W.P.Hwang
CS-CCS-NCTU

Lecture – Class and ADT

Code reusability

- Recall the advantages of ADT

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

- Code reusability

  An ADT, once defined, may be kept in a user-defined library and reused in case of need. This is often accompanied with separate compilation.

- Example

  For illustration purpose, consider the **`stack`** class and 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;
};
```

- Example (Cont'd)

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--; }
```

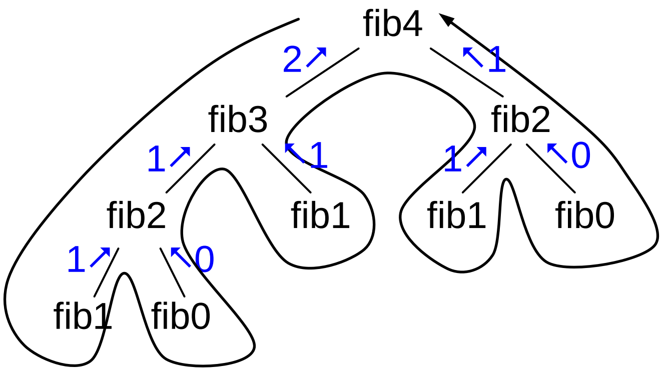
One definition rule (odr)

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

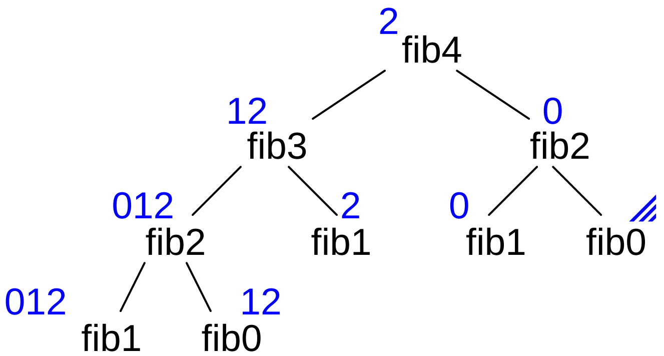
Applications that use the user-defined library

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

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



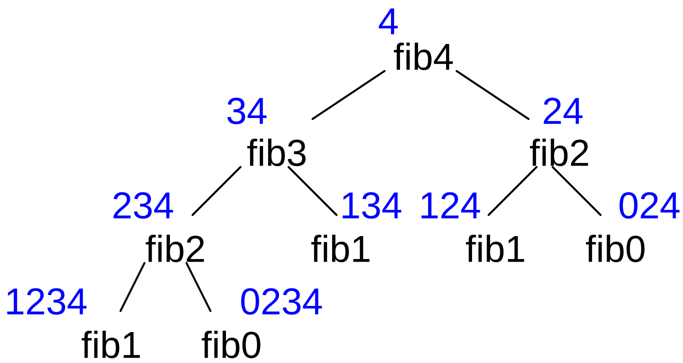
Iterative backtracking employs a stack.



- Example (Cont'd)

```
#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);
}
```

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;
   }
}
```

- Example (Cont'd)

  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.

  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);
      cout << "By runtime stack simulation: ";
      s.push(n);
      cout << "fib(" << n << ") = " << fib();
      cout << "\n\nEnter an integer: ";
   }
}
```

- Example (Cont'd)

On declarations and definitions of functions and objects

	Function	Object
Definition	with body	w/o **`extern`** or with initializer
Declaration	without body	with **`extern`** and w/o 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);
```

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 that aren't **`static`** (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

- Example (Cont'd)

File 3 – **fibback.cpp** (rev. 1)

```
// File 4: extern stack s
// File 5: stack s;
```

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

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** (rev. 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 above
```

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 << ::x;      // ambiguous, global x
}
```

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

```
                          compiled to
namespace { body }   ⇒   namespace unique { body }
                          using namespace unique;
```

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

  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.

Compile and run a stack application

- How to compile and run a stack application?

  bsd2> **`g++48`** *application-files* **`stack.cpp`**
  bsd2> **`./a.out`**

  Drawback

  For each stack application, **`stack.cpp`** has to be recompiled. To avoid the recompilation, it shall first be compiled to an object file **`stack.o`**.

  bsd2> **`g++48 -c stack.cpp`**
  bsd2> **`g++48`** *application-files* **`stack.o`**

  Drawback

  In general, suppose an application needs the code contained in several pre-compiled object files, it is tedious to write

  bsd2> **`g++48`** *application-files all-pre-compiled-object-files-used*

  It is desired to place all the pre-compiled code into a library and uses that library instead.

Create your own library

Static libray

- Property

  1 The executable contains not only the application code but also all the referenced code retrieved from the static library.
  2 The static library isn't need at run time.
  3 The static library is an archive file.

- Naming convention

  Unix **`lib`***name***`.a`** (**`.a`** for archive)
  Windows *name***`.lib`**

- Example (Unix)

  Step 1: Compile those files for library into object code

  bsd2> **`g++48 -c stack.cpp`**

  Step 2: Create a static library

  insert at end with replacement   all object files go to the library

  bsd2> **`ar r libsnoopy.a *.o`**
  `ar: creating libsnoopy.a`

  Step 3: Obtain an executable

  bsd2> **`g++48 main.cpp fibsimu.cpp fibback.cpp`**
  `-lsnoopy -L.`

  the path to search for **`libsnoopy.a`** for the linker
  `.` means current directory

  library **`libsnoopy.a`**: prefix and suffix shall not be written

  Step 4: Execute the excutable

  bsd2> **`./a.out`**
  `Enter an integer:`

Dynamically-linked libray (Shared library)

- Property

  1 The executable conatins the application code and the information needed at run time to locate the referenced code from the shared library.
  2 The shared library must be accessible at run time.
  3 The shared library is essentially an executable, but it needs a host to run.

- Naming convention

Unix	**`lib`***name***`.so`**	(**`.so`** for shared object)
Windows	*name***`.dll`**	(**`.dll`** for dynamically linked library)

- Example (Unix)

  Step 1: Compile those files for library into object code

  bsd2> **`g++48 -c -fPIC stack.cpp`**

  poinsition independent code; needed for shared library

  Step 2: Create a shared library

  bsd2> **`g++48 -shared -o libsnoopy.so *.o`**

  create a shared library   essentially an excutable

  Step 3: Obtain an executable

  bsd2> **`g++48 main.cpp fibsimu.cpp fibback.cpp`**
  `-lsnoopy -L. -rpath=.`

  the path to search for **`libsnoopy.so`** at run time

  first, look up **`libsnoopy.so`**, then **`libsnoopy.a`**

  Step 4: Execute the excutable

  bsd2> **`./a.out`**
  `Enter an integer:`

Separate compilation vs inclusion compilation

- Separate compilation

  1 The interface (**`.h`**) and implementation (**`.cpp`**) are separated into two files.
  2 Required for non-template functions and classes

- Inclusion compilation

  1 The interface and implementation are combined into a single file (**`.h`**).
  2 Required for template functions and classes (C++11)

  N.B. Prior to C++11, the keyword **`export`** is used to designate separate compilation for templates. But, C++11 reserves it for future use.

Stack and queue

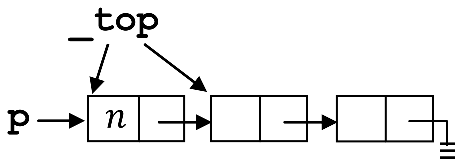
- Stack implementation – linked list representation

```
class stack {
public:
   stack() : _top(nullptr) {}
   ~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==nullptr; }
private:
   struct node {
      node(int,node*);    // public members of class node
      int datum;          // but, visible only within classes
      node* succ;         // node and stack
   };
   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() {}
```

- Queue implementation – linked list representation

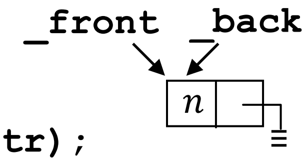
```
class queue {
public:
   queue() : _front(nullptr) {}
   ~queue() { while (!empty()) pop(); }

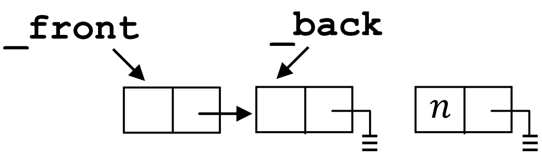
   void push(int);          // enqueue
   void pop();              // dequeue

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

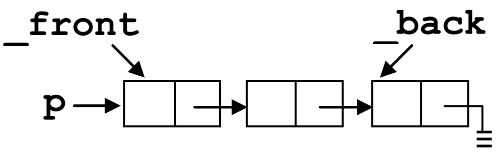
struct queue::node {        // class definition
   node(int d,node* s) : datum(d),succ(s) {}
   int datum;
   node* succ;
};

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





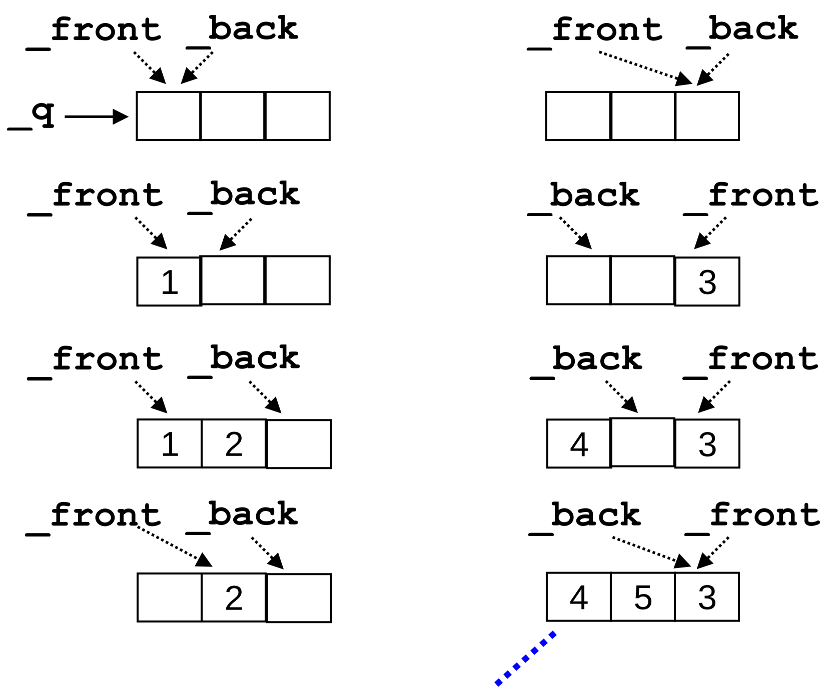

```
inline void queue::pop()
{
   node* p=_front;
   _front=_front->succ;
   delete p;
}
```



- Queue implementation – sequential array representation

```
class queue {
public:
   queue(int=80);
   ~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! Can't distinguish between queue full and queue empty

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

- Queue implementation (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])
{}

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;
}
```

- Stack and queue application – palindrome

  Determine if an integer reads the same forwards as backwards

```
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;
}
```

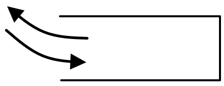
Before return to the caller, call `q.~queue()` and `s.~stack()`, in that order.

For example, **n** = 12321
stack s _top → 1 2 3 2 1
queue q _front → 1 2 3 2 1 ← **_back**

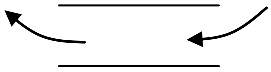
Comment
The loop may be terminated earlier, if the stack or queue size is known. See the next example.

STL stack, queue, and deque

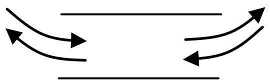
- Stack (FILO, First-In-Last-Out)
  Insert and delete at the same end

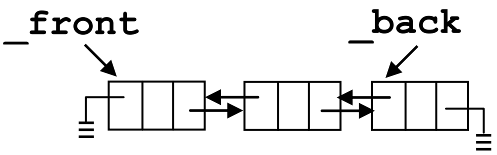


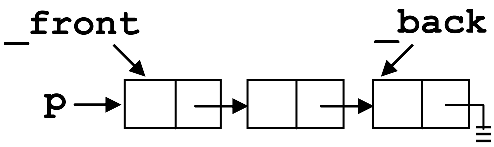
- Queue (FIFO, First-In-First-Out)
  Insert at one end and delete at the other end



- 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, a container (i.e. data structure) is 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 queue or deque
void p(T& c)
{
   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

  2 **value_type** is an enumerator or a static data member of class **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(nullptr),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& val)
{
    _top=new node(val,_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 **T = double** is desired, 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 rep.

```
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 _top+1; }
   bool empty() const { return _top==-1; }
private:
   T* stk;
   int _top;
   size_type maxsz;
};

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

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.
It is exactly the same reason why the stack shall not be implemented by a static array, say
T stk[80];

Q: What is the data member **maxsz** for?
A: It is needed when a stack is going to be copied to another or enlarged in case of no more available space.

- Example (Cont'd)

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

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

Q: Can we write `stk[_top]=val;` in the starred line?
A: No, since `stk[_top]` is uninitialized (it might be occupied and destroyed before), it can't be modified semantically.
In particular, if `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--;    //*
}
```

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`*.
For example, the two statements in the starred line may be combined into
`stk[_top--].~T();`

Nonstatic member

- Nonstatic members belong to each object of the class.
- 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 **`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
}

