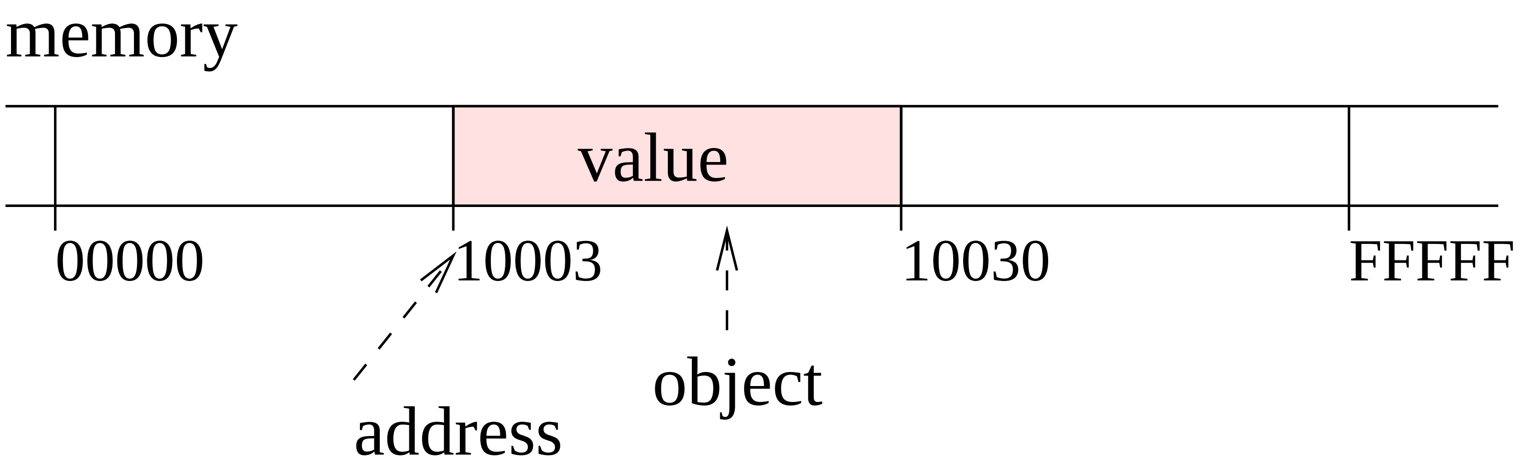
overview

1. C++ fundamentals; built-in types
2. Functions and **structured programming**
3. User-defined types: classes and **abstract data types**
4. Built-in type constructors: pointers and arrays
5. User-defined type constructors: templates
6. **Generic programming** using the STL
7. Subtypes, inheritance and **object-oriented programming**
8. Example application

objects, values, types



- An object has a **type** which implies a **size** (e.g. **int**)
- A type supports a set of **operations** (e.g. **+**) ⇒ ADT

variables, constants, expressions

Ways to **refer** to a value:

- A **variable** refers to an object (and its value).

$$\mathbf{x} \rightarrow address \rightarrow value$$

- A **constant** refers to a value: **`123`**
- An **expression** refers to a value: **`x + 123`**

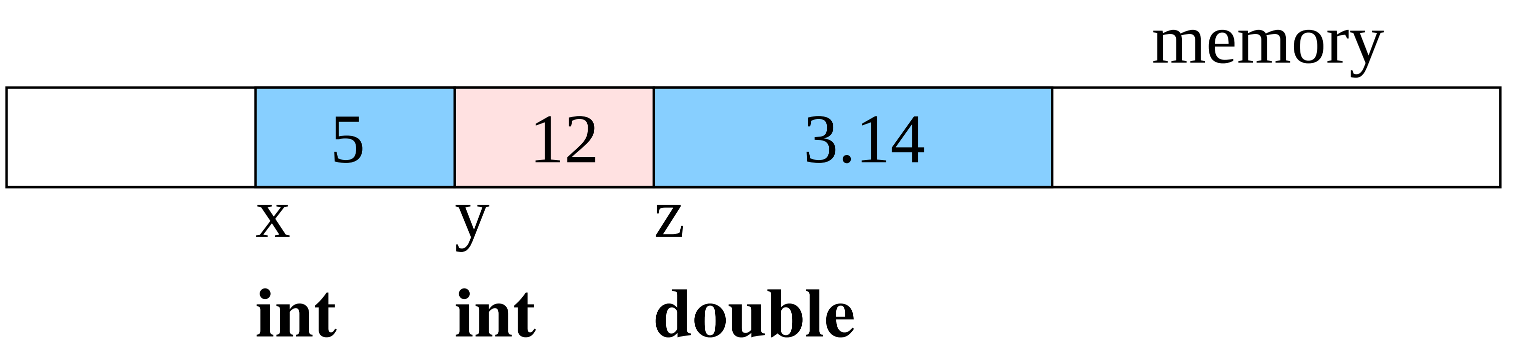
An value with an address is an **`lvalue`**, otherwise it is an **rvalue**.

`x`	lvalue
`123`	rvalue
`x+123`	rvalue

defining objects

```
NameOfType NameOfVariable(InitialValue);
```

int x(5);
int y(x+7); // *initial value is specified by expression*
double z(3.14);



Alternative syntax (not recommended, read only):

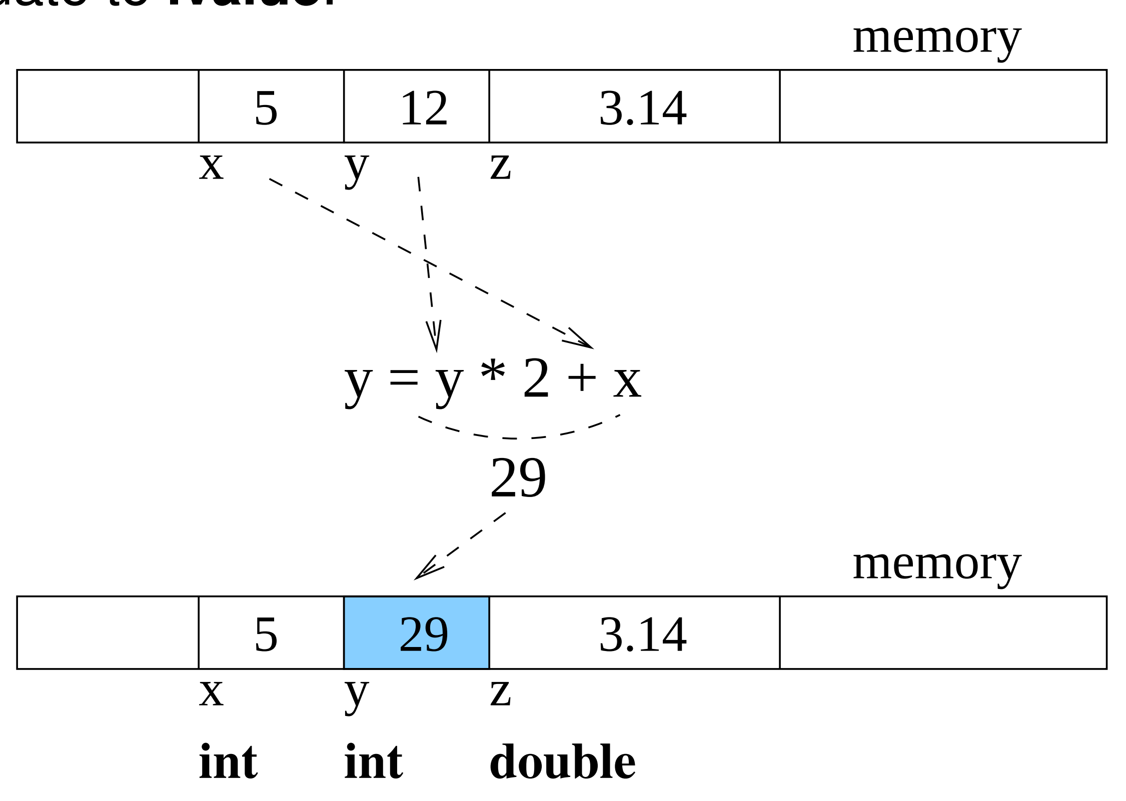
int x(5),y(12);
double z = 3.14;

manipulating objects: assignment

```
LeftExpression = RightExpression
```

LeftExpression must evaluate to **lvalue**.

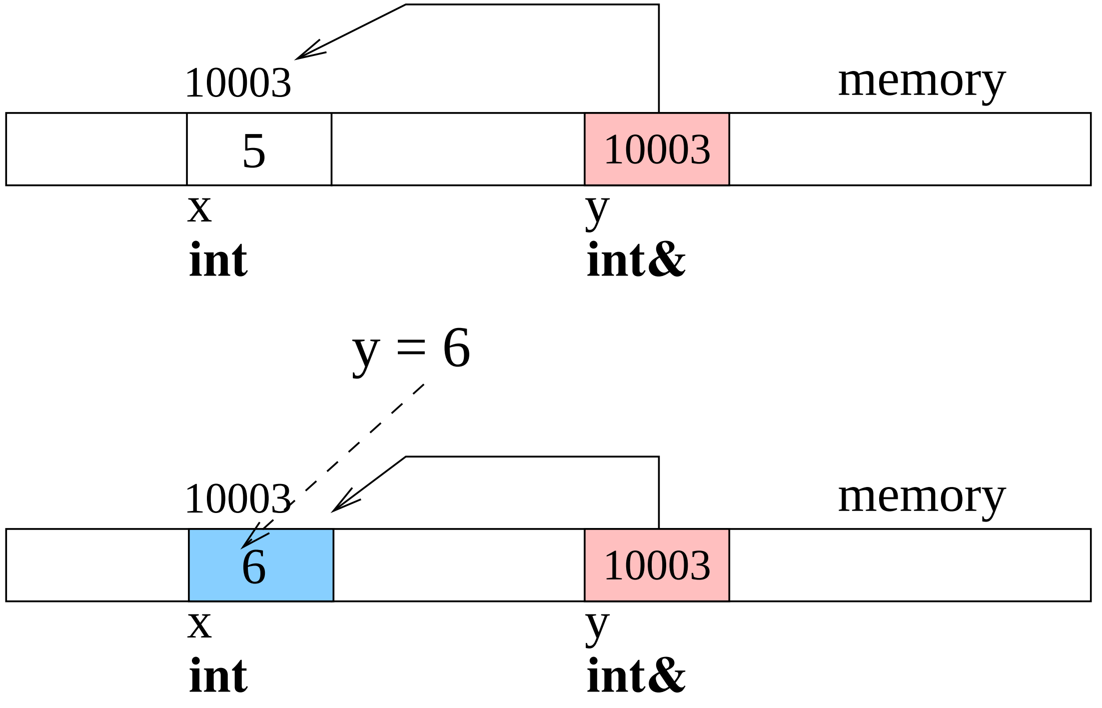
int x(5);
int y(x+7);
double z(3.14);
$y = y * 2 + x;$



reference variables

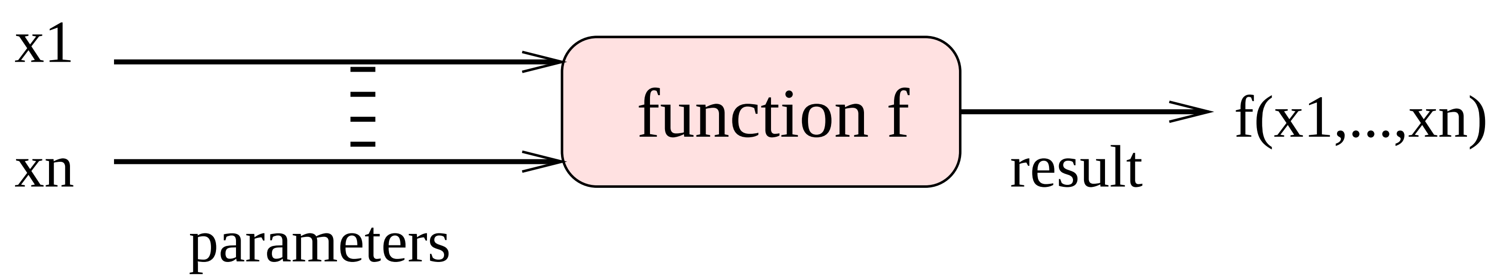
```
NameOfType& NameOfVariable(Expression);
```

