cout << s;                  // SnoopyPlutoSnoopyPluto
   cout << s.size() << s.capacity();     // 22 31  (22 24)
   s="c++";
   cout << s;                            // c++
   cout << s.size() << s.capacity();     // 3 31   (3 24)
}
```

VC++ GNU C++