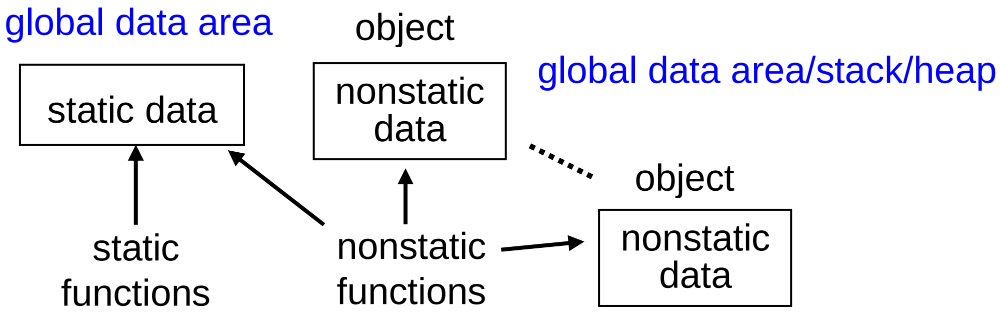
string a("snoopy");
const string b("Snoopy");
a[0]='S';                 // both are viable; ① is the best viable
cout << b[0];             // only ② is viable
```

  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 member

- 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.



- Example (Hypothetical)

```
class string {          watch the space
public:
   string(const char* =_dstring);            ①
   static void dstring(const char*);         ②
private:
   static constexpr int _dsize=16;           ③
   static char _dstring[_dsize];
   char* _data;
};
char string::_dstring[_dsize]="";            ③
void string::dstring(const char* s)
{
   strncpy(_dstring,s,_dsize-1);
}
string::dstring("Snoopy");                   ④
string a;
a.dstring("Pluto");                          ④
```

- Example (Cont'd)

  ① Only static members can be used as default parameters.
  ② The keyword **`static`** can only appear in the declaration, but not in the definition. Why?
  ④ Static members may be accessed in either way.
  In *exp*`.`static-member or *exp*`->`static-member, *exp* will be evaluated, e.g.
  `(cout << a,a).dstring("pluto");`
  ③ See below for initializations of static data members

- Static data members are not initialized by ctors. They have to be initialized outside the class body in the implementation file.

- Exceptions

  1 Static data members of **`const`** integral type can have in-class initializers.
  2 Static data members of **`constexpr`** literal type must have in-class initializers.

  Even if they are initialized, they are treated as declarations, and must be defined outside the class body w/o the initializer, if they are odr-used (i.e. if they need storage).

  N.B. Nonstatic data members can't be declared **`constexpr`**.

- Example (Cont'd)

  Although not required, we may define

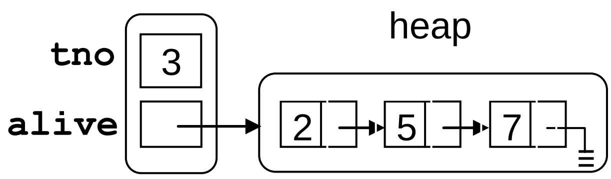
```
constexpr int string::_dsize;
```

  outside the class body. The definition is needed, if **`_dsize`** is referenced, e.g.

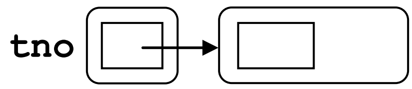
```
class string {
public:
   static const int& dsize;
private:
   static constexpr int _dsize=16;
};
const string::int& dsize=_dsize;
```

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

  Let **tank** be a class that has 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




```
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;
}
```

- Example (Cont'd)

  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 members.
  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`**.

  Q: May we have a "dtor" to delete static datum **`tank::tno`** and call it *manually* before returning from **`main`**?
  A: Bad idea – prefer automaic deletion

  What we need is a smart pointer, e.g. shared_ptr or unique_ptr. Here is a rewrite of the preceding example

```
#include <memory>
class tank {
public:
   // same as above
private:
   int tid;
   static unique_ptr<int> tno;
};
unique_ptr<int> tank::tno(new int(0));
```

- 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 (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 reevaluated 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/move ctor, copy/move assignment operator, destructor are special member functions
- If a class doesn't explicitly declare them, they will be
  1 implicitly declared
  2 implicitly defined, if they are needed.
- Example

```
class X {};
```

is compiled to something like

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

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

```
X a;                  // ctor and dtor
X b(a);               // copy ctor
X c(std::move(a));    // move ctor
b=a;                  // copy assignment operator
c=std::move(a);       // move assignment operator
```

They are defined as **`inline`** functions:

```
X::X() {}
X::~X() {}
X::X(const X&) : memberwise copy {}
X::X(X&&) : memberwise move {}
X& X::operator=(const X&){ memberwise copy-assign }
X& X::operator=(X&&){ memberwise move-assign }
```

Constructor

- 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/move ctor), a default ctor is declared implicitly.
- Non-delegating constructor

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

If *expressions* is omitted, **`mem_id`** is value-initialized .

If a nonstatic data member is not named by a **`mem_id`** in the mem-initializer list, then

1 if it has a brace-or-equal-initializer, initialize it as specified
2 otherwise, it is default-initialized

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

Const and reference data members must be initialized.

- Delegating constructor

```
X::X(...)
:   X(expression…)                // delegate to another ctor
{
    statements, if any
}
```

In this case, **`X(`***expression…***`)`** shall be the only mem-initializer.

If a ctor delegates to itself directly or indirectly, the program is ill-formed

- Example

```
class X {
public:
   X(int=0); 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; }     // error
```

or

```
class X {
public:
   X(int x=0) : x(x) {}
   const int& vx=x;      // C++11
private:
   int x;
};
```

Comment

```
struct Y {
   int a=2;      // must be in-class; must use = or {}
   static int b;
};
int Y::b(2)      // must not be in-class; may use =, {} or ()
```

A brace-or-equal-initializer serves as the default initialization of a member common to various ctors. When in need, it can be overwritten in a mem-initializer.

```
class X {
public:
   X() {}
   X(int x) : x(x) {}
   const int& vx=x;
private:
   int x{0};
};
```

- Example (Cont'd)

```
class X {
public:
   X() : X(0) {}          // delegating constructor
   X(int x) : x(x) {}
   const int& vx=x;
private:
   int x;
};
```

- Example

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

string::string(const char* s)
:  _size(strlen(s)),
   _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, for readability,

List member initializers in the declaration order.

- Example (Cont'd)

  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 are independent of the declaration order of 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))
{}
```

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

  Next, as in STL, the default ctor constructs an empty **string** object in which **_data** isn't null, **_size**==0, and **_capacity** is unspecified.

```
string::string()
:  _size(0),
   _capacity(0),
   _data(&(*new char[1]='\0')) {}
```
  or
```
string::string()
:  string("")              // delegating constructor
{}
```

  The latter implementation in effect combines the two ctors into **string::string(const char* ="");**

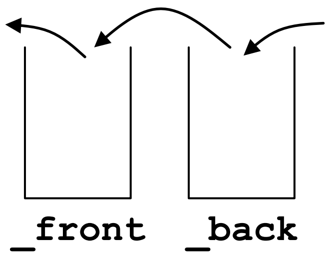
  Q: Which is better? two ctors or a single combined ctor?
  A: Having two ctors is better.

- 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();
      }
}
```

Clearly, every class must have at least one constructor. The question is: do we need to define a ctor for the **queue** class by ourselves?

First, observe that a **queue** object contains no data other than two **stack** subobjects 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.

There are three cases to consider.

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

For example,

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

In this case, we simply rely on the implicitly generated 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])
{}
```

- Example (Cont'd)

  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])
{}
```

  In this case, we have to choose between case 1 and case 2.

Destructor

- 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.
- Example

// Version A – Boolean flag

```
class X {
public:
   X() : p(new int),single(true) ()
   X(int n) : p(new int[n]),single(false) {}
   ~X()
   {
      if (single) delete p; else delete [] p;
   }
private:
   int* p;
   bool single;
};
```

// Version B – Lambda expression

```
#include <functional>
class X {
public:
   X()
   : p(new int),deleter([this]{ delete p; })
   {}
   X(int n)
   : p(new int[n]),deleter([this]{ delete[] p; })
   {}
   ~X() { deleter(); }
private:
   int* p;
   function<void(void)> deleter;
};
```

- Example (Cont'd)

  // Version C – Function object

```
#include <memory>
#include <functional>
class X {
public:
   X()
   : p(new int),deleter(default_delete<int>())
   {}
   X(int n)
   : p(new int[n]),deleter(default_delete<int[]>())
   {}
   ~X() { deleter(p); }
private:
   int* p;
   function<void(int*)> deleter;
};
```

- 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)

  Clearly, every class must have a destructor. The question is: do we need to define a dtor for the **queue** class by ourselves?

  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

- Example (Cont'd)

  ```
  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/move constructor (Part A)

Copy constructor

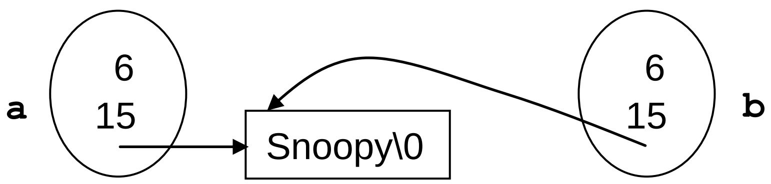
- A copy ctor is used to initialize one object by another object of the same class.
- Example

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

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



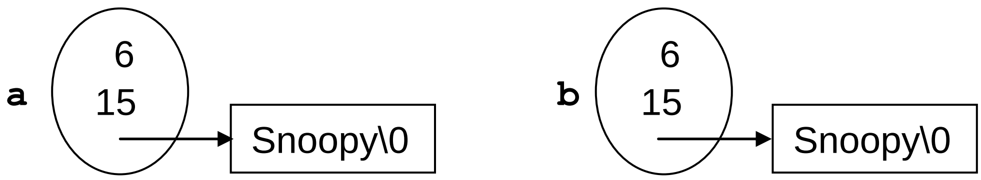
Remark
Sharing is efficient, but suffers from the problems illustrated above.

- Principle

  Define a copy/move 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))
{}
```

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

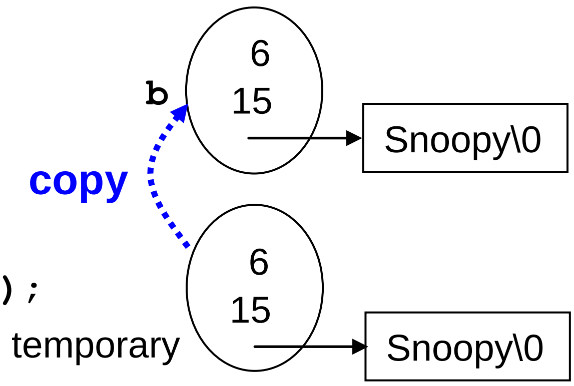


  Remark
  Copying is time and space consuming, but solves the problems caused by sharing.

Move ctor and resource stealing

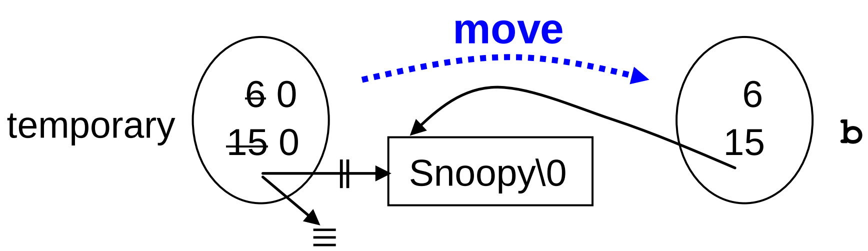
- Example A

```
void p(string b) {}
int main()
{
   p(string("Snoopy"));
}
```



  The temporary is copied to **`b`** and then destroyed as the last step in evaluating the function call.

  Since the temporary is invisible, any change to it doesn't visibly modify anything. Therefore, its resources may be moved to **`b`**.



- Example A (Cont'd)

Prior to C++11, we may try to define another copy stor:

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

Since a temporary is an rvalue, the result is unsatisfactory:

```
string a("Snoopy");
p(a);                    // call non-const copy ctor
p(string("Snoopy"));     // call const copy ctor
```

C++11 introduces rvalue reference and move ctor.