Expression must yield **lvalue** of type **`NameOfType`**



int x(5);
int& y(x);
// type of y is int&
y = 6;

functions




```
int     x(3);
int     y(0);
int     z(5);

y = x * x;
y = y + z * z;
```

⇒

```
int  // return type
square(int u) { // u is formal parameter
return u*u; // return value
}

y = square(x) + square(z);
```

```
ReturnType
FunctionName(FormalParameterDeclarationList) {
StatementList
}
```

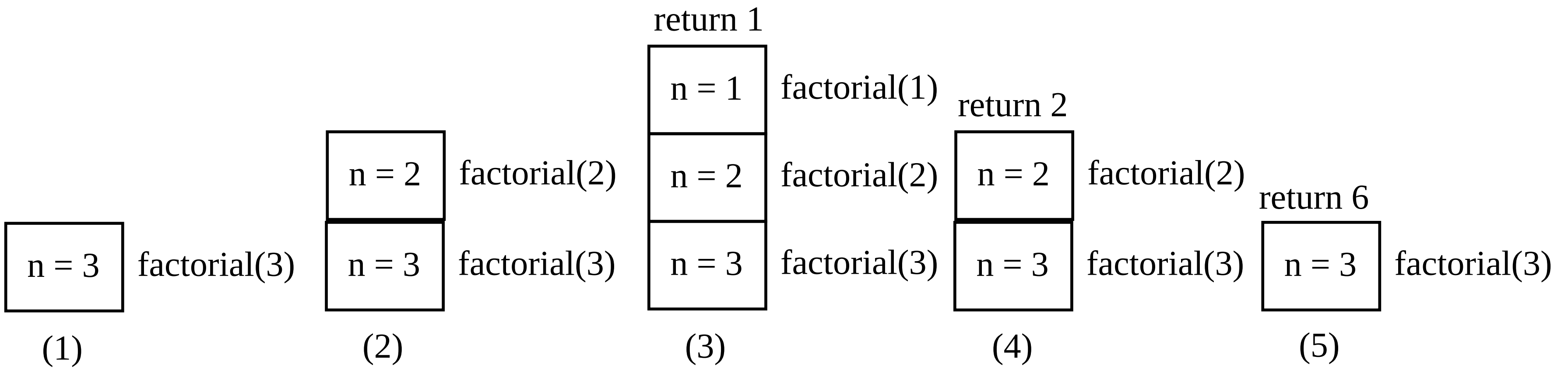
calling a function

$y\ =\ \mathrm{square}(x{+}1);$

1. Evaluate expression corresponding to **formal parameter** (**`u`**) $\Rightarrow$ 4
2. Initialize fresh **local parameter object** with this value $\Rightarrow$ $\textbf{int}\ u(4);$
3. Execute statements in the function **body**. This occurs in an **environment** containing local (**`u`**) and non-local names.
4. To evaluate **`return exp`**: create new object initialized with value of **`exp`** $\Rightarrow$ $\textbf{int}\ \mathrm{tmp}(u*u);$
5. Caller processes new object $\Rightarrow$ $y = \mathrm{tmp};$
6. Deallocate memory of function call **frame**.

the frame stack

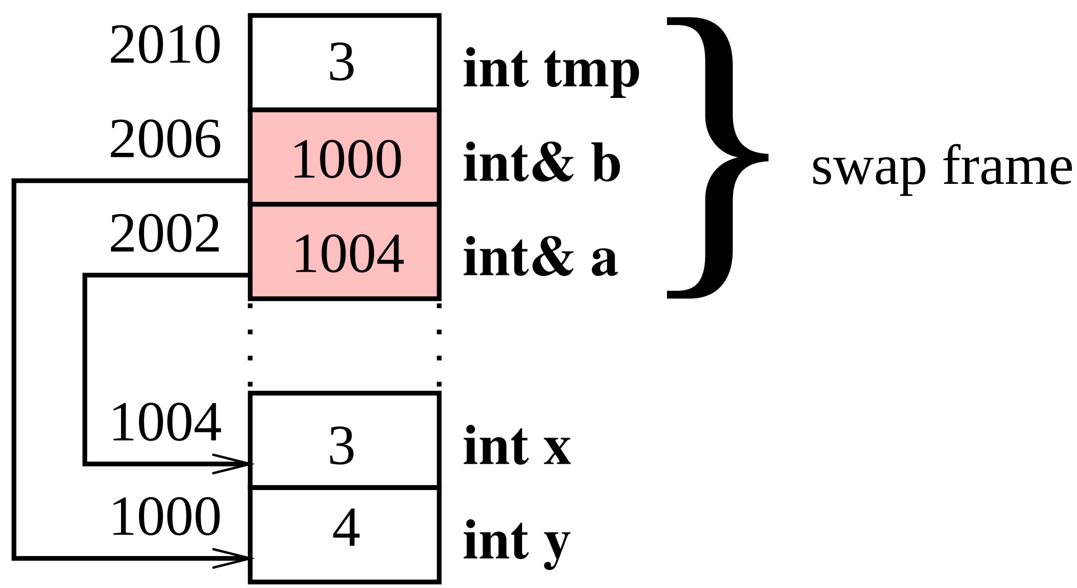
```
int
factorial(int n) {
if (n<2)
  return 1;
else
  return n * factorial(n-1);
}

factorial(3);
```



parameter passing

C++ only supports **call by value** *but* call by value on parameters of reference type is equivalent to **call by reference**:

```
int     x(3);
int     y(4);

void
swap(int& a,int& b) {
int     tmp(a);
a = b;
b = tmp;
}

swap(x,y);
```



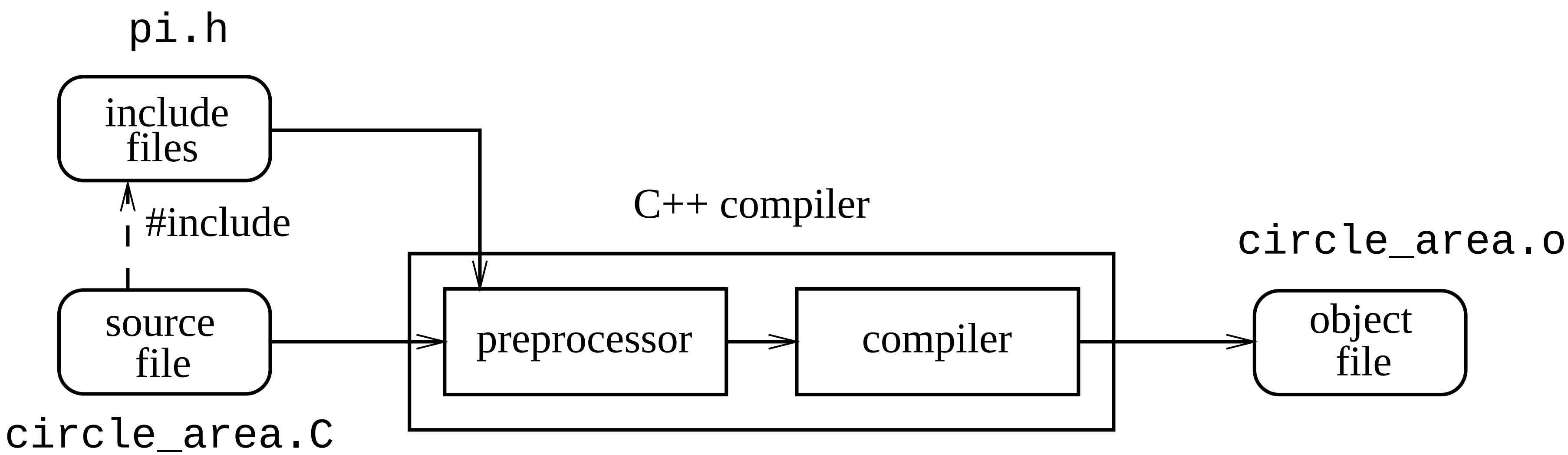
program structure

- A C++ program consists of **translation units**
- A translation unit consists of **definitions** and **declarations** of **types**, **variables** and **functions**.
- A **declaration** tells the compiler about the existence and name of something.

```
extern double pi; // declaration of a variable
double circle_area(double radius); // declaration of a function
```

- A **definition** causes the compiler to
  - generate code for a function definition
  - allocate memory for a variable definition
  - compute the size and other properties of a newly defined type

the compilation process



- An **include file** contains **declarations** of concepts defined in another translation unit.
- The **preprocessor** is a macro processor that interprets **directives** like **`#include`**, **`#define`**, **`#ifdef`**, **`#ifndef`**.

program organization example

- **source files** (translation units) contain definitions
  - `circle.C` contains **definition** of `circle_area` function
  - `pi.C` contains **definition** of `pi` variable
  - `prog1.C` contains **definition** of `main` function
- **include files** contain declarations
  - `circle.h` contains **declaration** of `circle_area` function
  - `pi.h` contains **declaration** of `pi` variable

Dependencies:

- `prog1.C` needs `circle.h`
- `circle.C` needs `pi.h` (and `circle.h`)
- `pi.C` (needs `pi.h`)

pi.h

```
#ifndef PI_H // read stuff until endif ONLY if PI_H is not defined
#define PI_H // assign 1 to the preprocessor variable PI_H
extern double pi; // declaration of variable pi
#endif // end of conditional inclusion
```

---

pi.C

```
#include "pi.h" // include declaration to ensure declaration
                // and definition are consistent
double pi(3.14); // definition of variable pi
```

---

circle.h

```
#ifndef CIRCLE_H // ensures that declaration is not read twice (see pi.h)
#define CIRCLE_H
double circle_circumference(double radius); // function declaration
double circle_area(double radius); // function declaration
#endif
```

circle.h

```
#ifndef CIRCLE_H // ensures that declaration is not read twice (see pi.h)
#define CIRCLE_H
double circle_circumference(double radius); // function declaration
double circle_area(double radius); // function declaration
#endif
```

---

circle.C

```
#include "pi.h" // needed for declaration of pi which is used here
#include "circle.h" // declaration of functions defined here;
                    // including them ensures the consistency
                    // of the declarations with the definitions
double // function definition
circle_circumference(double radius) {
return 2 * pi * radius;
}

double // function definition
circle_area(double radius) {
return pi * radius * radius;
}
```

prog1.C

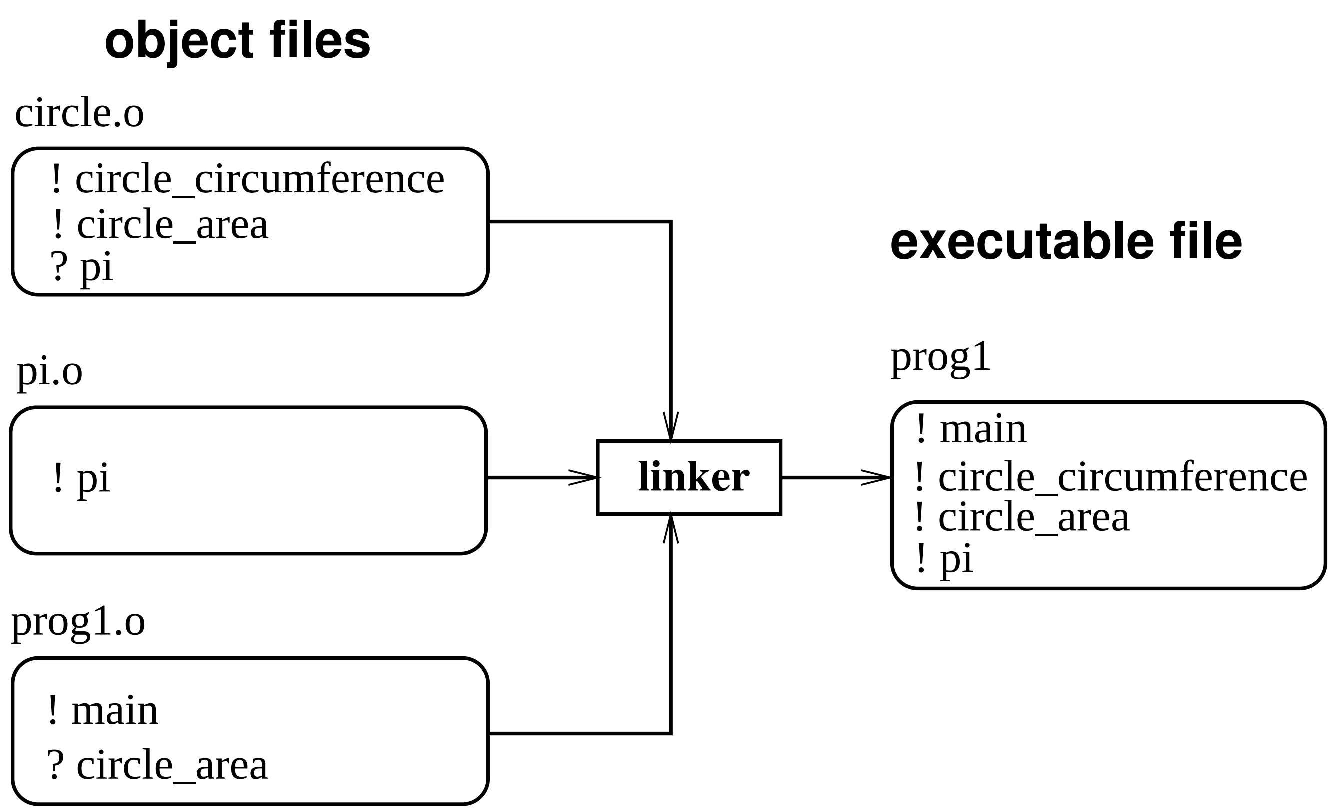
```
#include <iostream> // contains declaration for operator<<(ostream&,X)
#include "circle.h" // we use a function declared in circle.h

int // function definition
main() {
double r(35); // local variable definition, r will live in the call frame
cout << circle_area(r) << endl; // writes area of circle to standard output
return 0; // all is well: return 0
}
```

When a C++ program is executed, the **`main`** function is called. The return value is an **`int`**. By convention, **`0`** means that the execution went ok. Any other value indicates an error. The return value can be tested from outside, e.g. from the shell.

```
WEndy% prog1 || echo "error"
```

linking a program

- resolve symbols (e.g. **`circle_area`** in **`prog.o`**)
- relocate code



linking example

- **prog1.o**

```
00000000 <main>:
   0:   55                  pushl  %ebp
  ...
  1a:   dd 1c 24            fstpl  (%esp,1)
  1d:   e8 fc ff ff ff      call   1e <main+0x1e>
  ...
```

- **prog1**

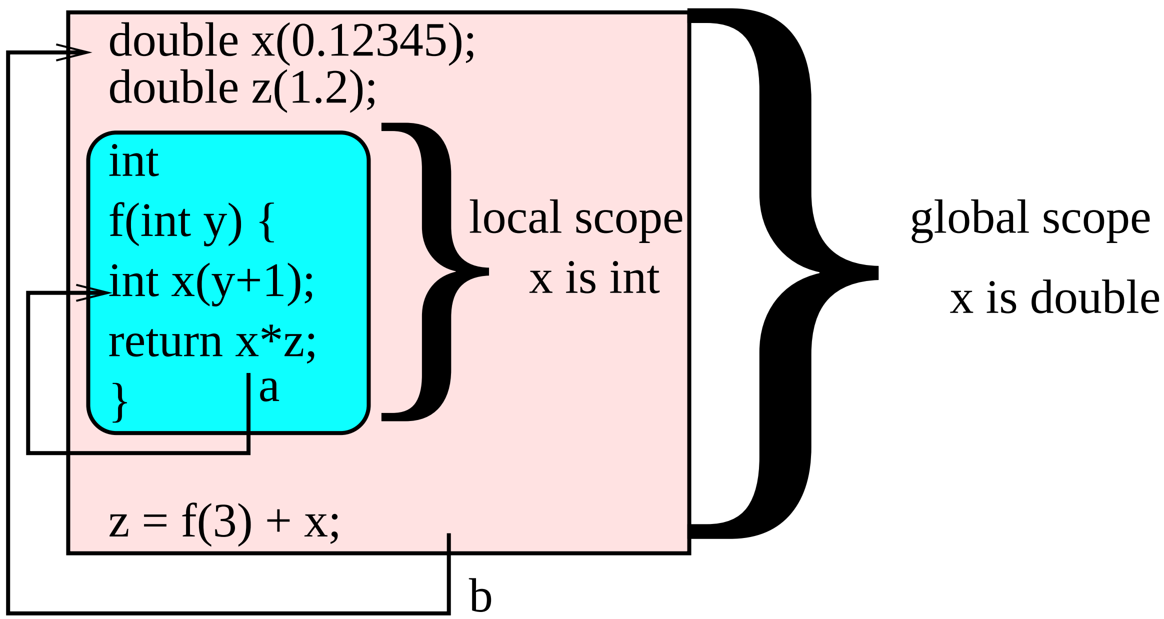
```
08048710 <main>:
 8048710:     55               pushl  %ebp
 ..
 804872a:     dd 1c 24         fstpl  (%esp,1)
 804872d:     e8 36 00 00 00   call   8048768 <circle_area(double)>
 ..
08048768 <circle_area(double)>:
 8048768:     55               pushl  %ebp
```

lexical considerations

- name of defined type/function/variable: **identifier**:
  - start with letter or _
  - contains letters, digits or _
  - convention: **`MyClass`**,**`a_simple_function`**, **`MAXINT`**
  - not a **keyword**
  - *choosing good names is crucial for understandable programs*
- Comments: **`//`** until end of line
  - *important documentation for code users and maintainers*
  - should be consistent with the code
  - should have added value

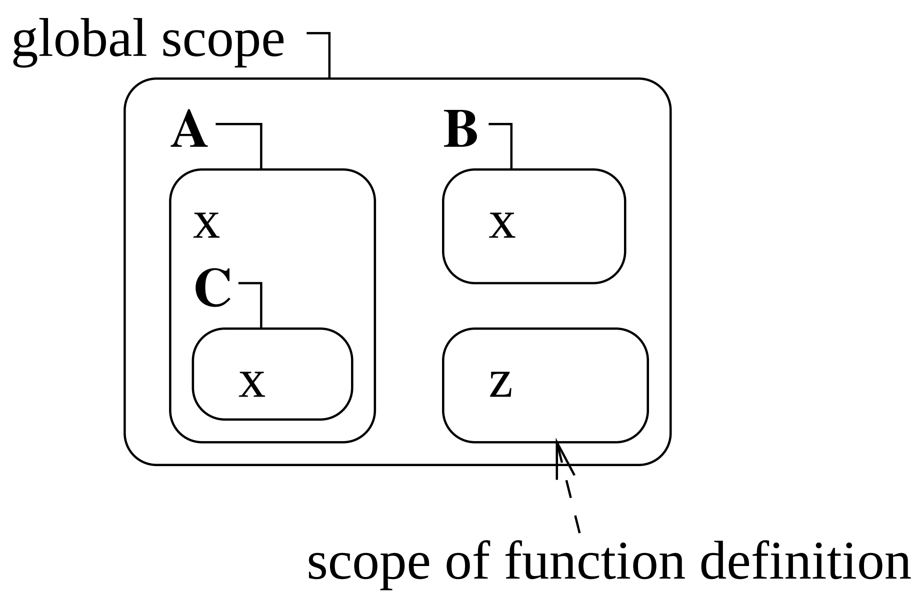
scopes

- C++ programs may consist of thousands of translation units: difficult to avoid name conflicts.
- Names are organized hierarchically in (nested) **scopes**
- Some constructs, e.g. a class type or a function body automatically define a new scope (nested in the encompassing scope).



namespaces: user-defined scopes

```
namespace A {
  int x;
  namespace C {
    int x;
    }
  }
namespace B {
  int x;
  }
int
main() {
int z(2);
// error: which x?
z = x;
}
```




```
// possible correction for z=x
z = A::C::x; // ok
// other possibility:
using A::C::x;
z = x;
```

using namespaces

```
namespace A {
  namespace C {
    extern int x; // only a declaration
    }
  }
namespace B {
  int x;
  }
int A::C::x; // definition of name "x" declared in ::A::C

namespace A { // continuation of namespace A
  int x;
  }
int
main() {
int z(2);
using namespace A::C;
z = x; // thus "x" refers to A::C::x
}
```

primitive types

arithmetic types

integral types

integer types

```
int
unsigned int
short
unsigned short
long
unsigned long
```

```
char
unsigned char
signed char
```

```
bool
```

floating point types

```
float
double
long double
```

```
wchar_t
```

char	c('\n'); // *newline*
int	i(0777); // *octal*
unsigned short	s(0xffff); //*hex*
long double	pi(4.3E12); // *4.3 ∗ 1000000000000*
bool	stupid(**true**);

conversions

```
long    l('a'); // char fits into l, no problem; l will be 97
int     i(3.14); // compiler will generate a warning; i will be 3
double  d(0.5);
float   f(0.6);
int     i(0);

i = d + f; // i will become 1
```