```
string::string(string&& rhs)         // move ctor
:  _size(rhs._size),
   _capacity(rhs._capacity),
   _data(rhs._data)
{
   rhs._size=rhs._capacity=0;
   rhs._data=nullptr;
}
```

and stipulates by overload resolution that

```
p(a);                    // lvalue, call copy ctor (const)
p(string("Snoopy"));     // rvalue, call move ctor
```

Comment

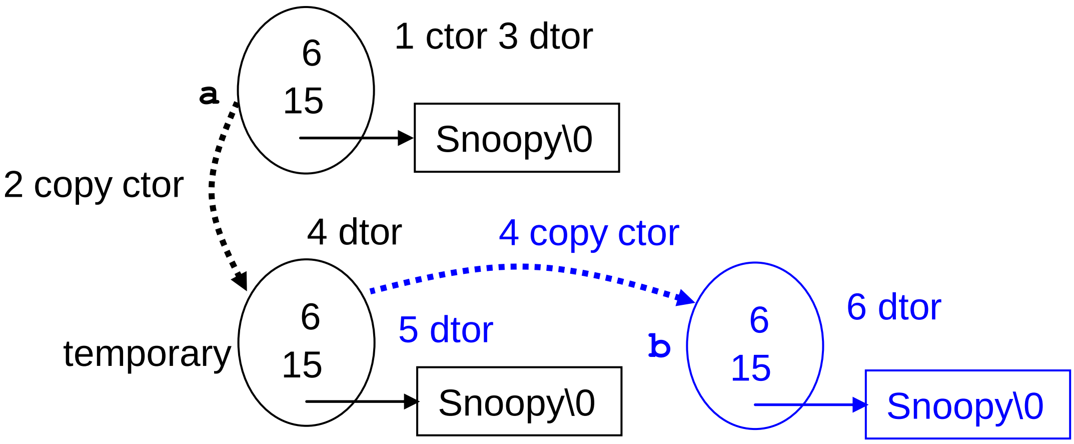
```
string::string(string&);             ①  copy ctor
string::string(const string&);       ②  copy ctor
string::string(string&&);            ③  move ctor
string::string(const string&&);      ④  move ctor
```

where ② and ③ are most often used.

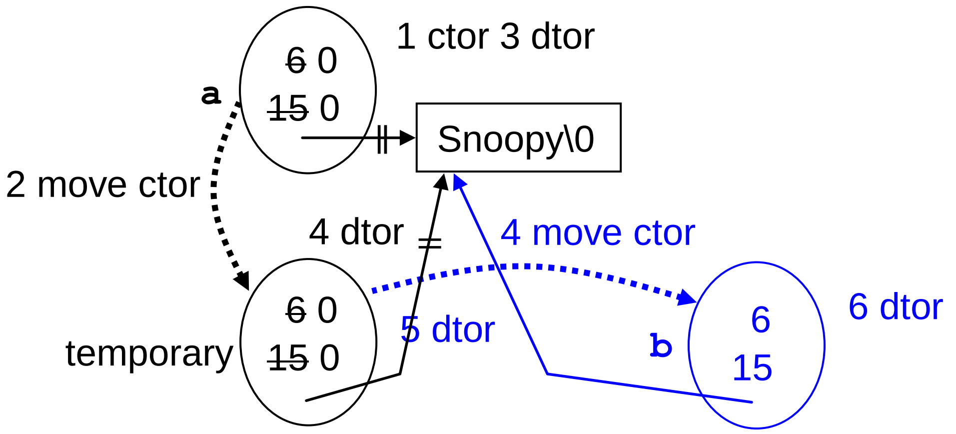
- Example B

```
string f() { string a("Snoopy"); return a; }
int main() { cout << f(); }          // black
int main() { string b(f()); }        // black + blue
```

Without move ctor



With move ctor



Comments

1 C++11 stipulates that the object **a** in the **return** statement is considered as an rvalue for the purpose of overload resolution in selecting a constructor.

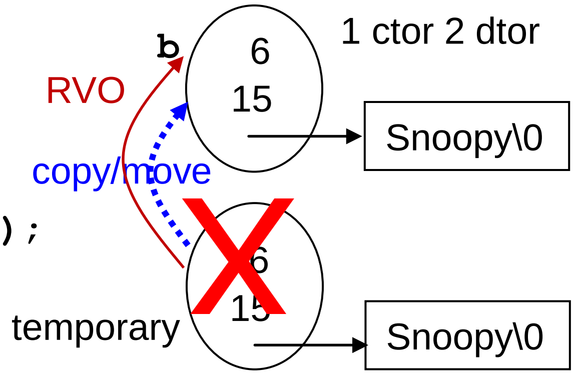
2 The behavior is the same if function **f** is written as
string f() { return string("Snoopy"); }
since the returned temporary is an rvalue.

3 To test the preceding code, compile it with no-elision option:
bsd2> **g++48 str.cpp -std=c++11 -fno-elide-constructors**

Copy elision

- Copy elision: the copy/move ctor may be elided by RVO.
- Copy elision is more efficient than resource stealing.
- Example A

```
void p(string b) {}
int main()
{
   p(string("Snoopy"));
}
```



With return value optimization (RVO), the temporary is eliminated and the copy/move ctor isn't called – the object is directly constructed in **b**. With RVO, the compiled code looks like

Callee

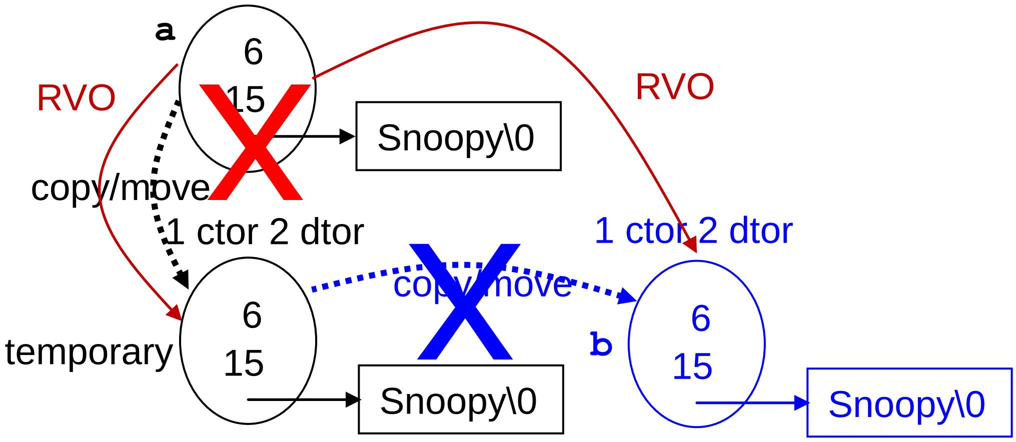
```
void p(string b) {}
⇒  void p() { string b("Snoopy"); }
```

Caller

```
p(string("Snoopy")); ⇒ p();
```

- Example B

```
string f() { string a("Snoopy"); return a; }
int main() { cout << f(); }      // red
int main() { string b(f()); }    // red + blue
```



- Example B (Cont'd)

  With RVO, the compiled code looks like

  `string f() { string a("Snoopy"); return a; }`
  ⇒
  Callee – *Let x be the caller-provided location*
  `string f(`*x***`) { string `***x***`("Snoopy");}`**

  Caller
  `cout << f();` ⇒ **`cout << f(`***a temporary location***`);`**
  `string b(f());` ⇒ **`f(`***location of* **`b);`**

- Even if copy elision elides the copy/move ctor. The semantics must be respected, i.e. the program must be executable in case of no copy elision.

- Example

```
class X {
public:
   X(int n) : p(new int(n)) {}
   ~X() { delete p; }
   X(const X& x) = delete;
private:
   int* p;
};
void p(X) {}
int main()
{
   p(X(3));       // error, no callable copy/move ctor
}
```

  Comment
  Since the user declares a copy ctor (even if it is deleted), the compiler won't generate a move ctor for class **X**.

Lvalue reference and rvalue reference

- Recall the syntax and semantics

  cv **`T&`** lvalue reference
  cv **`T&&`** rvalue reference (C++11)

  Except where explicitly noted, they are semantically equivalent (e.g. they must be initialized, the binding can't be altered, etc) and commonly referred to as references.

- Recall the example

```
int x=2;
int& y=x;           // int& y=2;     ✗
const int& z=2;
int&& z=2;          // int&& z=x;    ✗
```

Reference binding and overload resolution rules

- The table below summarizes the binding and resolution rules.

Expression / Reference type	`const T` rvalue	`T` rvalue	`const T` lvalue	`T` lvalue	Priority
`const T&`	O	O	O	O	low
`T&`				O	
`const T&&`	O	O			
`T&&`		O			high

Comments

1 Lvalues prefer lvalue references, whereas rvalues prefer rvalue references.

2 **`const T&&`**
This type is rare, but useful in certain cases. (See later.)

No-name rule

- Named rvalue references are treated as lvalues.
  Unnamed rvalue references are treated as rvalues (or, more precisely, xvalues).

  Comments

  1 rvalues = xvalues (eXpiring values) + prvalues (pure rvalues)
  2 An xvalue is the result of certain kinds of expressions involving rvalue references. (See next page)
  3 Value category in C++11: lvalue, xvalue, prvalue

- Example

```
string::string(const char*);      // ctor
string::string(string&&);         // move ctor
string::string(const string&);    // copy ctor

void q(string y) {}

void p(string&& x)         // x is bound to an rvalue
{
    q(x);                  // x is an lvalue; call copy ctor
    // x is visible here
}

p(string("Snoopy"));
q(string("Snoopy"));       // call move ctor, may be elided
```

  Comment

  Were **x** treated as an rvalue inside the function **p**, it would be moved to **y** by move constructor and the rest of the function would see a modified **x**, violating the guarantee that resource stealing doesn't visibly modify anything.

- Example

```
const int& x=2;
x++;              // error
int&& y=2;
y++;              // ok, y is a non-const lvalue
```

Moving from lvalues

- **std::move** is a function that turns its argument into an rvalue without doing anything else.
- Here is a simplified version of **move** that works only for lvalues:

```
template<typename T>
T&& move(T& a)
{
   return static_cast<T&&>(a);
}
```

  Comments on **static_cast<T>(***exp***)**

  1 If **T** is an lvalue reference type or an rvalue reference to function type, the result is an lvalue;
  2 If **T** is an rvalue reference to object type, the result is an xvalue;
  3 Otherwise, the result is a prvalue.

- Example (Cont'd)

  To steal the resource bound to **x**, we have to write

```
void p(string&& x)
{
   q(std::move(x));          // xvalue; call move ctor
}
```

  Put another way, **std::move(x)** is an unnamed rvalue reference, hence, by the no-name rule, it is an rvalue.

- Example

```
string s("Snoopy");
string t(s);                // call copy ctor
string u(std::move(s));     // call move ctor
```

  Even though **s** isn't a temporary, we may still steal its resource. This is sometimes useful. (See **std::swap** blow).

- Example

```
int s(2);
int t(s);                 // copy
int u(std::move(s));      // copy, too. move is redundant
s=2;                      // lvalue, ok
std::move(s)=2;           // rvalue, no
```

- Example

Assume that the **string** class adopts the implicitly-generated move ctor that does memberwise move.

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

Here, all the **move**'s are redundant.

- Example

```
template<typename T>
void std::swap(T& x,T& y)
{
   T z=std::move(x);
   x=std::move(y);
   y=std::move(z);
}
```

Comments

1 If **T** is a non-class type, **swap<T>** has no harm.
2 If **T** is a class type with callable move constructor and move assignment operator, say,

```
T::T(T&&);
T& T::operator=(T&&);
```

they will be invoked; otherwise, **T** shall have callable copy constructor and copy assignment operator, say,

```
T::T(const T&);              // must be const
T& T::operator=(const T&);   // must be const
```

- The simplified version of **`move`** can't be called on an rvalue, e.g.
  `move(2)`
  is illegal.
  However, although redundant, **`std::move`** actually works fine when called on an rvalue.

- Example

  ```
  int t(2);                 // copy
  int u(std::move(2));      // copy, too
  ```

- Here is the complete definition of **`move`**:

  ```
  template<typename T>
  typename remove_reference<T>::type&&
  std::move(T&& t)
  {
     typedef typename remove_reference<T>::type X;
     return static_cast<X&&>(t);
  }
  ```

  where **`remove_reference`** is defined as

  ```
  template<class T>
  struct remove_reference {
     typedef T type;
  };

  template<class T>
  struct remove_reference<T&> {
     typedef T type;
  };

  template<class T>
  struct remove_reference<T&&> {
     typedef T type;
  };
  ```

  For it to work, there are special reference collapsing rules and template argument deduction rule for **`T&&`**.

Reference collapsing rules

- Prior to C++11, a reference to a reference, i.e. **& &**, is illegal. In C++11, it is still illegal; but, compiler-generated references to references undergo reference collapsing:

```
T& &    ⇒ T&
T& &&   ⇒ T&
T&& &   ⇒ T&
T&& &&  ⇒ T&&
```

- Example

```
typedef string& lrs;
typedef string&& rrs;
string s("Snoopy");
string && & z=s;        // error, not generated
lrs& a=s;               // string& &   ⇒ string&
lrs&& b=s;              // string& &&  ⇒ string&
rrs& c=s;               // string&& &  ⇒ string&
rrs&& d=std::move(s);   // string&& && ⇒ string&&
```

Template argument deduction rule for **T&&**

- ```
  template<typename T>
  void foo(T&&);
  ```

  1 **foo** is called on an lvalue of type **A**
  T = A& ⇒ T&& = A& && = A&
  2 **foo** is called on an rvalue of type **A**
  T = A ⇒ T&& = A&&

- Universal reference
  Some people call **T&&** for some deduced type **T** a universal reference type, as it can bind lvalues and rvalues, const and non-const, etc.
  Syntax: **T&&**
  Semantics
  Retain the nature of its initializer, lvalue/rvalue, const/non-const etc. (e.g. given a const lvalue, it becomes an lvalue reference to a const object.)
  ```

- Example

```
// universal reference
auto&& x=2;
template<typename T>
typename remove_reference<T>::type&&  // rvalue ref
std::move(T&&);              // universal ref → rvalue ref
```

```
// rvalue reference
string::string(string&&);
template<typename T> stack<T>::push(T&&);
template<typename T> void foo(vector<T>&&);
template<typename T>
void foo(const T&&);      // not universal; only for const
```

- Example

```
void foo(string s) { std::move(s); }
void foo(string& s) { std::move(s); }
void foo(string&& s) { std::move(s); }
```

In type reduction, the reference part of a type is stripped off. Thus, the type of each **s** is **string**, then, because each **s** is an lvalue, the deduced type **T = string&**. Therefore, all end up with the same instantiation:

```
string&& move<string&>(string& a)  // T = string&
{
   return static_cast<string&&>(a);
}
```

Next, consider

```
std::move(string("Snoopy"))
```

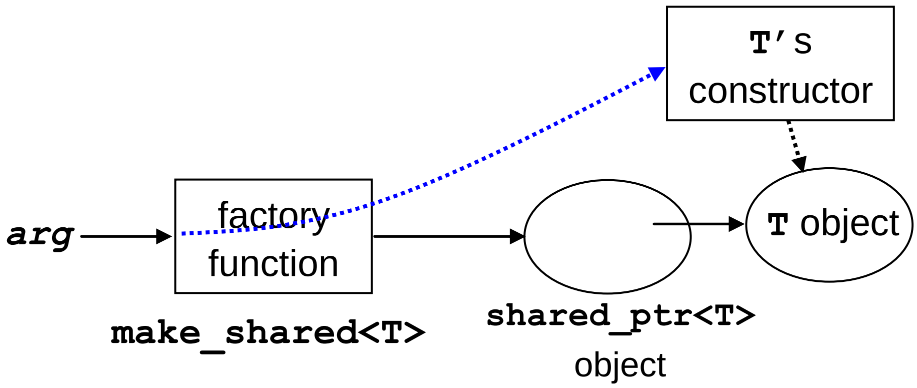
Since the argument is an rvalue, the deduced type **T = string** and the call ends up with the instantiation:

```
string&& move<string>(string&& a)  // T = string
{
   return static_cast<string&&>(a);
}
```

Perfect forwarding

- Perfect forwarding

  Consider the factory function **`make_shared<T>`** that forwards the argument ***`arg`*** to **`T`**'s constructor



  Perfect forwarding: As if the factory function weren't there and the constructor were called directly.