Compiler

- performs operation in "widest" type
- tries to do a reasonable conversion (warns if target too small) for assignment

operations on primitive types

- Operations are functions (can be redefined for user-defined types).

```
bool  operator&&(bool,bool);
T&    operator=(T& lvalue,T value);
```

- Assignment is expression and can be compounded.

```
x = y = z = 0; // x = ( y = ( z = 0 ));
x += 3; // x = x + 3;
```

- I/O

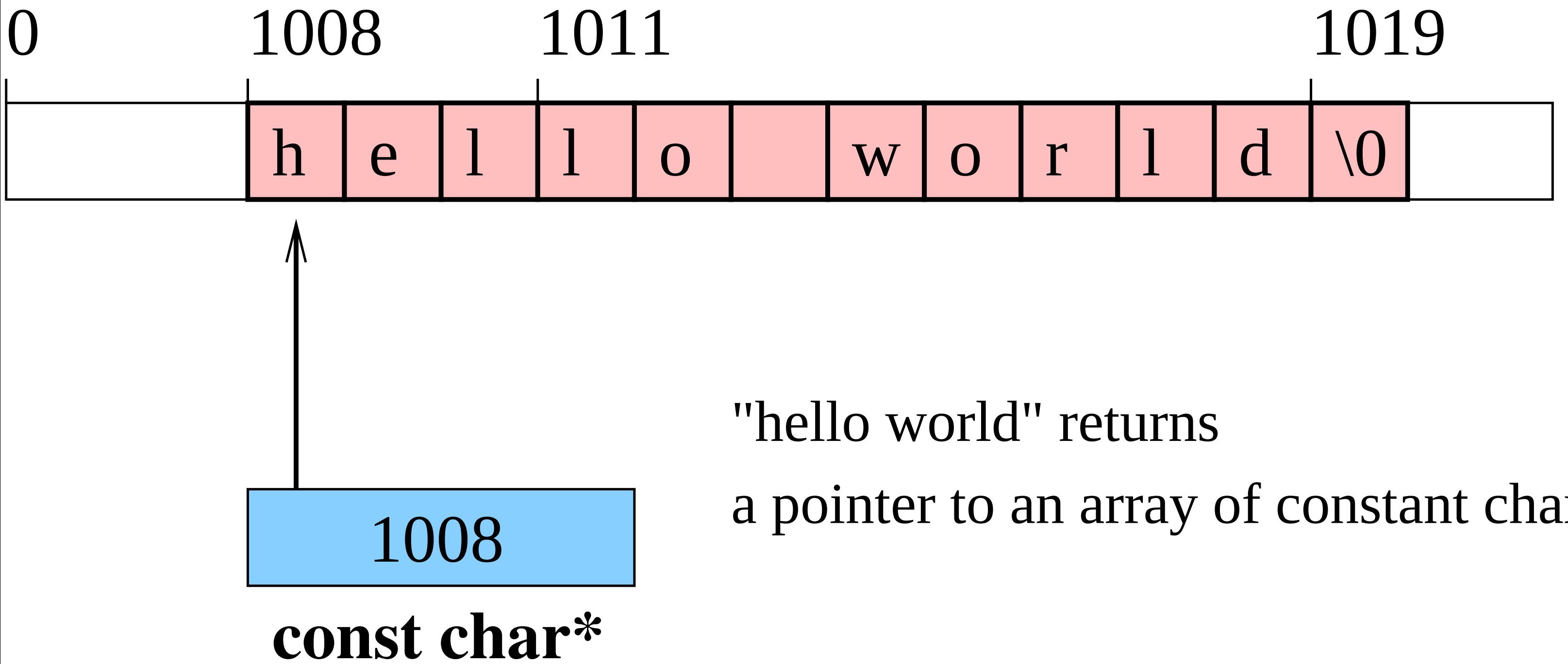
```
ostream& operator<<(ostream&,int);
istream& operator>>(istream&,int&);

cin >> x >> y;
cout << x << y;
```

operations on arithmetic types

operator	function	example	remark
`type()`	value construction	`double('c')`	
`++`	post increment	la++	lvalue only
`--`	post decrement	la--	lvalue only
`sizeof()`	size of object or type	`sizeof(a)`	or `sizeof(double)`
`++`	pre increment	++la	lvalue only
`--`	pre decrement	--la	type lvalue only
`~`	bitwise complement	~i	integral type only
`!`	logical not	!a	
`-`	unary minus	-a	
`+`	unary plus	+a	
`*`	multiplication	a * a	
`/`	division	a / a	
`%`	modulo (remainder)	i % i	integral types only
`+`	addition	a + a	
`-`	subtraction	a - a	
`<<`	shift left	i << j	integral types only
`<<`	output to stream	`cout<< a`	
`>>`	shift right	i >> j	integral types only
`>>`	input from stream	`cin >> la`	to lvalue only

<table>
<tr><td><</td><td>less than</td><td>a < a</td><td></td></tr>
<tr><td><=</td><td>less than or equal to</td><td>a <= a</td><td></td></tr>
<tr><td>></td><td>greater than</td><td>a > a</td><td></td></tr>
<tr><td>>=</td><td>greater than or equal to</td><td>a >= a</td><td></td></tr>
<tr><td>==</td><td>equal to</td><td>a == a</td><td></td></tr>
<tr><td>!=</td><td>not equal to</td><td>a != a</td><td></td></tr>
<tr><td>&</td><td>bitwise and</td><td>i & j</td><td>integral types only</td></tr>
<tr><td>^</td><td>bitwise xor</td><td>i ^ j</td><td>integral types only</td></tr>
<tr><td>|</td><td>bitwise or</td><td>i | j</td><td>integral types only</td></tr>
<tr><td>&&</td><td>logical and</td><td>i && j</td><td></td></tr>
<tr><td>||</td><td>logical or</td><td>i || j</td><td></td></tr>
<tr><td>=</td><td>assign</td><td>la = a</td><td>left operand must be lvalue</td></tr>
<tr><td>*=</td><td>multiply and assign</td><td>la *= a</td><td></td></tr>
<tr><td>/=</td><td>divide and assign</td><td>la /= a</td><td></td></tr>
<tr><td>%=</td><td>modulo and assign</td><td>la %= a</td><td></td></tr>
<tr><td>+=</td><td>add and assign</td><td>la += a</td><td></td></tr>
<tr><td>−=</td><td>subtract and assign</td><td>la −= a</td><td></td></tr>
<tr><td><<=</td><td>shift left and assign</td><td>li <<= i</td><td>integral types only</td></tr>
<tr><td>>>=</td><td>shift right and assign</td><td>li >>= i</td><td>integral types only</td></tr>
<tr><td>&=</td><td>bitwise “and” and assign</td><td>li &= i</td><td>integral types only</td></tr>
<tr><td>|=</td><td>bitwise “or” and assign</td><td>li |= i</td><td>integral types only</td></tr>
<tr><td>^=</td><td>bitwise “xor” and assign</td><td>li ^= i</td><td>integral types only</td></tr>
</table>

string literals



"hello world" returns

a pointer to an array of constant characters

```
ostream& operator<<(ostream&,const char*);
const char* hi("hello world");

cout << "hello world"; // hello world
cout << "hello" "world"; // helloworld
cout << "hello world\n";
```

function declarations

$$\boxed{\mathbf{T_0\ f(T_1,..,T_n)}}$$

type of function	$T_f = [T_1 \times \ldots \times T_n \rightarrow T_0]$
signature of function	$T_1 \times \ldots \times T_n$
return type of function	T_0

side effects

```
int    x(0); // global variable

int f() { return x++: }

int
main() {
f(); // will return 0
f(); // will return 1
}
```

default parameters

A formal parameter may be given a **default** value.

- Only the last parameters
- If one parameter is omitted, all the following should be also (why?).

```
void printline(int i1,int i2,char separator = '\t', int base = 10);

printline(10,12,',',8); // 12,14
printline(10,12);       // 10    12
printline(10,12,'|');   // 10|12
printline(10,12,8);     // interpreted as printline(10,12,'\b',10);

print(const Student&,ostream& os = cout);
print(fred); // print(fred,cout);
print(lisa,cerr);
```

unspecified number of parameters

Function can find out the number of actual parameters at run time.

```
void err_exit(int status,char* format,...);

err_exit(12,"cannot open file %s: error #%d\n",filename,errorcode);
// analyzing format tells function that there are 2 further parameters
// of type const char* (%s) and int (%d), respectively
```

More info (and how to write such functions): book, man page for *cstdarg* macros.

inline functions

```
inline int
maximum(int i1,int i2) { // return the largest of two integers
if (i1>i2)
   return i1;
else
   return i2;
}
```

Compiler will replace calls “in line”:

```
i = maximum(12,x+3)    ⇒    int tmp(x+3);
                             if (12>tmp)
                               i = 12;
                             else
                               i = tmp;
```

Where to put definitions of inline functions? why?

overloading function definitions

Functions with the same name but different signatures are different.

```
#include <math.h> // contains declaration for rint()

int sum(int i1,int i2,int i3) { return i1+i2+i3; }
int sum(int i1,int i2,int i3,int i4) { return i1+i2+i3+i4; }

int round(double a) { return int(rint(a)); } // rint will round a
int round(int i) { return i; }

int
main() {
sum(1,2,3,4); // calls sum(int,int,int,int); result will be 10
round(1.1); // will call round(double)
round('a'); // error?
}
```

determining which function is called

$$f(e_1, \ldots, e_n)$$

1. Determine the set F of **candidate functions** that could apply by finding the closest encompassing scope S containing one or more function declarations for f.
2. Find the **best match** in F for the call, i.e. the declaration f' whose *signature* best matches the call's signature $(T_{e_1}, \ldots, T_{e_n})$.

Note that:

- Definitions in a closer scope hide definitions in a wider scope.
- Default and unspecified arguments are taken into account when determining F.

overloaded example

```
ostream& operator<<(ostream&,char c);
ostream& operator<<(ostream&,unsigned char c);
ostream& operator<<(ostream&,signed char c);
ostream& operator<<(ostream&,const char *s);
ostream& operator<<(ostream&,const unsigned char *s);
ostream& operator<<(ostream&,const signed char *s);
int f(int i,int j=0) { return 1; }
int f(int k) { return 2; }

int
main() {
f(3); // which f?
cout << "hello world"; // operator<<(ostream&,const char *s);
}
```

function definition

```
ReturnType
FunctionName(FormalParameterDeclarationList) {
StatementList
}
```

Kinds of statements:

expression statement	**`OptionalExpression ;`**	incl. function calls, assignment
declaration statement	**`ObjectDefinition ;`**	e.g. local variables
control flow statement	...	