- Example

```
struct X {
   X(int&) {}                   // 1
   X(const int&) {}             // 2
};

int i=3;
shared_ptr<X>(new X(i))         // 1
shared_ptr<X>(new X(5))         // 2
```

  Approach 1

```
template<typename T,typename A>
shared_ptr<T> make_shared(A arg)    // by value
{
   return shared_ptr<T>(new T(arg));
}

make_shared<X>(i)               // copy + 1
make_shared<X>(5)               // copy + 1
```

  This approach needs an extra copy and invokes a different ctor when the argument is an rvalue.

- Example (Cont'd)

  Approach 2

```
template<typename T,typename A>          // A
shared_ptr<T> make_shared(A& arg)
{
   return shared_ptr<T>(new T(arg));
}

template<typename T,typename A>          // B
shared_ptr<T> make_shared(const A& arg)
{
   return shared_ptr<T>(new T(arg));
}
make_shared<X>(i)              // A + 1
make_shared<X>(5)              // B + 2
```

This approach has two problems.

Firstly, if there are several arguments, the factory functions have to be overloaded for all combinations of non-const and const references for the various arguments.

Secondly, it blocks out move semantics, since **arg** is an lvalue within the body of factory function.

Solution (by universal reference)

```
template<typename T,typename A>
shared_ptr<T> make_shared(A&& arg)
{
   return
   share_ptr<T>(new T(std::forward<A>(arg)));
}
```

where **std::forward** is defined as

```
template<class T>     // lvalue ref → universal ref
T&& forward(typename remove_reference<T>::type& t)
{
   return static_cast<T&&>(t);
}
```

- Example (Cont'd)

```
template<class T>      // rvalue ref → universal ref
T&& forward(typename remove_reference<T>::type&& t)
{
   return static_cast<T&&>(t);
}
```

Notice that the parameter **`T`** of **`forward`** must be specified and is used to determine the meaning of the universal reference.

Let's see how it works. First of all, since **`arg`** is an lvalue within the body of factory function, the first version of **`std::forward`** will be used in the sequel.

```
struct X {
   X(int&) {}
   X(const int&) {}
   X(int&&) {}
};
```

Case 1: **`X(int&)`**

```
int i=3;
make_shared<X>(i)                // 1 is desired
shared_ptr<X>(new X(i))          // 1
```

Since **`i`** is an lvalue, **`A = int&`** and the compiler will create

```
// A = int& ⇒ A&& = int& && = int&
shared_ptr<X> make_shared<X,int&>(int& arg)
{
   return
   share_ptr<X>(new X(std::forward<int&>(arg)));
}
```

```
// T = int& ⇒ T&& = int& && = int&
int& forward<int&>(int& t)
{
   return static_cast<int&>(t);
}
```

- Example (Cont'd)

  Perfect forwarding – the ctor **`X(int&)`** is invoked, as desired.

  Case 2: **`X(const int&)`**

```
const int j=4;
make_shared<X>(j)                // 2 is desired
shared_ptr<X>(new X(j))          // 2
```

  Since **`j`** is an lvalue, **`A = const int&`** and the compiler will create

```
// A = const int& ⇒ A&& = const int& && = const int&
shared_ptr<X>
make_shared<X,const int&>(const int& arg)
{
   return share_ptr<X>
          (new X(std::forward<const int&>(arg)));
}
```

```
// T = const int& ⇒ T&& = const int& && = const int&
const int& forward<const int&>(const int& t)
{
   return static_cast<const int&>(t);
}
```

  Perfect forwarding again – the ctor **`X(cost int&)`** is invoked, as desired.

  Case 3: **`X(int&&)`**

```
make_shared<X>(7)                // 3 is desired
shared_ptr<X>(new X(7))          // 3
```

  Since **`7`** is an rvalue, **`A = int`** and the compiler will create

```
// A = int ⇒ A&& = int&&
shared_ptr<X> make_shared<X,int>(int&& arg)
{
   return
   share_ptr<X>(new X(std::forward<int>(arg)));
}
```

- Example (Cont'd)

```
// T = int ⇒ T&& = int&&
int&& forward<int>(int& t)
{
   return static_cast<int&&>(t);
}
```

Perfect forwarding again – the ctor **X(int&&)** is invoked, as desired.

Finally, to invoke the second version of **std::forward**, try

```
new X(std::forward<int&>(6))        // X(int&)
new X(std::forward<const int&>(6))  // X(const int&)
new X(std::forward<int&&>(6))       // X(int&&)
```

Copy/move constructor (Part B)

- A non-template ctor for class **`X`** is a copy constructor 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:

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

```
string::string(string&,int);      // ctor; not copy ctor
string::string(string,int);       // ctor; 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
string::string(string,int=1);     // illegal
```

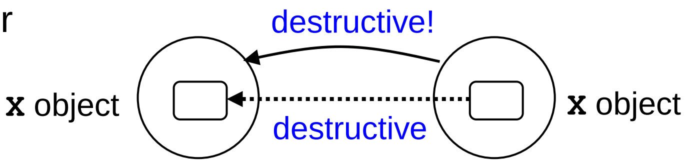
- A non-template ctor for class **`X`** is a move constructor if its first parameter is of type **`X&&`** (possibly cv-qualified) and all the other parameters (if any) have default arguments.

- If a class doesn't declare a copy ctor, one is implicitly declared as defaulted iff
  - there is no user-declared move ctor
  - there is no user-declared move assignment operator

  Such an implicit declaration is deprecated if
  - there is a user-declared copy assignment operator
  - there is a user-declared dtor

- If a class doesn't declare a move ctor, one is implicitly declared as defaulted iff
  - there is no user-declared copy ctor
  - there is no user-declared copy assignment operator
  - there is no user-declared move assignment operator
  - there is no user-declared dtor

- The implicitly-declared copy ctor for a class **`X`** is of type
  `X::X(const X&)` or
  `X::X(X&)`



- The implicitly-declared move ctor for a class **`X`** is of type
  `X::X(X&&)`

- The implicitly-defined copy/move ctor for a class performs a memberwise copy/move of its members.
  *Note: brace-or-equal-initializer*s of non-static data members are ignored.

- Moving (or move-assigning) object A to object B shall satisfy the following properties.

  1 The value of object B should be the same as the original value of object A.
  2 The value left in object A is unspecified.
  However, object A must remain valid, meaning that it must be destructible and may be manipulated in some ways that do not depend on its current value.
  N.B. Object A usually becomes empty, but not guaranteed.

- Example

  Recall the move ctor of **`string`** class

```
string::string(string&& rhs)          // move ctor
:   _size(rhs._size),
    _capacity(rhs._capacity),
    _data(rhs._data)
{
    rhs._size=rhs._capacity=0; rhs._data=nullptr;
}
```

- Example (Cont'd)

Case 1: the moved-from object is a temporary

`string a(string("Snoopy"));` // assume no copy elision

The temporary is safely destructed by the dtor
`string::string() { delete [] _data; }`
since **`delete []`** does nothing when **`_data`** is a null pointer.

Case 2: the moved-from object isn't a temporary

```
string a("Snoopy");
string b(std::move(a));
```

Correct use

```
a=b;                  // ok, assign a new value to object a
cout << a.size();     // ok
```

Incorrect use

`cout << a.size();`
This is undefined. Our move ctor will cause it to output 0.

`cout << a;`
This is undefined. Our move ctor will cause it to crash, in case the **`string`** output operation is implemented by
`cout << _data;`

Although undefined, we might not wish it to crash. To this end, there are two choices.

Choice 1
Implement the **`string`** output operation by
`cout << (_data!=nullptr? _data: "");`
This choice wastes time for non-moved-from objects.

Choice 2
Modify the move ctor by substituting an empty string
`rhs._data=&(*new char[1]='\0');`
for the empty pointer
`rhs._data=nullptr;`
This choice wastes time and space for moved-from objects.

- Digression: 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 T>
stack<T>::stack(const stack<T>& rhs)
```

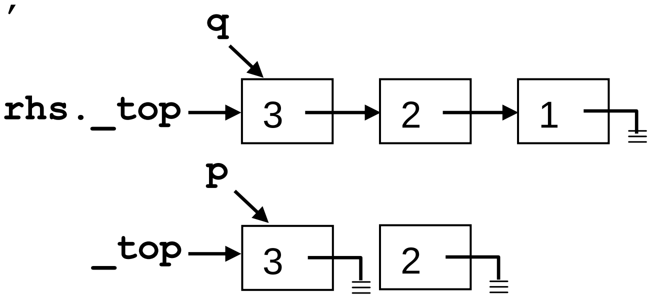
Outside class scope

1 Always use **X<***template-parameters***>**

```
   stack<string> s;
   pair<String,String>("Snoopy","Pluto")
```

- Example: Class template stack – linked-list representation

```
// copy ctor
template<typename T>
stack<T>::stack(const stack<T>& rhs)
:  _top(nullptr),sz(rhs.sz)
{
   if (rhs._top!=nullptr) {
      node* q=rhs._top;
      _top=new node(q->datum,nullptr);
      node* p=_top;
      while ((q=q->succ)!=nullptr) {
         p->succ=new node(q->datum,nullptr);
         p=p->succ;
      }
   }
}
```




```
// move ctor
template<typename T>
stack<T>::stack(stack<T>&& rhs)
:  _top(rhs._top),sz(rhs.sz)
{
   rhs._top=nullptr; rhs.sz=0;
}
```

- Example (Cont'd)

// push by copying

```
template<typename T>
void stack<T>::push(const value_type& val)
{
   _top=new node(val,_top);
   sz++;
}
```

// push by moving (C++11)

```
template<typename T>
void stack<T>::push(value_type&& val)
{
   _top=new node(std::move(val),_top);
   sz++;
}
```

// push by perfect forwarding (C++11)

```
template<typename T>
template<class... Args>
void stack<T>::emplace(Args&&... args)
{
   _top=new node(_top,std::forward<Args>(args)...);
   sz++;
}

template<typename T>
struct stack<T>::node {
   node(const T&,node*);
   node(T&&,node*);
   template<class...Args> node(node*,Args&&...);
   T datum;
   node* succ;
};

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

- Example (Cont'd)

```
template<typename T>
stack<T>::node::node(T&& d,node* s)
:  datum(std::move(d)),          // call T(T&&)
   succ(s)
   {}
```

```
template<typename T>
template<typename... Args>
stack<T>::node::node(node* s,Args&&... args)
:  datum(std::forward<Args>(args)...),
   succ(s)                        // call T(args...)
{}
```

For example,

```
stack<string> s;
string a("Snoopy"); s.push(a);  // ctor + copy
s.push(string("Snoopy"));        // ctor + move
s.emplace("Snoopy");             // ctor
```

```
stack<pair<string,string> > s;
pair<string,string> a("Snoopy","Pluto");
s.push(a);                       // (ctor + copy) × 2
s.push(pair<String,String>("Snoopy","Pluto"));
                                 // (ctor + move) × 2
s.emplace("Snoopy","Pluto");     // ctor × 2
```

- Example: Class template stack – sequential-array rep.

```
// copy ctor
template<typename T>
stack<T>::stack(const stack<T>& rhs)
:  stk((T*)operator new[](rhs.maxsz*sizeof(T))),
   _top(rhs._top),maxsz(rhs.maxsz)
{
   for (int i=0;i<=_top;i++)
      new (stk+i) T(rhs.stk[i]);
}
```

- Example (Cont'd)

```
// move ctor
template<typename T>
stack<T>::stack(stack<T>&& rhs)
:  stk(rhs.stk),_top(rhs._top),maxsz(rhs.maxsz)
{
   rhs.stk=nullptr;       // ok, won't print out this pointer
   rhs._top=-1;
   rhs.maxsz=0;
}

// push by copying
template<typename T>
void stack<T>::push(const value_type& val)
{
   ++_top;
   new (stk+_top) T(val);
}

// push by moving
template<typename T>
void stack<T>::push(value_type&& val)
{
   ++_top;
   new (stk+_top) T(std::move(val));
}

// push by perfect forwarding
template<typename T>
template<class... Args>
void stack<T>::emplace(Args&&... args)
{
   ++_top;
   new (stk+_top) T(std::forward<Args>(args)...);
}
```

Initialization

- Direct initialization

  `T x(a);`
  `T x{a};`

1	new	**`new T(a)`**, **`new T{a}`**
2	cast	**`T(a)`**, **`(T)a`**, **`static_cast<T>(a)`**
3	member initializer	**`U::U() : x(a),y{b} {}`**

- Copy initialization

  `T x=a;`

  1 parameter passing
  2 function return
  3 throwing/handling exception
  4 braced initialization list

- Direct initialization = copy initialization, unless class types are involved.

- Direct initialization: Call a constructor

- Copy initialization where **`a`**'s type = **`T`**
  1 call a copy/move constructor

- Copy initialization where **`a`**'s type ≠ **`T`**
  1 a temporary **`T`** object is created (may be elided by RVO)
  2 the initializer **`a`** is converted to the temporary **`T`** object by a user-defined conversion
  3 the object **`x`** being initialized is then direct-initialized from the temporary **`T`** 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.

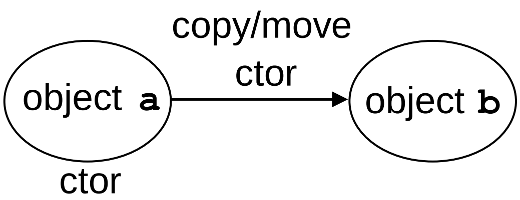
- Example

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

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

Direct initialization

```
string a("Snoopy");
string a{"Snoopy"};
string a({"Snoopy"});
string b(a);
string b(std::move(a));
```

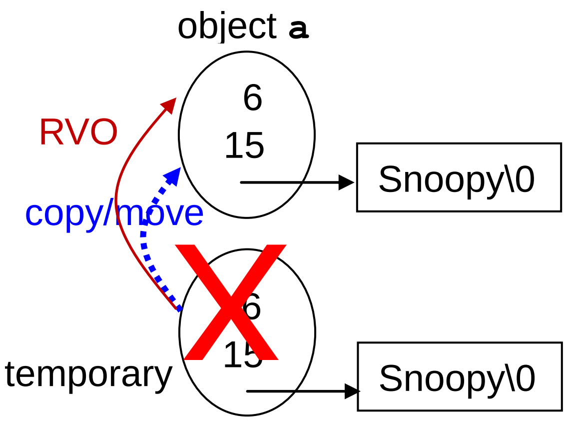


Copy initialization

```
string b=a;

string a="Snoopy";
string a={"Snoopy"};

void p(string a) {}
int main()
{
    p("Snoopy"); p({"Snoopy"});
}
```



Comments

1. For copy initialization, **`"Snoopy"`** has to be converted to a **`string`** object by the ctor.
   Making the conversion explicit obtains equivalent code, e.g.
   `p(string("Pluto"));`

2. With RVO, the copy/move ctor is elided. But, the semantics must be respected, i.e. a callable copy/move ctor must exist. For example, since a temporary is an rvalue, it is erroneous if the **`string`** class contains only the non-const copy ctor:
   `string::string(string&);`

Explicit constructor

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

```
class string {
public:
   explicit string(const char* ="");
};
void p(string);
void q(const string&);
void r(string&&);
```

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

Explicit constructors won't be used for implicit conversions.

	explicit	non-explicit
`string a("Snoopy");`	✓	✓
`string a="Snoopy";`	×	✓
`p("Snoopy")`	×	✓
`q("Snoopy")`	×	✓
`r("Snoopy")`	×	✓

Instead, explicit conversions are needed, e.g.

```
p(string("Snoopy"))
p((string)"Snoopy")
p(static_cast<string>("Snoopy"))
```

- An explicit copy/move ctor prevents objects from being passed or returned by value/move.
- Example

```
class string {
public:
   explicit string(const string&);
   explicit string(string&&);
};
```

- Example (Cont'd)

		explicit	non-explicit
`string a("Snoopy");`		–	–
`string b(a);`		✓	✓
`string b(std::move(a));`		✓	✓
`string b=a;`		×	✓
`string b=std::move(a);`		×	✓
`p(a)`		×	✓
`p(std::move(a))`		×	✓
`q(a)`	(don't care)	✓	✓
`r(std::move(a))`	(don't care)	✓	✓

- Example

```
struct o_O {
   o_O() { cout << 1; }
   o_O(const o_O&) { cout << 2; }
   o_O(o_O&&){ cout << 3; }
   ~o_O() { cout << 4; }
};

o_O f(o_O s) { return s; }

int main() { o_O oO; f(oO); }
int main() { f(o_O()); }
```

copy ctor	move ctor	`f(oO)`	`f(o_O())` with RVO
-	-	`123444`	`1344`
explicit deleted	-	Illegal call	`1344`
-	explicit deleted	`122444`	`1244`
explicit deleted	explicit deleted	Illegal call	Illegal call

- Prior to C++11, explicit constructors with multiple arguments are effectless, as they can't take part in implicit conversions. But, in C++11, they are no longer effectless.

- Example

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

	explicit	non-explicit
`string a{"Snoopy",3};`	✓	✓
`string a={"Snoopy",3};`	×	✓

- Principle

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

- Example

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

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

```
void p(const string&);
void p(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`** $\to$ **`stack`** so that the function

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

  can't be invoked by the call

```
p(7);            // must be invoked by the call p(stack(7));
```

User-defined conversion

- User-defined conversions

  1 Converting ctor

  `X::X(T...)` // `T...` → `X`

  A converting ctor is a ctor declared without the function specifier **`explicit`**.

  Note: Prior to C++11, it must have 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 specifie

- New in C++11
  An explicit conversion function can only be invoked in direct initialization, especially, in casts.

- Example

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

```
string a("Snoopy");
```

	explicit	non-explicit
`const char* s(a);`	✓	✓
`const char* s=a;`	×	✓
`strlen(a);`	×	✓

Similarly, explicit conversion functions won't be used for implicit conversions. Instead, explicit conversions are needed, e.g.

```
strlen(static_cast<const char*>(a));
strlen(a.operator const char*());
```

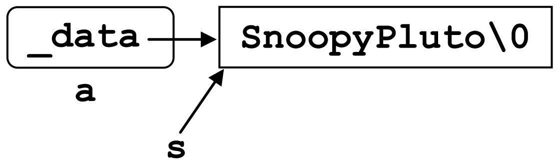
- Example (Cont'd)

Version A – Sharing

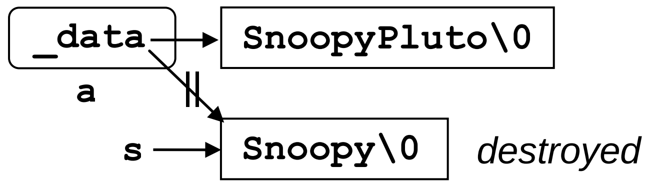
```
string::operator const char*() const;
{
   return _data;
}
string a("Snoopy");
const char* s(a);
a+="Pluto";
cout << s;                // SnoopyPluto or dangling pointer
```

Drawback – May become a dangling pointer

Case1: enough storage



Case 2: shy of storage



Version B – Copying

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

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

- Example (Cont'd)

  Remark

  The conversion **string** → **const char*** isn't as natural as the conversion **const char*** → **string**. Thus, the **string** class doesn't support **operator const char*()**, not even explicit.

  Instead, it provides **c_str()** and **data()**.

  c_str() and **data()** 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;
}
const char* string::data() 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);**
     Non-explicit **X::operator T();**

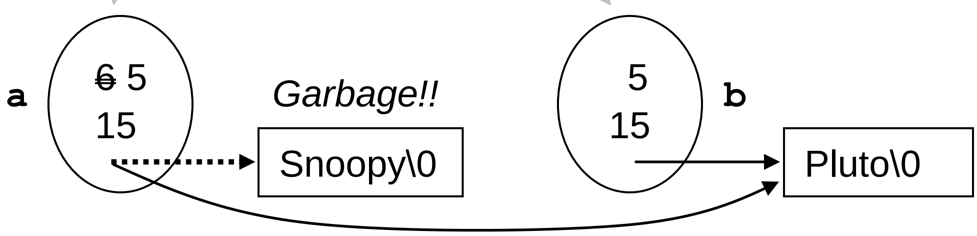
  2) Explicit **X** → **T** conversion is desired
     Explicit **T::T(X);**
     Explicit **X::operator T();**
     or
     X::toT();
     where **toT** is a member function performing the conversion.

Copy/move assignment operator

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

  Assume that the `s tring` class adopts the implicit copy/move assignment operator that does memberwise copy/move 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");
a=b;                           // a.operator=(b);
```




```
a=std::move(b)                 // a.operator=(std::move(b));

string& string::operator=(string&& rhs)
{
   if (this!=&rhs) {
      _size=std::move(rhs._size);
      _capacity=std::move(rhs._capacity);
      _data=std::move(rhs._data);
   }
   return *this;
}
```

- Principle

  Define a copy/move 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;
}

string& string::operator=(string&& rhs)
{
   if (this!=&rhs) {           // self-assignment
      delete [] _data;
      _size=rhs._size;
      _capacity=rhs._capacity;
      _data=rhs._data;
      rhs._size=rhs._capacity=0;
      rhs._data=nullptr;       // or, an empty string
   }
   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**
  i.e. **this==&rhs** within copy/move assignment operator
  3 Class independent, easy and efficiency

- Object equality (Value equality)

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

- Comments

  1 Object identity ⇒ object equality, but not *vice versa*
  2 Object identity is most often used.

- Example

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

Note: In STL, the last two cases are equal.

- For move assignment operator, the meaning of self-assignment under object identity is controversial.

```
a=std::move(d);        // a unchanged, d undefined – ok
a=std::move(a);        // a unchanged, a undefined – ?
```

In fact, the meaning of the last statement is undefined.

- Example – GNU C++

```
string a("Snoopy");
a=std::move(a);        // a unchanged; the same as ours
cout << a;             // Snoopy

vector<int> v(3,5);
v=std::move(v);        // v emptied
cout << v.size();      // 0
```

"Canonical implementation" of copy assignment

- On the parameter of copy/move assignment operator

  1. Unlike copy ctor, copy assignment operator may be passed by lvalue reference or value.
  2. Like move ctor, move assignment operator must be passed by rvalue reference.

- Canonical assignment: copy and swap
  A class that defines type-specific **swap** may use call-by-value to define its copy assignment operator.

- Example

```
void string::swap(string& rhs)
{
   using std::swap;
   swap(_size,rhs._size);
   swap(_capacity,rhs._capacity);
   swap(_data,rhs._data);
}

string& string::operator=(string rhs)  // by value
{
   swap(rhs);
   return *this;
}
```

Comments

1. This is copy assignment operator, rather than move assignment operator.
2. There is no need to detect self-assignment.
3. Having this copy assignment operator, the STL-style copy/move assignment operator shan't be defined, since call-by-value and call-by-reference will cause ambiguous calls.
4. Pro Easy to define
   Needn't define move assignment operator
   Con Inefficiency (See the comparisons below)

- Example (Cont'd)

  ```
  string a("Snoopy"),b("Pluto");
  a=b;
  ```

  Canonical copy assignment

1	copy **b**	// copy **b** to **rhs**
2	swap	// swap **a** and **rhs** (= **b**)
3	delete **a**	// delete **rhs** (= original **a**)

  C++11-style copy assignment

1	self-assignment test	
2	delete **a**	
3	copy **b**	// copy **b** to **a**

  Since swapping two strings takes 9 operations, the swap is more expensive than the self-assignment test.

  ```
  a=std::move(b);
  ```

  Canonical copy assignment

1	move **b**	// move **b** to **rhs**
2	swap	// swap **a** and **rhs** (= **b**)
3	delete **a**	// delete **rhs** (= original **a**)

  C++11-style move assignment

1	self-assignment test	
2	delete **a**	
3	move **b**	// move **b** to **a**

  Again, the swap is more expensive than the test.

- Example

  ```
  vector<string> a(9,"Snoopy"),b(8,"Pluto");
  a=b;
  ```

  The canonical copy assignment for **vector** class has to manage heap storage, since vector **b** is passed by value.
  But, the C++11-style copy assignment needn't, since vector **b** is passed by reference and vector **a** has enough capacity.

On the return type of copy/move assignment operator

- There is no constraint on the return type of copy/move assignment operator.
- Example

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

Q: Why do they return an lvalue reference to **`*this`**? rather than **`rhs`**?
A: To allow
`(a=b)=c;`
so that the semantics is consistent with built-in types.

Q: What about other return type?
A: All the other return types are inconsistent with built-in types.
For examples,

`void`	**`a=b=c;`**	disallowed
`string&&`	**`a=b=c;`**	**`b`** is moved to **`a`**
`const string&`	**`(a=b)=c;`**	disallowed
`string`	**`(a=b)=c;`**	**`c`** is copied to a temporary

Ref qualifier

- Observe that in the last case of the last example, the statement
`(a=b)=c;`
or
`(a.operator=(b)).operator=(c);` // *
is meaningless, as it modifies the rvalue **`a=b`** and then discards the modified value.
Observe also that in the starred line **`operator=`** is called on the lvalue **`a`** and the rvalue **`a=b`**.
- Prior to C++11, a member function can be called on lvalues and rvalues.
C++11 allows one to restrict a member function to be called on lvalues alone or rvalues alone.

- Example

```
struct X {
   void p() & {}                // 1
   void p() && {}               // 2
   void p() const & {}          // 3
};
X a;
const X& b=a;
a.p();                          // call 1
b.p();                          // call 3
X().p();                        // call 2
```

Comment

g++48 doesn't support ref qualifiers yet. So, test this program under **clang++**, say,
bsd2> **clang++ -std=c++11 ref.cpp**

- Member functions with the same name and the same parameter list can be overloaded only when all of them have a ref qualifier.