Example:

x = f(y); // *expression statement*
double d(3.14); // *declaration statement*

the compound statement and the sequence operator

A compound statement defines a new scope.

```
{
OptionalStatementList
}
```

Example:

```
{
int tmp(x); x = y; y = tmp;
}
```

```
Expression1 , Expression2 , ...  , ExpressionN
```

Example:

$x = (cin >> y,\ 2 * y);\ \approx$

```
cin >> y;
x = 2 * y;
```

the if statement

`if (Expression)`
`  StatementIfTrue`
	`if (Expression)`
	`  StatementIfTrue`
	`else`
	`  StatementIfFalse`

Example:

```
if (a>b) // StatementIfTrue is if statement
  if (c>d) // a>b && c>d; StatementIfTrue is expression statement
    x = 1;
  else
    x = 2; // a>b && c<=d

if (x>10) { // StatementIfTrue is compound statement
  int tmp(y);
  y = x;
  x = tmp;
  }
```

the ? operator

```
Condition ?  ExpressionIf :  ExpressionElse
```

m = (x>=y ? x : y) + 1; $\approx$

```
if (x>=y)
  m = x+1;
else
  m = y+1;
```

What about

```
int x(3);
int y(4);
m =  x>=y ? x : y + 1;
```

the while statement

```
while (Expression)
  Statement
```

≡

```
if (Expression) {
   Statement;
   while (Expression)
      Statement;
   }
```

```
int
factorial(int n) {
int result(1);
while (n>1)   {
  result *= n; // can be done in 1 line
  --n;
  }
return result;
}
```

the do statement

`do`
`  Statement`
`while (Expression)`
	$\equiv$	`Statement`
		`while (Expression)`
		`  Statement`

```
int sum(0);
do {
  int i(0);
  cin >> i;
  sum += i;
  }
while (cin); // while state of input stream is ok
```

the for statement

```
for (ForInitialization; (ForCondition); ForStep)
    Statement
```

is equivalent to

```
ForInitialization;
while (ForCondition) {
    Statement
    ForStep;
    }
```

```
int
factorial(int n) {
int     result(n?n:1); // make a copy of the input, but 0 becomes 1
if (result>2) // if n <=2, the result is n and we don't have to do anything
  for (int j=2;(j<n);++j)  // multiply with every number < n, except 1
    result *= j; // use compound assignment for multiplication
return result;
}
```

the switch statement

```
switch (Expression) {
    case Constant1:  Statement1 break;
    ...
    case ConstantN: StatementN break;
    default:  Statement
    }
```

```
int tmp(Expression);
if (tmp==Constant1)
   Statement1
else if (tmp==Constant2)
   Statement2
else if ...
else Statement // default
```

Expression must be integral (why?).

switch example 1

```
#include <stdlib.h> // for the random() functions
#include <time.h> // for the time() function
#include "err.h" // for the err_exit() function
int
throwdice(int score) { // throw dice and return updated score
srandom(time(0)); // use current time() value as seed for the random number generator
int result(random() % 6 + 1);
switch (result) { // update score, depending on the result of throwing the dice
  case 1: score += 1; break;
  case 2: score += 3; break;
  case 3: score += 4; break;
  case 4: score += 6; break;
  case 5: score += 6; break;
  case 6: score += 8; break;
  default:
    err_exit(1,"error throwing dice: illegal result %d", result);
  }
return score;
}
```

switch example 2

```
bool
isvowel(char c) { // return true iff c is a vowel letter
switch(c) {
  case 'a': case 'e': case 'i':
  case 'o': case 'u':
  case 'A': case 'E': case 'I':
  case 'O': case 'U':
        return true; // no need for break; we exit immediately from the function
  default:
        return false;
  }
abort(); // should never get here
}
```

the return and break statements

```
return Expression;
break;
```

The **break** statement breaks out of the enclosing loop or switch.

```
#include <iostream>
void
break_demo() { // output?
for (int j=0;(j<10);++j) {
  for (int i=0;(i<5);++i) {
    if (i>3)
      break;
    cout << i << " ";
    }
  cout << "\n";
  }
}
```

the continue statement

interrupts current iteration and immediately starts next iteration of enclosing loop

```
#include <iostream>
void process(int);

int
main() { // demo of usage of continue statement
int i(0);
while (cin>>i) { // an input stream can be converted to a bool; the result is true iff input ok
  if (i<=0) // i not positive, throw away
    continue;
  if (i%2) // i odd, throw away
    continue;
  process(i); // i ok, process it
  }
}
```

automatic local objects

objects for local variables that are defined in a function body of compound statement scope are destroyed when control leaves the scope (pop frame stack)

```
void
f() {
  { // start compound statement that has its own nested scope
  int x(3); // local object definition
  x *= 2;
  } // end compound statement and scope
cout << x; // ???
}
```

static local objects

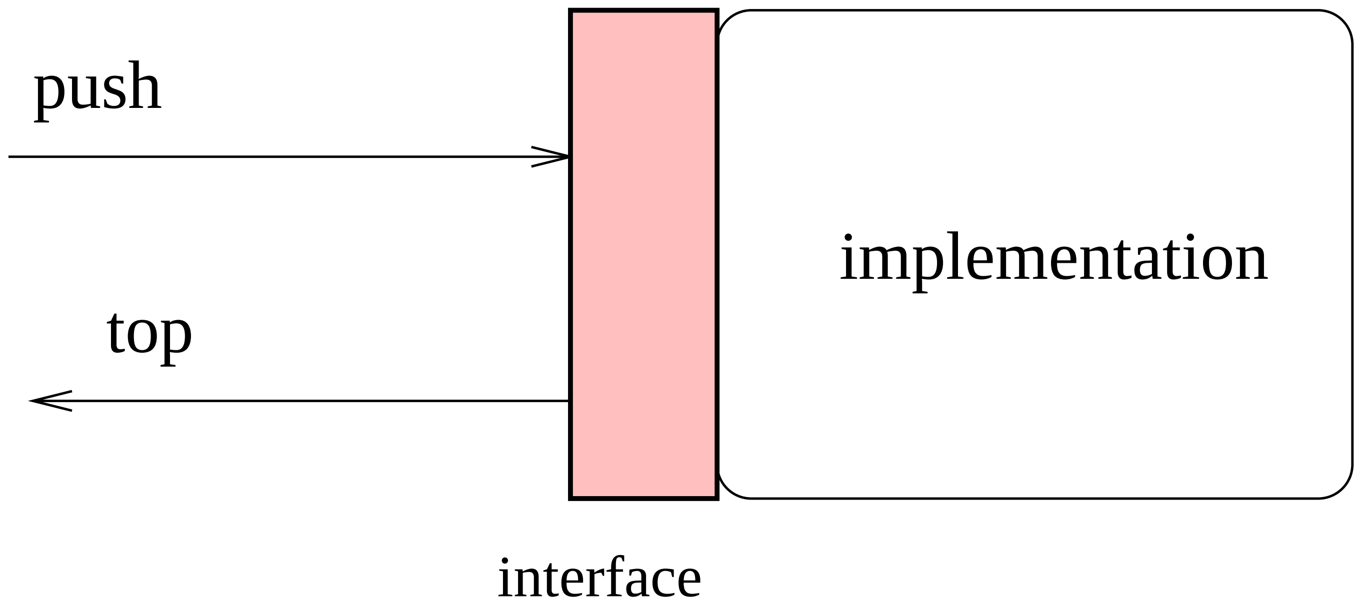
are persistent across function calls or scope entries (but same accessability rules as other local variables)

```
int
counter() {
static int n(0); // created when function is first called
return ++n;
}
```

it is safe to return a reference to a static local object:

```
int&
silly() {
static int n(0);
return n; // ok, n is static
}
silly() += 3; cout << silly(); // output?
}
```

abstract data types (adt's)

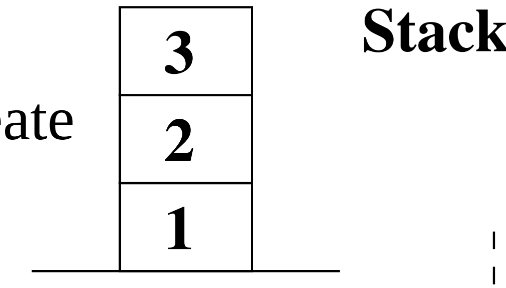


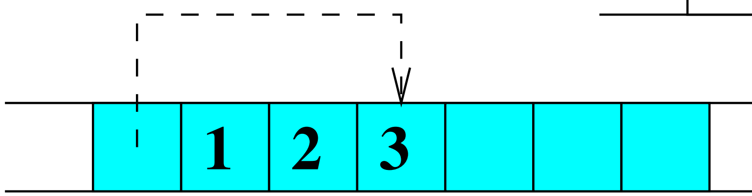
An ADT has

- a public **interface** specifying the available operations on the type
- a private **implementation** that describes
  - how information for an object of the ADT is represented
  - how operations are implemented

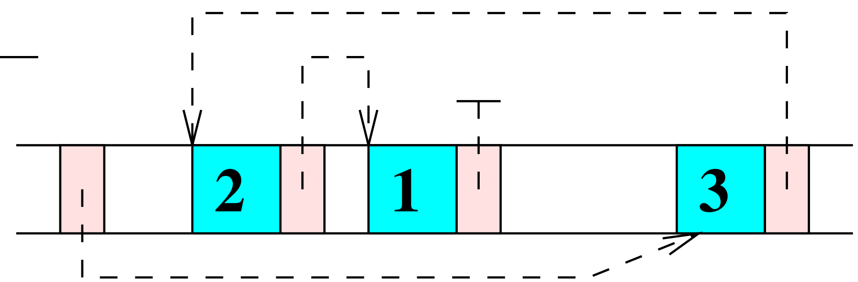
adt example: stack

operations:
top, push, pop, create





implementation using array



implementation using linked list

advantages of ADT:

- **abstraction**: use an ADT like a built-in type
- **encapsulation**: users are shielded from changes in the implementation

classes

C++ ADTs:

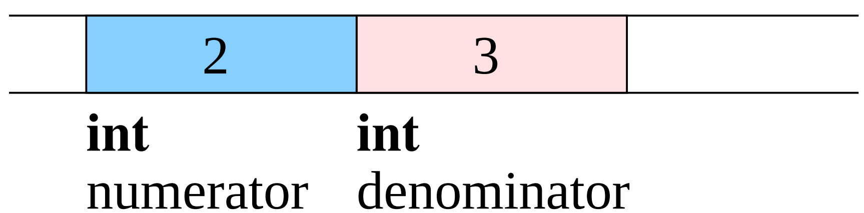
```
class NameOfClass {
public:
    MemberDeclarations
private:
    MemberDeclarations
}
```

- interface is provided by public **function member** declarations
- implementation is provided by
  - private **data member** declarations
  - function member definitions

class objects and data members

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational { // defines a scope called Rational
public:
  // public interface to be supplied
private:
  // implementation part
  int     numerator_;
  int     denominator_; // must not be 0!
};
#endif
```

Rational `(2/3)`



Rational r; // *just like another object definition?*

member function declarations

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
public:
  Rational     multiply(Rational r); // public member function declaration
  Rational     add(Rational r); // public member function declaration
private:
  // implementation part
  int    numerator_;
  int    denominator_; // must not be 0!
};
#endif
```

Calling a member function: specify **`target class object`**

```
Rational r,r1,r2;
r = r1.multiply(r2);
```

constructors: initializing a class object

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
public:
                // constructor member function declaration
                Rational(int num,int denom);
  Rational      multiply(Rational r);
  Rational      add(Rational r);
private:
  int     numerator_;
  int     denominator_; // must not be 0!
};
#endif
```

Constructors are more flexible than just giving an initial value.

```
Rational        r(2,3); // will initialize r using Rational::Rational(2,3)
```

overloading constructors

```
class Rational {
public: // overloaded constructors
    Rational(int num,int denom); // initializes to num/denom
    Rational(int num); // initializes to num/1
    Rational(); // default constructor; initializes to 0/1
    Rational(const Rational& r); // copy constructor; initializes to copy of r
  Rational        multiply(Rational r);
  Rational        add(Rational r);
private:
  int     numerator_;
  int     denominator_; // must not be 0!
};
```

(Also other member functions may be overloaded).

```
Rational        r1; // calls Rational::Rational(); why not Rational r1();?
Rational        r2(r1); // calls Rational::Rational(r2)
Rational        r3(5); // calls Rational::Rational(5)
```

the default constructor

```
class Rational {
public:
  Rational(int num,int denom);
private:
  int     numerator_;
  int     denominator_;
};
Rational          r;
// compile error; why?
```

```
class Rational {
private:
  int     numerator_;
  int     denominator_;
};
Rational          r;
// what exactly happens?
```

the copy constructor (cctor)

```
class Rational {
public:
  Rational(int num,int denom);
private:
  int     numerator_;
  int     denominator_;
};

int f(Rational r) {
...
}
```

Cctor is used for passing class objects by value

```
Rational x;
f(x); // compiler will copy x using Rational::Rational(x);
```

The compiler will provide a default cctor if it is not explicitly defined.

member function definition

```
#include "rational.h"

Rational // note scope operator: multiply in scope Rational
Rational::multiply(Rational r) { // a/b * c/d = (a*c)/(c*d)
int num_result(numerator_ * r.numerator_);
int den_result(denominator_ * r.denominator_);
return Rational(num_result,den_result); // init. return value using ctor
}

Rational
Rational::add(Rational r) { // a/b + c/d = (a*d + c*b)/(b*d)
int num_result = numerator_ * r.denominator_ + r.numerator_ * denominator_;
int den_result = denominator_ * r.denominator_;
return Rational(num_result,den_result);
}
```

ctor definition

Use **member initialization list** to initialize data members.

```
Rational::Rational(int num,int denom): numerator_(num), denominator_(denom) {
// check that denom is not 0, if it is we simply abort the program
if (denom==0)
  abort();
}
```

alternative (inferior w.r.t efficiency; why?):

```
Rational::Rational(int num,int denom) {
numerator_ = num;
denominator_ = denom;
if (denom==0)
  abort();
}
```

inline member function definition

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
public:  // interface
  Rational(int num,int denom): numerator_(num), denominator_(denom) {
    if (denom==0)
      abort();
    }
  Rational multiply(Rational r) {
    return Rational(numerator_ * r.numerator_, denominator_ * r.denominator_);
    }
  Rational add(Rational r) {
    return Rational(numerator_ * r.denominator_ + r.numerator_ * denominator_,
                    denominator_ * r.denominator_);
    }
private: // implementation part
  int     numerator_;
  int     denominator_; // must not be 0!
};
#endif
```

member functions with default parameters

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
public:
    Rational(int num=0,int denom=1); // saves 2 overloaded ctor functions
  Rational       multiply(Rational r);
  Rational       add(Rational r);
private:
  int     numerator_;
  int     denominator_; // must not be 0!
};
#endif
```

user-defined conversions

Ctors are conversion functions (unless forbidden by **explicit** prefix).

```
Rational r;
r.multiply(2); // Rational tmp(2); r.multiply(tmp);
```

You can define explicit conversion member functions (why is this needed?).

```
class Rational {
public:
  ... // explain definition below
  operator double() { return double(numerator_)/denominator_; }
..
};

Rational r(1,3);
// the following prints 0.333
cout << r; // double tmp(r.operator double()); operator<<(cout,tmp);
```

operator overloading

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
public:
                Rational(int num=0,int denom=1);
  Rational      operator+(Rational r) { return add(r); }
  Rational      operator*(Rational r) { return multiply(r); }
  Rational      multiply(Rational r);
  Rational      add(Rational r);
private:
  int     numerator_;
  int     denominator_; // must not be 0!
};
#endif
```

Example:

```
r1+r2*r3; // r1.add(r2.multiply(r3));
```

operator overloading by non-member functions

problem:

```
Rational r;
r+2; // Rational tmp(2); r.operator+(tmp);
2+r; // compile error: no function operator+(int,Rational); defined
```

solution:

```
inline Rational operator+(Rational r1,Rational r2) { return r1.add(r2); }
```

now, normal conversion strategy will work:

```
2+r; // Rational tmp(2); operator+(tmp,r);
```

operators that can be overloaded

`[]`	`()`	`++`	`--`	`~`	`!`	`-`	`+`
`*`	`new`	`delete`	`delete []`	`new []`	`/`	`%`	`,`
`->*`	`<<`	`>>`	`<`	`<=`	`>`	`>=`	`==`
`!=`	`&`	`^`	`|`	`&&`	`||`	`=`	`*=`
`/=`	`%=`	`+=`	`-=`	`<<=`	`>>=`	`&=`	`|=`
`->`	`^=`						

- **`=`** (assignment), **`[]`** (subscript), **`()`** and **`->`** (member selection) must be defined as non-static member function (to ensure that first argument is lvalue).
- only for (at least one operand of) user-defined type

overloading the assignment operator

```
#include        <math.h> // for rint(double) which rounds a double to the nearest integer
class Rational {
public:         Rational(int num=0,int denom=1);
  ..
  Rational&     operator=(double d) {
                    int units(rint(d));
                    int hundreds(rint((d - units) * 100));
                    numerator_ = units * 100 + hundreds;
                    denominator_ = 100;
                    return *this;
                    }
private:
  ..
};
```

A default implementation of C& **operator**=(**const** C&) is available for each class C

```
Rational        r1(2,3);
r = r1 + 3; // r = 11/3
```

overloading increment, decrement operators

```
class Rational {
public:
 ..
  Rational        operator++() { // prefix version, e.g. ++r
                      Rational r(numerator_+denominator_,denominator_);
                      numerator_ += denominator_;
                      return r;
                      }
  Rational        operator++(int) { // postfix version, e.g. r++
                      Rational r(numerator_,denominator_);
                      numerator_ += denominator_;
                      return r;
                      }
private:
  ..
};
Rational r(1,2);
r1 = ++r;
r2 = r++;
```

forbidding operators

```
class Server {
public:
  Server(ostream& log,int port);
  ~Server();
  // lots of stuff omitted
private:
  // we forbid making copies of a Server object by declaring the
  // copy constructor and assignment operators to be private
  // (no definition is needed, by the way).
  Server(const Server&);
  Server& operator=(const Server&);
  ostream&        logfile;
  // stuff omitted
};

void start_protocol_bad(Server s); // error: call needs copy constructor
void start_protocol_ok(Server& s); // ok: call by reference
```

finalizing objects using destructors

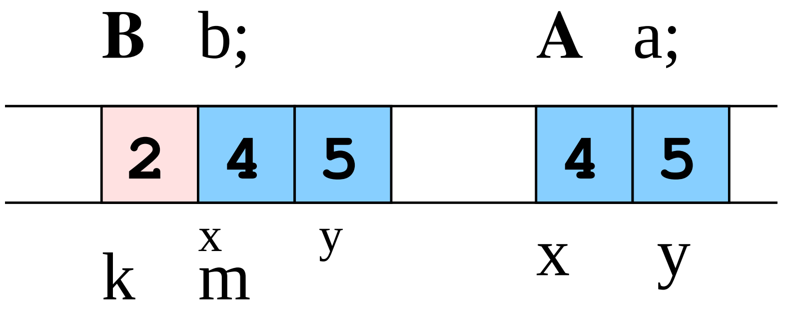
```
class ClassName {
    ~ClassName();
    ...
}
```

Called by system before the object is destroyed.

```
#include <unistd.h> // for close(int)
class File { // this is just an example, not a realistic design
public:
  File(const char* filename); // e.g. File f("book.tex");
  ~File() {   // destructor; why no parameters?
        close(fd_); } // close file descriptor corresponding to open file
  File& operator<<(const char*); // write to the file
  // lots of stuff omitted
private:
  int fd_; // file descriptor corresponding to open file
  // stuff omitted
};
```

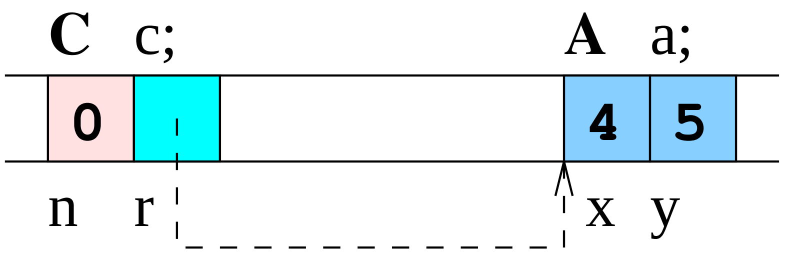
member objects

```
class A { // ...
public:
  A(int i,int j): x(i), y(j) {}
private:
  int    x;
  int    y;
};
class B { // ...
public:
  B(int i,A& a): k(i), m(a) {}
private:
  int   k;
  A      m; // a member object
};
A         a(4,5);
B         b(2,a); // what's happening?
```



member references

```
class A {
public:
   A(int i,int j): x(i), y(j) {}
private:
   int     x;
   int     y;
};
class C {
public:
   C(A& a,int i): r(a), n(i) {}
   // ...
private:
   int     n;
   A&      r; // not a member object, *must* be initialized
};
A          a(4,5);
C          c(a,0); // what's happening?
```



the life of a server class object

```
#include "file.h"
class Server {
public:
  Server(string logfilename,int port): logfile_(logfilename), port_(port) {
    // set up server
    logfile << "server started\n"; // why does this work
    }
  ~Server() { // close down server
    logfile << "server quitting\n"; // why does this work?
    }
  void serve() {  // handle requests
    }
  // lots of stuff omitted
private:
  File logfile_;
  int port_;
};
```