- Example

```
struct X {
   void p() & {}
   void p() && {}               // ok
   void p() {}                  // error
   void p(int) const {}         // ok, different parameter
};
```

More on copy/move assignment operator

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

- A move assignment operator **X::operator=** is a non-template member function with exactly one parameter of type **X&&** (possibly cv-qualified).

- If a class doesn't declare a copy assignment operator, one is implicitly declared as defaulted iff
  - there is no user-declared move ctor
  - there is no user-declared move assignment operator

  Such an implicit declaration is deprecated if
  - there is a user-declared copy ctor
  - there is a user-declared dtor

- If a class doesn't declare a move assignment operator, one is implicitly declared as defaulted iff
  - there is no user-declared copy ctor
  - there is no user-declared move ctor
  - there is no user-declared copy assignment operator
  - there is no user-declared dtor

- The implicitly-declared copy assignment operator for a class **`X`** is of type
  `X& X::operator=(const X&)` or
  `X& X::operator= (X&)`

- The implicitly-declared move assignment operator for a class **`X`** is of type
  `X::X(X&&)`

- The implicitly-defined copy/move assignment operator performs a memberwise copy/move assignment of its members.

- Example – Overloading **`operator=`**

```
string& string::operator=(const char*);  //*
string& string::operator=(char);
string& string::operator=(initializer_list<char>);
```

  Notice that these are ordinary assignment operators, rather than the special copy/move assignment operators.

  Also notice that without the starred version

```
a="pluto";        // equivalent to  a=string("pluto")
```

  can still be executed by move assignment operator, except that an extra temporary object will be created.

- Example

  Aiming to save that temporary, the starred version shall not be written as

```
string& string::operator=(const char* rhs)
{
   return (*this)=string(ths);
}
```

  Instead, it shall be written as

```
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;
}
```

  Notice that

```
a=a.c_str();
```

  is a self-assignment and shall be detected.

- Example – Heap array initialization

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

- Example (Cont'd)

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

```
template<typename T>
vector<T>::~vector()
{
   for (int i=_size-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
}
```

// 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]())           // value initialization
{
   for (int i=0;i<n;i++)
      _data[i]=val             // copy assignment
}
```

- Example (Cont'd)

// Version C – warning

```
template<typename T>
vector<T>::vector(size_type n,const T& val)
:  _size(n),_capacity(n),
   _data(new T[n]())              // value initialization
{
   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");`

Finally, default ctor: Q: Why don't we write `new T[n]()`?

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

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 can’t be altered.
- **`:: ?: . .*`** cannot be overloaded
- Example – **`operator+=`** as a member function

  `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 can’t be done too earlier for self-appending to work.

- Example (Cont'd)

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

- Example – **operator+** as a member function

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 of the **string** class.

Remarks

1 For other classes, **operator+** may be a member function. For example,

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

2 As a member function, **a+b** ≡ **a.operator+(b)**
As a non-member function, **a+b** ≡ **operator+(a,b)**

- Example – **`operator+`** as a member function

  Move semantics: string addition

  Motivation

```
string s,t,u,v;
s+t                 // lvalue + lvalue
((s+t)+u)+v         // rvalue + lvalue
s+(t+(u+v))         // lvalue + rvalue
(s+t)+(u+v)         // rvalue + rvalue
```

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

Recall that RVO applies to the **`return`** statement

`return` *name*; // *name* isn't a function parameter

as follows:

Callee – *Let x be the caller-provided location*

```
string operator+(lhs,rhs,x)
{
   ~~string s(lhs); s+=rhs; return s;~~
   string x(lhs); x+=rhs;
}
```

Caller

`s+t` ⇒ **`operator+(s,t,`***a temporary location***`)`**

Notice that the *name* must not be a function parameter:

```
string operator+(lhs,rhs,x)
{
   ~~string s(lhs); s+=rhs; return lhs;~~
   string s(x); s+=rhs;          // no
}
```

Recall also that in the absence of RVO, the object **`s`** may be moved by the move ctor, since it is considered as an rvalue for the purpose of overload resolution in selecting a constructor.

- Example (Cont'd)

Comment – Compare the following three definitions

```
string s(lhs); s+=rhs; return s;     // copy + elision
string s(lhs); return s+=rhs;        // copy + copy
return string(lhs)+=rhs;             // copy + copy
```

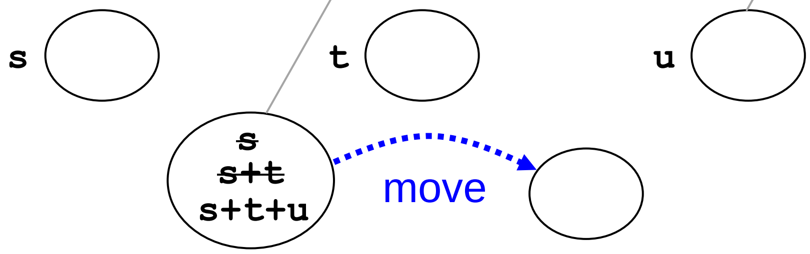
For the last two definitions, neither RVO (∵ not **`return`** *name*) nor move ctor (∵ **`+=`** yield an lvalue that isn't considered as an rvalue) applies. To invoke the move ctor, call **`std::move`**

```
string s(lhs); return std::move(s+=rhs);
return std::move(string(lhs)+=rhs);
```

```
// rvalue + lvalue
string operator+(string&& lhs,const string& rhs)
{
   return std::move(lhs+=rhs);
}
```

Comment

This STL function is inefficient. To see why, consider **`(s+t)+u`**



temporary **`s+t`**   temporary **`(s+t)+u`**

The move clearly is redundant, since the computation is done and the temporary **`s+t`** is alive in the caller side. To save this move, we need only change the return type to **`string&&`**:

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

- Example (Cont'd)

```
// lvalue + rvalue
string operator+(const string& lhs,string&& rhs)
{
   return std::move(rhs.insert(0,lhs));
}
```

```
// rvalue + rvalue
string operator+(string&& lhs,string&& rhs)
{
   return std::move(lhs+=rhs);
// return std::move(rhs.insert(0,lhs));
}
```

Again, for efficiency reason, the return type of the last two functions should be changed to **string&&**.

Other overloaded **operator+**

```
// lvalue/rvalue + C, C + lvalue/rvalue
string operator+(const string&,const char*);
string operator+(string&&,const char*);
string operator+(const char*,const string&);
string operator+(const char*,string&&);
```

Overload resolution

```
string a("Snoopy"),b("Pluto");
a+b
```

Viable	lvalue + lvalue	
Best viable	lvalue + lvalue	

a+"Pluto"

Viable	lvalue + lvalue	identity + array-2-pointer, user-defined conversion
	lvalue + rvalue	*same as above*
	lvalue + C	identity + array-2-pointer
Best viable	lvalue + C	

- Example (Cont'd)

  `a+string("Pluto")`

Viable	lvalue + lvalue, lvalue + rvalue
Best viable	lvalue + rvalue

  `operator+("Snoopy","Pluto")`

Viable	All of them
Best viable	rvalue + C, C + rvalue Ambiguous!

  `"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.

  ### Preventing **`a+b=c`**

  The STL **`string`** class permits the following code, which is inconsistent with built-in types.

```
string a("Snoopy"),b("Pluto"),c("Garfield");
a+b=c;            // modify a temporary object
```

  There are two ways to prevent such assignment statements.

  Method 1

  Declare all **`operator=`**'s with **`&`** ref-qualifier, e.g.

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

  Method 2

  Declare the return type of **`operator+`**'s with **`const`** qualifier, e.g.

```
const string operator+(const string&,const string&);
const string operator+(string&&,const string&);
```
  or
```
const string&& operator+(string&&,const string&);
```

  Observe that each of the last two prevents the assignment

```
string("Snoopy")+b=c;
```

- Example

  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);
```

  Since **cin** and **cout** are used to be the left operand for built-in types, **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 rep.)

```
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!=nullptr) {
      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

- 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;
};

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

- Example (Cont'd)

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

Version 3

```
class node;                     // class declaration

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

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

// definitions of **stack::pop()** and **stack::top()** omitted

Friends of class templates

- There are three kinds of friends of class templates:
  1 Nontemplate friends
  2 Bound friend templates
  3 Unbound friend templates

- Example – **`operator<<`** as a friend of class template **`stack`**

```
using OS=ostream;          // type alias
typedef ostream OS;        // the same
```

Nontemplate friends

```
template<typename T>
class stack {
public:
friend OS& operator<<(OS&,const stack<int>&);
// other declarations omitted
};
```

This means that the ordinary operator function
`OS& operator<<(OS&,const stack<int>&)`
is a friend of the instantiated class
`stack<int>`.

```
OS& operator<<(OS& os,const stack<int>& s)
{
   stack<int>::node* p=s._top;
  // code omitted
}

OS& operator<<(OS& os,const stack<char>& s)
{
   stack<char>::node* p=s._top;      // no, not friend
  // code omitted
}

stack<int> s; cout << s;             // ok
stack<char> t; cout << t;            // no
```

- Example (Cont'd)

```
template<typename T>
class stack {
public:
friend OS& operator<<(OS&,const stack<T>&);
// other declarations omitted
};
```

This means that, for all **T**, the ordinary operator function **OS& operator<<(OS&,const stack<T>&)** is a friend of the instantiated class **stack<T>**.

```
OS& operator<<(OS& os,const stack<int>& s)
{
   stack<int>::node* p=s._top;
   // code omitted
}

OS& operator<<(OS& os,const stack<char>& s)
{
   stack<char>::node* p=s._top;      // ok, friend
   // code omitted
}
stack<int> s; cout << s;             // ok
stack<char> t; cout << t;            // ok
stack<float> u; cout << u;           // no
```

Bound template friends

```
template<typename T> class stack;

template<typename T>
OS& operator<<(OS&,const stack<T>&);

template<typename T>
class stack {
public:
friend OS& operator<< <T>(OS&,const stack<T>&);
// other declarations omitted
};
```

- Example (Cont'd)

```
template<typename T>
ostream& operator<<(ostream& os,const stack<T>& s)
{
   typename stack<T>::node* p=s._top;
   // code omitted
};
```

This means that, for all **T**, the instantiated operator function
OS& operator<< <T>(OS&,const stack<T>&)
is a friend of the instantiated class
stack<T>.

```
stack<int> s; cout << s;            // ok
stack<char> t; cout << t;           // ok
stack<float> u; cout << u;          // ok
```

Unbound template friends

```
template<typename T> class stack;

template<typename T>
OS& operator<<(OS&,const stack<T>&);

template<typename T>
class stack {
public:
template<typename U>
friend OS& operator<<(OS&,const stack<U>&);
// other declarations omitted
};

template<typename U>
ostream& operator<<(ostream& os,const stack<U>& s)
{
   typename stack<U>::node* p=s._top;
   // code omitted
};
```

- Example (Cont'd)

  This means that, for all **`T`**, the operator function template

```
template<typename U>
OS& operator<<(OS&,const stack<U>&)
```

  is a friend of the instantiated class
  `stack<T>`.

  In particular, the instantiated operator function

```
OS& operator<< <T>(OS&,const stack<T>&)
```

  is a friend of the instantiated class
  `stack<T>`.

```
stack<int> s; cout << s;            // ok
stack<char> t; cout << t;           // ok
stack<float> u; cout << u;          // ok
```

  Since the operator function template is a friend of **`stack<T>`** for all **`T`**, we may write something like

```
// print out the values of type U of stack s from bottom to top
// after converting them to bool values
template<typename U>
OS& operator<<(OS& os,const stack<U>& s)
{
   typename stack<U>::node* p=s._top;
   stack<bool> t;
   while (p!=nullptr) {
      t.push(p->datum);          // convert to bool
      p=p->succ;
   }
   stack<bool>::node* q=t._top;     //*
   while (q!=nullptr) {             //*
      os << q->datum; q=q->succ;    //*
   }                                //*
   return os;
}
```

  But, the code in the starred lines is illegal for bound template friends.

- Example – **`stack`** as a friend of class template **`node`**

```
template<typename T>
class stack {
    // other declarations omitted
private:
    class node;
    // other declarations omitted
};
```

Nontemplate friends

```
template<typename T>
class stack<T>::node {
    friend class stack<int>;
    // other declarations omitted
};
stack<int> s;           // ok
stack<char> s;          // no
```

Bound template friends

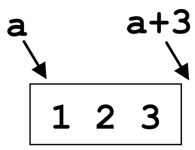
```
template<typename T>
class stack<T>::node {
    friend class stack<T>; // friend class stack;
    // other declarations omitted
};
stack<int> s;           // ok
stack<char> s;          // ok
```

Unbound template friends

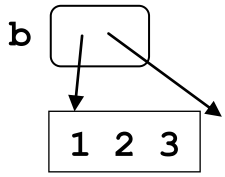
```
template<typename T>
class stack<T>::node {
    template<typename U>
    friend class stack;
    // other declarations omitted
};
stack<int> s;           // ok
stack<char> s;          // ok
```

Iterator

- Motivation



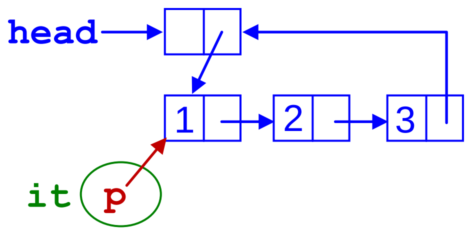

```
int a[3]{1,2,3};
for (int* it=a;it!=a+3;++it)
   cout << *it;
⇒ for (int* it=begin(a);it!=end(a);++it)
      cout << *it
⇒ for (int x : a) cout << x;
```




```
initializer_list<int> b{1,2,3};
for (const int* it=b.begin();it!=b.end();++it)
   cout << *it;
⇒ for (const int* it=begin(b);it!=end(b);++it)
      cout << *it;
⇒ for (int x : b) cout << x;
```

Next, consider traversing a singly-linked list with a header node

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



Arrays, initializer lists, and linked lists are all **sequences**. It is desirable that they can be traversed in a uniform manner.

```
#include <forward_list>        // C++11
forward_list<int> c(begin(a),end(a));
using iter=forward_list<int>::iterator;
for (iter it=c.begin();it!=c.end();++it)
   cout << *it;
⇒ for (iter it=begin(c);it!=end(c);++it)
      cout << *it;
⇒ for (int x : c) cout << x;
```

- Motivation (Cont'd)

  The uniform traversal of sequences permits generic algorithms.

```
template<typename iterator>
void print(iterator first,iterator last)
{
   for (iterator it=first;it!=last;++it)
      cout << *it;
}

print(begin(a),end(a));    // iterator = int*
print(begin(b),end(b));    // iterator = const int*
print(begin(c),end(c));    // iterator = iter
```

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

- 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. **`typedef const T* iterator;`**

Forward	**`forward_list`**
Bidirectional	**`list`**
Random access	**`initializer_list`**
	`array, vector, deque`,
	`string`

`typedef char* iterator;` **`typedef T* iterator;`**

- STL has 3 components.

  1 Containers
  2 Generic algorithms
  3 Iterators
  Iterators are the glue that holds containers and generic algorithms together.

- Example

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

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator,BidirectionalIterator);
```

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

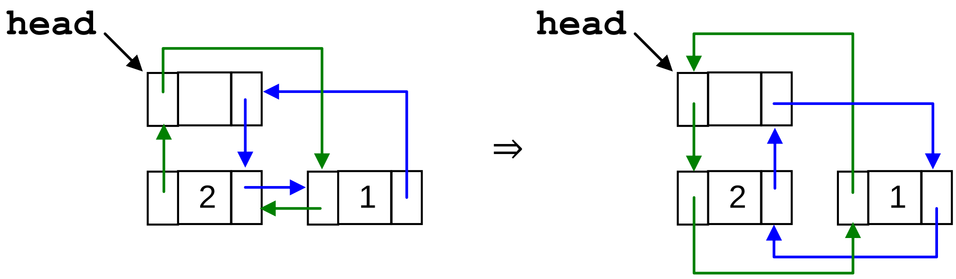
- Example

```
#include <iostream>
#include <string>
#include <array>              // for array
#include <list>               // doubly-linked list
#include <algorithm>          // for sort, reverse
#include <numeric>            // for accumulate
using namespace std;

int main()
{
   string a("pluto");
   sort(begin(a),end(a));
   cout << a;                               // loptu
   array<int,5> b{3,1,5,4,2};
   cout << accumulate(begin(b),end(b),0);  // 15
   list<int> c(begin(b),end(b));
   reverse(begin(c),end(c));
   for (int x : c) cout << x;              // 24513
   c.sort();
   for (int x : c) cout << x;              // 12345
}
```

Comment

1 Sorting a list can be done by pointer adjustment.



2 `std::array` is a container for constant-size arrays.

```
template<class T,size_t N>
struct array {
    // other members omitted
    T _elem[N];
};
```

- Example – Doubly-linked list

```
template<class T>
class list {
public:
// types
   class iterator;
   class const_iterator;
   typedef T& reference;
   typedef const T& const_reference;
   typedef T* pointer;
   typedef const T* const_pointer;
   typedef size_t size_type;
// ctor/dtor
   list();
   list(const list<T>&);
   list(list<T>&&);
   list(initializer_list<T>);
   template<class InputIterator>
   list(InputIterator,InputIterator);
   ~list();
// modifiers
   iterator insert(const_iterator,const T&);
   iterator insert(const_iterator,T&&);
   iterator erase(const_iterator);
   void push_front(const T&);
   void push_front(T&&);
   void push_back(const T&);
   void push_back(T&&);
   void pop_front();
   void pop_back();
   void swap(list<T>&);
// list operation
   void remove(const T&);
// capacity
   bool empty() const { return begin()==end(); }
   size_type size() const { return sz; }
```

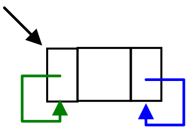
- Example (Cont'd)

// element access
```
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
```
// iterator
```
    iterator begin();
    const_iterator begin() const;
    const_iterator cbegin() const;
    iterator end();
    const_iterator end() const;
    const_iterator cend() const;
private:
    struct node;
    node* head;
    size_type sz;
};

template<class T>
struct list<T>::node {
    T datum;
    node *pred,*succ;
    node(const T& d,node* p,node* s)
    :  datum(d),pred(p),succ(s)
    {}
    node(T&& d,node* p,node* s)
    :  datum(std::move(d)),pred(p),succ(s)
    {}
};
```

ctor/dtor

head



```
template<class T>
list<T>::list()
:  head((node*)operator new(sizeof(node)))
{
    head->pred=head->succ=head;
}
```

- Example (Cont'd)

```
template<class T>
template<class InputIterator>
list<T>::list(InputIterator first,InputIterator last)
:  list()
{
   while (first!=last) {
      push_back(*first); ++first;
   }
};
```

```
template<class T>
list<T>::list(initializer_list<T> a)  // must
:  list(a.begin(),a.end()) {}
```

Comment

```
:  list(std::begin(a),std::end(a)) {} // ok
:  list(begin(a),end(a)) {}           // no
```

The latter is erroneous, since **begin** is found within the class scope as **list<T>::begin()**

```
template<class T>
list<T>::list(const list<T>& rhs)      // optional
:  list(rhs.begin(),rhs.end())
{}
```

```
template<class T>
list<T>::list(list<T>&& rhs)           // optional
:  list()
{
   swap(rhs);
}
```

```
template<class T>
list<T>::~list()
{
   while (!empty()) pop_back();
   operator delete(head);
}
```

STL `<itearor>`: Part A

Iterator category

- In STL, *category tag* classes are introduced for the five different kinds of iterators.

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag
:  public input_iterator_tag {};
struct bidirectional_iterator_tag
:  public forward_iterator_tag {};
struct random_access_iterator_tag
:  public bidirectional_iterator_tag {};
```

Iterator traits

- Each iterator class has to define 5 type traits:

```
iterator_category
value_type
difference_type
pointer
reference
```

- In STL, the **`std::iterator`** class template is introduced to ease the definition of type traits for new iterator classes.

```
template<class Category,
         class T,
         class Distance=ptrdiff_t,
         class Pointer=T*,
         class Reference=T&>
struct iterator {
   typedef Category    iterator_category;
   typedef T           value_type;
   typedef Distance    difference_type;
   typedef Pointer     pointer;
   typedef Reference   reference;
};
```

- Example (Cont'd)

```
template<class T>
class list<T>::iterator
:  public
   std::iterator<bidirectional_iterator_tag,T>
{ … }
```

- The **iterator_traits** class template is used to find out the type traits of an iterator.

```
template<class Iterator>
struct iterator_traits {
   typedef typename Iterator::iterator_category
                              iterator_category;
   typedef typename Iterator::value_type
                              value_type;
   typedef typename Iterator::difference_type
                              difference_type;
   typedef typename Iterator::pointer
                              pointer;
   typedef typename Iterator::reference
                              reference;
};
```

```
// Specialization for pointer types
template<class T>
struct iterator_traits<T*> {
   typedef random_access_iterator_tag
                          iterator_category;
   typedef T              value_type;
   typedef ptrdiff_t      difference_type;
   typedef T*             pointer;
   typedef T&             reference;
};
```

- Example (Cont'd)

  Bidirectional iterators

  Recall that bidirectional iterators support the following operations:

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

  Iterator type pointing to **T** (analogous to **T***)

```
template<class T>
class list<T>::iterator
:  public std::iterator<bidirectional_iterator_tag,T> {
public:
   iterator(node* cur=0) : current(cur) {}
   reference operator*() const;
   pointer operator->() const;
   iterator& operator++();
   iterator operator++(int);
   iterator& operator--();
   iterator operator--(int);
   bool operator==(iterator it) const;
   {
      return current==rhs.current;
   }
   bool operator!=(iterator it) const;
   {
      return current!=rhs.current;
   }
private:
   node* current;
};
```

- Example (Cont'd)

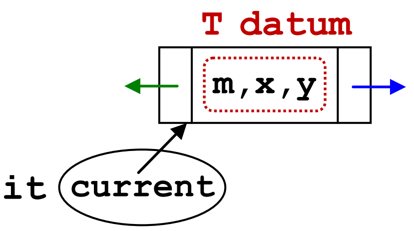
Required precondition: the iterator is *dereferenceable*
Note: An iterator `it` is *dereferenceable*, if `*it` is defined. That is, if `it.current` doesn't point to a header node.

```
template<class T>
typename list<T>::reference
list<T>::iterator::operator*() const
{
   return current->datum;
}
```

```
template<class T>
typename list<T>::pointer
list<T>::iterator::operator->() const
{
   return &operator*();      // or, &**this
}                            // or, &current->datum;
```

Remark

1 For an object `it` of class type
`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.

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

- Example (Cont'd)

```
template<class T>
typename list<T>::iterator&
list<T>::iterator::operator++()        // prefix++
{
   current=current->succ; return *this;
}

template<class T>
typename list<T>::iterator&
list<T>::iterator::operator++()        // prefix--
{
   current=current->pred; return *this;
}

template<class T>
typename list<T>::iterator
list<T>::iterator::operator++(int)     // postfix++
{
   iterator old=*this;  // iterator old(current);
                        // node* old=current;
   ++*this;             // current=current->succ;
   return old;
}

template<class T>
typename list<T>::iterator
list<T>::iterator::operator--(int)     // postfix--
{
   iterator old=*this;
   --*this;
   return old;
}
```

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

- Example (Cont'd)

The postfix **++** and **--** operators 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**.

Preventing it++++

In STL, the following code is permitted, which is inconsistent with built-in types.

```
list<string> a{"Snoopy","Pluto","Garfield"};
auto it=a.begin();
it++++;
cout << *it;              // Pluto
```

Notice that the second **++** doesn't modify the object **it**. Instead, it modifies the temporary produced by **it++**.

Again, there are two ways to prevent it.

Method 1

Declare the postfix **operator++** with **&** ref-qualifier, e.g.

```
template<class T>
typename list<T>::iterator
list<T>::iterator::operator++(int) &;
```

Method 2

Declare the return type of the postfix **operator++** with **const** qualifier:

```
template<class T>
const typename list<T>::iterator
list<T>::iterator::operator++(int);
```

- Example (Cont'd)

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

```
template<class T>
class list<T>::const_iterator
:  public std::iterator<bidirectional_iterator_tag,T> {
public:
   const_iterator(const node* cur=0)
   :  current(cur) {}
   const_reference operator*() const;
   const_pointer operator->() const;
   const_iterator& operator++();
   const_iterator operator++(int);
   const_iterator& operator--();
   const_iterator operator--(int);
   bool operator==(const_iterator) const;
   bool operator!=(const_iterator) const;
private:
   const node* current;
};
```

  This class can be defined in a similar way and is left to you.

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

  e.g.

```
list<int> b{1,2,3};
list<int>::const_iterator cit=b.begin();
```

  Method A – Friend + Converting ctor

```
class list<T>::iterator
:  public std::iterator … {        // define first
public:
   friend class const_iterator; // no <T> here
   // other members remain unchanged   // ∵ it isn't a template
};
```

- Example (Cont'd)

```
template<class T>
class list<T>::const_iterator
:  public std::iterator … {
public:
   const_iterator(iterator it)
   :  current(it.current) // node* → const node*
   {}
   // other members remain unchanged
};
```

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

Method B – Type conversion operator

```
template<class T>
class list<T>::const_iterator
:  public std::iterator …              // define first
{…}

template<class T>
class list<T>::iterator
:  public std::iterator … {
public:
   operator const_iterator() const
   {
      return current;      // node* → const node*
   }
   // 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.

- Example (Cont'd)

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

e.g.

```
const list<int> b{1,2,3};
list<int>::iterator it
                = list<int>::iterator(b.begin());
```

or,

```
 = static_cast<list<int>::iterator>(b.begin());
```

Note: STL `list` doesn't support this conversion.