the life of a class object

1. (allocate memory)
2. Construction using a (possibly default) constructor function:
   (a) Construct member objects in the order of the initialization list of the constructor.
   (b) Execute the body of the constructor.
3. Provide services via member function calls, or as parameter to ordinary functions.
4. Destruction:
   (a) Execute code of destructor body, if there is a destructor.
   (b) Destroy member objects.
5. (allocate memory)

friends

```
class Rational {
public:
                Rational(int num=0,int denom=1);
  Rational      multiply(Rational r);
  Rational      add(Rational r);
  friend ostream& operator<<(ostream&,Rational); // now this
  // non-member function has access to private members of Rational
private:
  int     numerator_;
  int     denominator_; // must not be 0!
};
inline Rational   operator+(Rational r1,Rational r2) { return r1.add(r2); }
inline Rational   operator*(Rational r1,Rational r2) { return r1.multiply(r2); }
inline ostream& operator<<(ostream& os,Rational r) {
        return os << r.numerator_ << "/" << r.denominator_;
        }

Rational r(2,3);
cout << r; // what happens?
```

more friends

```
class Node {
friend class IntStack; // everything is private but IntStack is a friend
private: // this is the default, so this line could be dropped
    Node(int,Node* next=0);
    ~Node();
    Node*        next() { return next_; }
    int     item;
    Node* next_;
};
class IntStack { // stack of integers
public: // all member functions can use Node's
  IntStack();
  ~IntStack();
  Stack& push(int);
  int      top();
  bool    empty();
private:
  Node* top_; // pointer to topmost node
};
```

nested classes

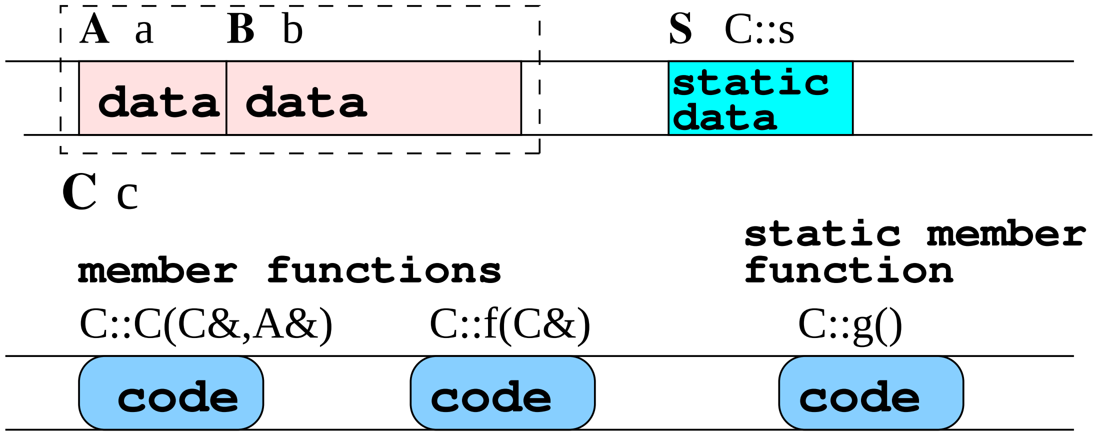
```
class IntStack { // stack of integers
public:
  IntStack();
  ~IntStack();
  Stack& push(int);
  // ..
private:
  class Node { // why is this solution better?
  public:
    Node(int,Node* next=0);
    ~Node();
    Node*        next();
  private:
    int    item;
    Node* next_;
  };
  Node* top_; // pointer to topmost node
};
inline  Node*
IntStack::Node::next() { return next_; }
```

static members

```
#ifndef POINT_H
#define POINT_H
class Point { // file point.h
public:
  Point(int X,int Y): x(X), y(Y) {
    ++count; }
  Point(const Point&p): x(p.x), y(p.y) {
     ++count; }
  ~Point() { --count; }
  static int get_count() {
    return count; }
  // ...
private:
  static int count; // declaration
  int x; // x-coordinate of point
  int y; // y-coordinate of point
};
#endif
```

```
// file point.C
#include "point.h"

// definition of static data member
int Point::count(0);

// usage:
Point p(1,2);
p.get_count();
// no target needed:
Point::get_count();
```

implementing classes




```
class C {
public:
  C(A& a);
  A f();
  static S g();
private:
  A a;
  B b;
  static S s;
};
```

- class objects:
  - have separate data area (**data members**)
  - share code (**function members**)
- **static data members** are shared and global
- **non-static member functions** have extra **target** object (lvalue) parameter.

enumeration types

finite integral types

```
class File {
public:
  enum Mode { READ, WRITE, APPEND}; // defines 4 names in scope File
  File(const char* filename, Mode mode=READ); // e.g. File f("book.tex");
  ~File();
  Mode mode() { return mode_; }
private:
  Mode  mode_;
  // ...
};
```

```
enum NameOfType { EnumeratorList };
```

```
File f("book.tex");
if (f.mode()==File::WRITE) {
  }
```

overloading operators on enumerated types

```
class Http {
public:
  enum Operation { GET, HEAD, PUT };
  enum Status { OK = 200, CREATED = 201, ACCEPTED = 202,
    PARTIAL = 203, MOVED = 301, FOUND = 302, METHOD = 303,
    NO_CHANGE = 304, BAD_REQUEST = 400, UNAUTHORIZED = 401,
    PAYMENT_REQUIRED = 402, FORBIDDEN = 403, NOT_FOUND = 404,
    INTERNAL_ERROR = 500, NOT_IMPLEMENTED = 501 };
..
};
ostream&
operator<<(ostream& os,Http::Status status) {
switch (status) {
  case OK: os << "OK"; break;
  case CREATED: os << "CREATED"; break;
  case ACCEPTED: os << "ACCEPTED"; break;
  ..
  }
return os;
}
```

typedef

```
typedef Declaration;
```

Defines short name for (complex) type expression.

```
typedef unsigned int uint;
uint    x; // equivalent with unsigned int x;

typedef Sql::Command::iterator    IT;

// also for function types:
typedef int UnaryFunction(int);
// UnaryFunction is type int -> int
UnaryFunction*   f(square); f(2); // pointer to function, see later
```

type constructors

A type constructor is a compile-time function $construct$ that, given a type t, returns another type $construct(t)$.
E.g. `&` is a type constructor

$$reference : T \rightarrow T\&$$

C++ supports

- built-in type constructors: `&` (references), `const` (constant), `*` (pointers), `[]` (arrays)
- user-defined type constructors: templates

constant objects

```
const NameOfType Variable(InitialValue);
```

Compiler will ensure that – after construction – the object referred to by **`Variable`** will not be changed.

```
const int x(4);
x = 5; // error

int    y(6);
const int& z(y);
z = 7; // error
y = 5; // ok, explain

const int u(7);
int& v(u); // what?
```

Note: construction (initialization) is not change!

constant reference parameters

- function promises not to modify the parameter; checked by compiler
- often (when?) more efficient that call-by-value

```
class Rational {
public:
  Rational(int num,int denom): numerator_(num), denominator_(denom) {
    if (denom==0)
      abort();
    }
  Rational multiply(const Rational& r) {
    return Rational(numerator_ * r.numerator_, denominator_ * r.denominator_);
    }
private:
  int     numerator_;
  int     denominator_; // must not be 0!
};
```

constant members: problem

```
class Rational {
  Rational(int num,int denom): numerator_(num), denominator_(denom) {
    if (denom==0)
      abort();
    }
  Rational multiply(const Rational& r) {
    return Rational(numerator_ * r.numerator_, denominator_ * r.denominator_);
    }
private:
  const int       numerator_;
  const int       denominator_; // must not be 0!
};

Rational          r1(1,2);
const Rational    r2(2,3);
r2.multiply(r1); // compile error
```

constant members: solution

A constant member function promises not to modify the target object.

```
class Rational {
  Rational(int num,int denom): numerator_(num), denominator_(denom) {
    if (denom==0)
      abort();
    }
  Rational multiply(const Rational& r) const {
    return Rational(numerator_ * r.numerator_, denominator_ * r.denominator_);
    }
private:
  const int       numerator_;
  const int       denominator_; // must not be 0!
};

Rational          r1(1,2);
const Rational    r2(2,3);
r2.multiply(r1); // ok
```

overloading and const

The usual matching rules apply (**`const T`** is a “normal” type).

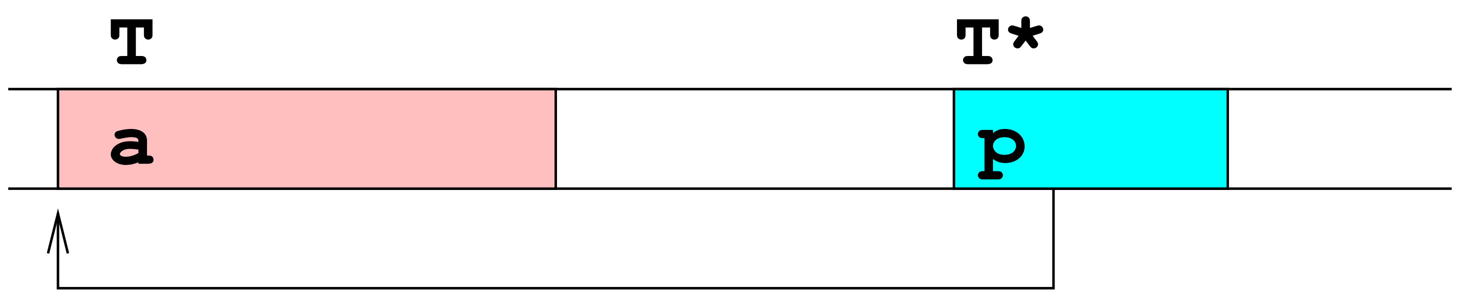
```
int
f(const int& i) { return i; }
int
f(int& i) { return ++i; }

int
main() {
const int c(5);
int d(5);
cout << f(c) << endl; // calls f(const int&); prints 5
cout << f(d) << endl; // calls f(int&); prints 6
}
```

pointers

```
NameOfType* Variable(InitialValue);
```

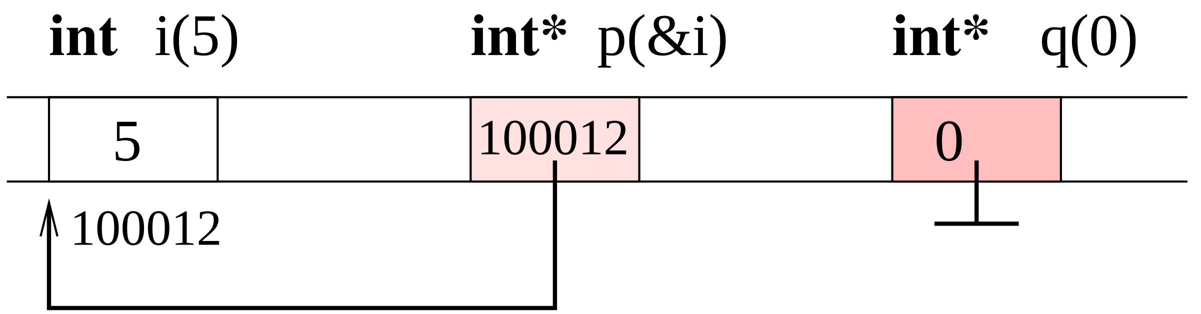
`NameOfType*` is the type of “addresses referring to objects of type **`NameOfType`**”.

```
T     a;
T*    p(&a);  // &a = address of a
T     b(*p);  // *p = dereferences p
```