Method A – Explicit type conversion operator

```
template<class T>
class list<T>::iterator
:  public std::iterator …              // define first
{…}

template<class T>
class list<T>::const_iterator
:  public std::iterator … {
public:
   explicit operator iterator() const
   {
      return const_cast<node*>(current);
   }                          // const node* → node*
   // other members remain unchanged
}
```

Method B – Friend + Explicit ctor

```
template<class T>
class list<T>::const_iterator
:  public std::iterator … {            // define first
public:
   friend class iterator;              // no <T> here
   // other members remain unchanged
};
```

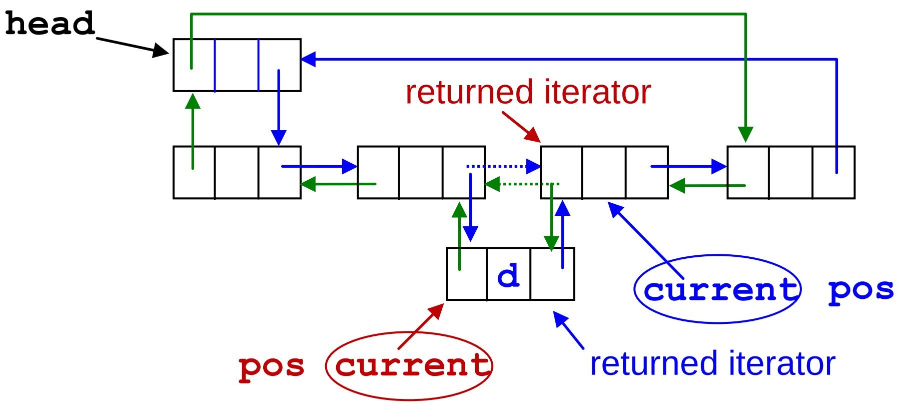
- Example (Cont'd)

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

Modifiers – insert and erase

```
template<class T>
typename list<T>::iterator
list<T>::insert (const_iterator pos,const T& d);
list<T>::insert (const_iterator pos,T&& 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




```
template<class T>
typename list<T>::iterator
list<T>::erase(const_iterator pos);
```

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

- Example (Cont'd)

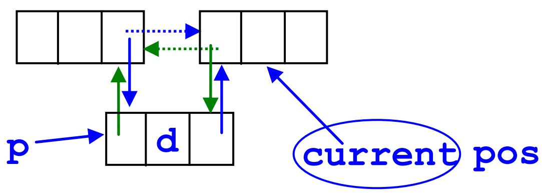
  1 erase the node pointed to by the iterator **`pos`**
  2 return an iterator pointing to the node immediately following the node just erased

How can **`insert`** and **`erase`** of class **`list<T>`** access the private member **`current`** of class **`const_iterator`**?

Method A – Friend

```
template<class T>
class list<T>::const_iterator
:  public std::iterator… {
   friend class list<T>;            // optional <T> here
   // other members remain unchanged
};
```

```
template<class T>
typename list<T>::iterator
list<T>::insert(const_iterator pos,const T& d)
{
   node* pos_current=(node*)pos.current;
   node* p=new node(d,pos.current->pred,pos_current);
   pos.current->pred->succ=p;    // pos_current ✓
   pos_current->pred=p;          // pos.current ✘
   sz++;
   return p;
}
```



Comment

1 Prior to C++11, **`pos`** is of the type **`iterator`**.

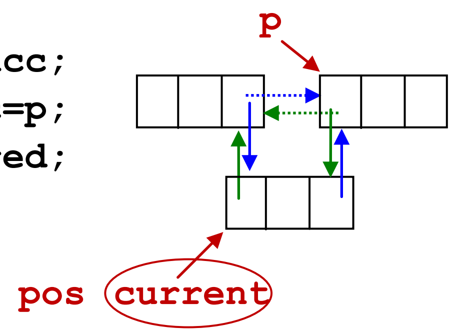
2 Expression / Type

Expression	Type
`pos`	`const_iterator`
`pos.current`	`const node*`
`pos.current->pred`	`node*const`
`pos.current->pred->succ`	`node*`

3 For the rvalue version, replace d by **`std::move(d)`**.

- Example (Cont'd)

```
template<class T>
typename list<T>::iterator
list<T>::erase(const_iterator pos)
{
   node* p=pos.current->succ;
   pos.current->pred->succ=p;
   p->pred=pos.current->pred;
   delete pos.current;
   sz--;
   return p;
}
```



Method B – Type conversion operator

```
template<class T>
class list<T>::const_iterator
:  public std::iterator… {
public:
   operator const node*() const { return current; }
   // other members remain unchanged
};
```

```
template<class T>
typename list<T>::iterator
list<T>::insert(const_iterator pos,const T& d)
{
   const node* pos_current=pos;
   node* p=new node
      (d,pos_current->pred,(node*)pos_current);
   pos_current->pred->succ=p;
   ((node*)pos_current)->pred=p;    // 1
   sz++;
   return p;
}
```

1 or, `(node*&)(pos_current->pred)=p;`

2 For the rvalue version, replace d by `std::move(d)`.

- Example (Cont'd)

```
template<class T>
typename list<T>::iterator
list<T>::erase(const_iterator pos)
{
   const node* pos_current=pos;
   node* p=pos_current->succ;
   pos_current->pred->succ=p;
   p->pred=pos_current->pred;
   delete pos;                // or, pos_current
   sz--;
   return p;
}
```

Modifiers – Insert and erase at two ends

```
template<class T>
void list<T>::push_front(const T& d)
{  insert(cbegin(),d); }

template<class T>
void list<T>::push_front(T&& d)
{  insert(cbegin(),std::move(d)); }

template<class T>
void list<T>::push_back(const T& d)
{  insert(cend(),d); }

template<class T>
void list<T>::push_back(T&& d)
{  insert(cend(),std::move(d)); }

template<class T>
void list<T>::pop_front()
{  erase(cbegin()); }

template<class T>
void list<T>::pop_back()
{  erase(--cend()); }
```

- Example (Cont'd)

  Modifier – swap

```
void swap(list<T>& x)
{
   std::swap(head,x.head);
   std::swap(sz,x.sz);
}
```

  List operation

```
template<class T>
void list<T>::remove(const T& value)
{
   for (const_iterator it=begin();it!=end();)
      if (*it==value) it=erase(it);
      else ++it;
}
```

  Element access

```
template<class T>
typename list<T>::reference front()
{  return *begin(); }

template<class T>
typename list<T>::const_reference front() const
{  return *begin(); }

template<class T>
typename list<T>::reference back()
{  return *--end(); }

template<class T>
typename list<T>::const_reference back() const
{  return *--end(); }
```

- Example (Cont'd)

  Iterators

```
template<class T>
typename list<T>::iterator list<T>::begin()
{  return head->succ; }

template<class T>
typename list<T>::const_iterator
list<T>::begin() const { return head->succ; }

template<class T>            // C++11
typename list<T>::const_iterator
list<T>::cbegin() const { return head->succ; }

template<class T>
typename list<T>::iterator list<T>::end()
{  return head; }

template<class T>
typename list<T>::const_iterator
list<T>::end() const { return head; }

template<class T>            // C++11
typename list<T>::const_iterator
list<T>::cend() const { return head; }
```

1 All of them return by value without **&** ref-qualifier or **const** qualifier on the return type. This allows the returned object to be modified and is useful. For example,

```
*--a.end()          // access the last element
*++a.begin()        // access the 2nd element
```

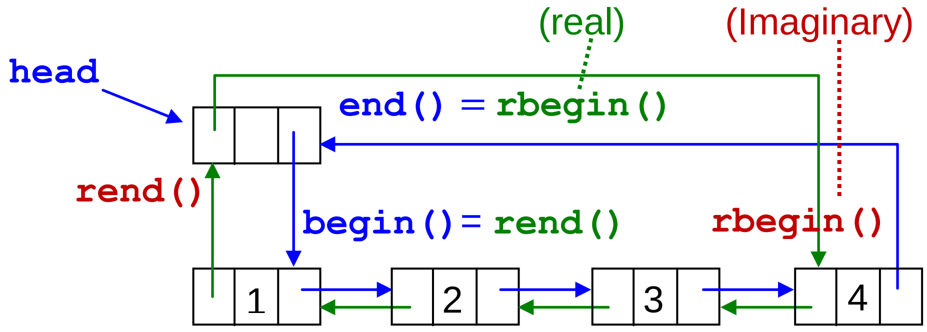
2 C++11 introduces **cbegin()** and **cend()** to facilitate the invocation of the new-style **insert** and **erase**. Without them, we have to write

```
// iterator → const_iterator
insert(begin(),d);
// list<T>* → const list<T>*
insert((const list<T>*)this->begin(),d);
```

STL <itearor>: Part B

Reverse iterators

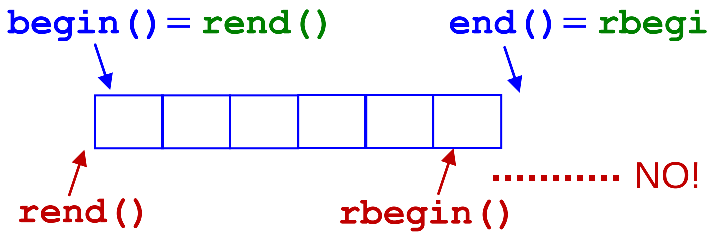
- Only for bidirectional and random-access iterators
  Iterate in the opposite direction



- Example

```
list<int> a{1,2,3,4};
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 real **`rbegin()`** and **`rend()`** are 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.



Observe that derefenencing a reverse iterator is slower than derefenencing an iterator.

- How to obtain a reverse iterator class?

```
template<class T>
class list {
public:
   class iterator;
   class const_iterator;
   typedef std::reverse_iterator<iterator>
                                            reverse_iterator;
   typedef std::reverse_iterator<const_iterator>
                                      reverse_const_iterator;
};
```

- **reverse_iterator** is an iterator adaptor that iterates in the opposite direction to the underlying iterator.

  Here is an abridged version of **reverse_iterator**

```
template<typename Iterator>
class reverse_iterator {
public:
// itearor traits
   typedef typename iterator_traits<Iterator>::reference
          reference;
// members
   reverse_iterator(Iterator it) : current(it) {}
   reverse_iterator& operator++()
   {
      --current; return *this;
   }
   reference operator*() const
   {
      Iterator tmp=current; return *--tmp;
   }
// other members omitted
private:
   Iterator current;
};
```

Move iterator

- **`move_iterator`** is an iterator adaptor with the same behavior as the underlying iterator except that it turns the value of **`operator*()`** from lvalue to rvalue reference.

- Here is an abridged version of **`move_iterator`**

```
template<typename Iterator>
class move_iterator {
public:
// itearor traits
   typedef typename iterator_traits<Iterator>::value_type
           value_type;
   typedef value_type&& reference;
// members
   move_iterator(Iterator it) : current(it) {}
   reference operator*() const     // ≠ Iterator
   {
      return std::move(*current);
   }
   move_iterator& operator++()     // = Iterator
   {
      ++current; return *this;
   }
// other members omitted
private:
   Iterator current;
};
```

- A non-member help function

```
move_iterator<Iterator>
make_move_iterator(const Iterator& i) {
   return move_iterator<Iterator>(i);
};
```

- Example

```
list<string> xs{"Snoopy","Pluto","Garfield"};
list<string> ys(make_move_iterator(xs.begin()),
                make_move_iterator(xs.end()));
cout << xs.size() << ys.size();    // 33
```

cf.

```
list<string> xs{"Snoopy","Pluto","Garfield"};
list<string> ys(std::move(xs));
cout << xs.size() << ys.size();    // 03
```

Generic algorithm

- Example

```
template<class ForwardIterator,class T>
ForwardIterator
remove(ForwardIterator first,
       ForwardIterator last,const T& value);
```

This STL generic function eliminates all the elements referred to by iterator **`it`** in the range **`[first,last)`** for which the condition **`*it`** == **`value`** holds and returns an iterator pointing to the end of the resulting sequence.

For example, removing 2 from the 9-element sequence
3, 2, 4, 5, 2, 2, 6, 7, 8
results in the sequence that consists of the elements
3, 4, 5, 6, 7, 8

Unlike **`list<T>::remove()`**, this function doesn't actually erase any element from the original sequence. Physically, the resulting sequence still has 9 elements; but, logically, it has only 6 elements – the remaining 3 elements are meaningless.

logical end (returned iterator)

3, 4, 5, 6, 7, 8, $\underbrace{?, ?, ?}_{\text{undefined}}$ physical end

- Example (Cont'd)

One way to implement **`remove`** proceeds as follows:

3, 2, 4, 5, 2, 2, 6, 7, 8 $\Rightarrow$ 3, 4, 5, 6, 7, 8, ?, ?, ?

where each arrow indicates a move assignment.

```
template<class ForwardIterator,class T>
ForwardIterator
remove(ForwardIterator first,
       ForwardIterator last,const T& value)
{
   ForwardIterator logical_end=first;
   for (ForwardIterator it=first;it!=last;++it)
      if (*it!=value) {
         if (it!=logical_end)
            *logical_end=std::move(*it);
         ++logical_end;
      }
   return logical_end;
}
```

Q: Why doesn't it erase elements from the original sequence?
A: Some containers, e.g. static arrays, that support **`Forward Iterator`** don't support the erasion operation.

Q: Given

```
int a[9]={3,2,4,5,2,2,6,7,8};
list<int> b(a,a+9);
```

What is the difference between

```
b.remove(2);
b.push_back(9);
```

and

```
remove(b.begin(),b.end(),2);
b.push_back(9);
```

A: The former yields the sequence 3, 4, 5, 6, 7, 8, 9.
The latter yields the sequence 3, 4, 5, 6, 7, 8, ?, ?, ?, 9.
It would be better to make use of the unoccupied slots, say

```
*remove(b.begin(),b.end(),2)=9;
```

- Example

Function dispatch using iterator category at compile time

```
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first,InputIterator last)
{
   return _distance(first,last,
      typename
      iterator_traits<InputIterator>::iterator_category());
}

template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
_distance(InputIterator first,InputIterator last,
          input_iterator_tag)
{
   typename
   iterator_traits<InputIterator>::difference_type n=0;
   while (first!=last) { ++first; ++n; }
   return n;
}

template<class RandomAccessIterator>
typename
iterator_traits<RandomAccessIterator>::difference_type
_distance(RandomAccessIterator first,
          RandomAccessIterator last,
          random_access_iterator_tag)
{
   return last-first;
}
```

Observe that for **InputIterator**, the distance must be $\geq$ 0.
For **RandomAccessIterator**, the distance may be $<$ 0.