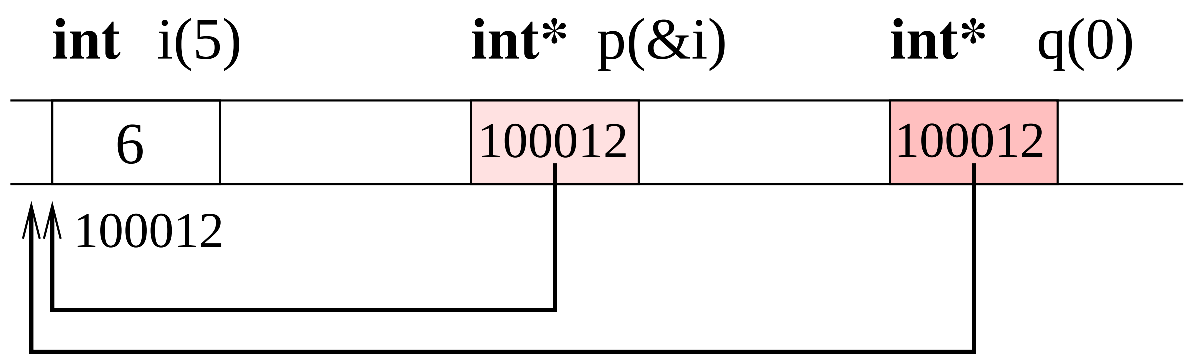


pointers example

```
int    i(5);
int*   p(&i);
int*   q(0);
```

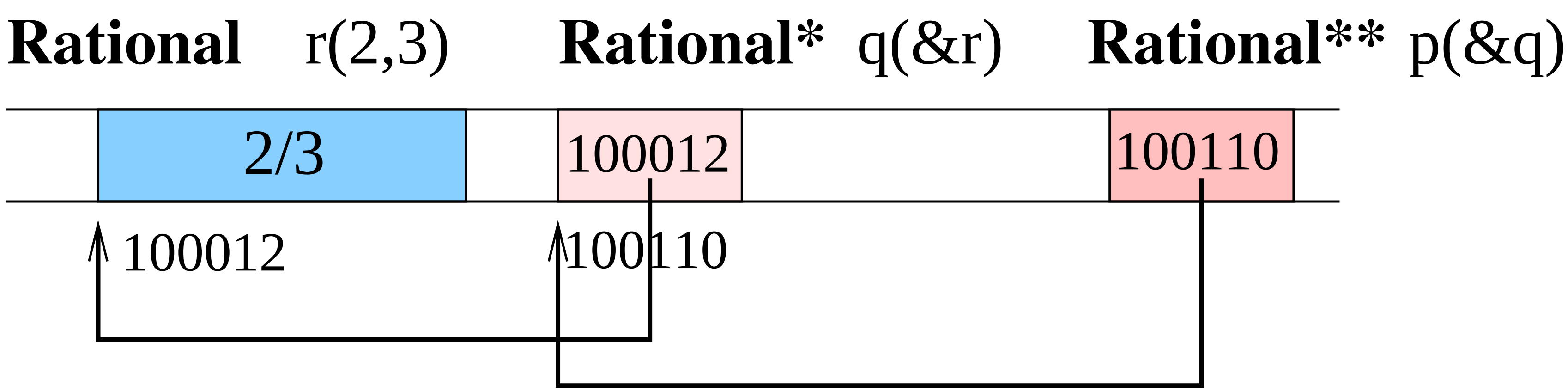



```
q = p;
*p = 6;
```



handles

```
Rational      r(2,3);
Rational*     q(&r);
Rational**    p(&q);

cout << **p + *q << "," << r << "," << *q << "," << **p;
cout << q->add(*q); // short for (*q).add(*q)
```




```
(*Expression).MemberName
Expression->MemberName
```

pointers as alternative for references

```
#include <iostream>

void
swap_p(int* px,int* py) {
int tmp(*px);
*px = *py; // copy contents of what py points to to area that px points to
*py = tmp;
}

void
swap_r(int& x,int& y) {
int tmp(x);
x = y; // copy contents of what y refers to to area that x refers to
y = tmp;
}
```

pointers as alternative for references

```
int
main() {
int        a(5);
int        b(6);

swap_p(&a,&b); // pass pointers to a, b
cout << a << ", " << b << endl; // prints 6, 5

swap_r(a,b);
cout << a << ", " << b << endl; // prints 5, 6
}
```

pointers and const

- forbidding modification of an object "through a pointer"

```
int            i(6);
const int*     p(&i);
*p = 5; // error
```

- forbidding modification of the pointer itself

```
int            j(4);
int * const    q(&i); // q is a constant pointer to i
*q = 5; // no problem: you can modify *q
q = &j; // error: you cannot modify q
```

- forbidding both

```
const int* const  pc(&i); // constant pointer to constant integer
```

pointers vs references

```
int          i(3);
int          &r(i); // reference must be initialized
int* const   p(&i);
```

A reference is like a constant pointer where dereferencing is automatic:

```
*p = 5; r = 5; // same effect

int    j;
p = &j; // ERROR, it is also impossible to make r refer to j
```

A pointer can however contain more information (NULL or not):

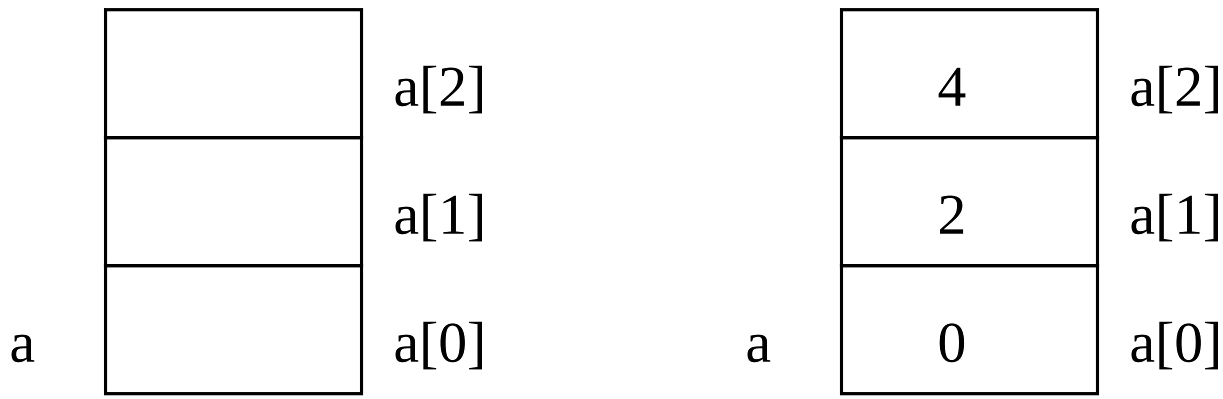
```
int    f(List* l); // l may be 0, i.e. not point anywhere
int    f(List& l); // l ALWAYS refers to a List object
```

the this pointer

```
class T {
   ReturnType f(ParameterList) {
      T* const this(PointerToTargetObject);
      // ...
      }
   ReturnType f(ParameterList) const {
      const T* const this(PointerToTargetObject);
      // ...
      }
}
```

```
Rational& // should return reference to target object to support x = y = z;
Rational::operator=(const Rational& r) {
numerator_ = r.num();
denominator_ = r.denom();
simplify();
return *this; // return reference to target object
}
```

arrays




```
const int SIZE = 3;
int       a[SIZE]; // array of 3 int objects
for (unsigned int i=0;(i<SIZE);++i) // array indices start from 0 to SIZE-1
          a[i] = 2*i;
for (unsigned int i=0;(i<SIZE);++i) // will print 0 2 4
          cout << a[i] << " ";
```

example: bubble sort an array

```
#include        <iostream>
#include        <string>
void
swap(string& x,string& y) {
string tmp(x);
x = y;
y = tmp;
}
const int       MAX_WORDS = 10;
string          words[MAX_WORDS];

int
main() { // read 10 strings from stdin and bubble-sort them
for (unsigned int i=0; (i<MAX_WORDS); ++i)
  cin >> words[i];
for (unsigned int size=MAX_WORDS-1;(size>0);--size)
  // find largest element in 0..size range and store it at size
  for (unsigned int i=0;(i<size);++i)
    if (words[i+1]<words[i])
      swap(words[i+1],words[i]);
for (unsigned int i=0; (i<MAX_WORDS); ++i)
  cout << words[i] << " ";
}
```

array initialization

```
#include        <iostream>

// compiler can figure out how large the array should be
float    vat_rates[] = { 0, 6, 20.5 };

int
main() {
// how to find the number of elements in vat_rates?
unsigned int size(sizeof(vat_rates)/sizeof(float));
const char message[] = "VAT rates"; // special case
cout << message;
for (unsigned int i=0;(i<size);++i)
  cout << " " << vat_rates[i];
cout << endl;
}
```

array initialization with default constructor

Arrays of class objects are initialized using default (without arguments) constructor.

```
class Rational {
public:
  Rational(int num=0,int denom=1): numerator_(num), denominator_(denom) {}
  // lots of stuff omitted
private:
  int     numerator_;
  int     denominator_;
};

Rational rationals[3]; // compiler wil call Rational::Rational() on each element
Rational more_rationals[] = { Rational(1,2), Rational(1,3) };
```

passing arrays as parameters

- Arrays are passed "by reference"
- The compiler doesn't care about the size of the array (but the programmer should!)

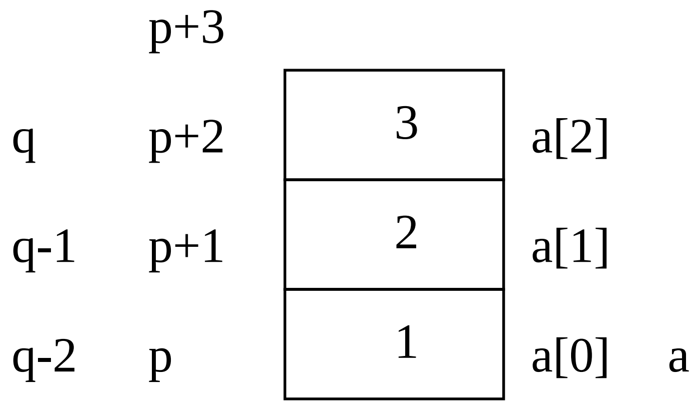
```
#include <iostream>

int
sum(int a[],unsigned int size) {
int total(0);
for (unsigned int i=0;(i<size);++i)
  total += a[i];
return total;
}

int
main() {
int numbers[] = { 1 , 2, 3, 4, 5 };
cout << sum(numbers,sizeof(numbers)/sizeof(int)) << endl;
}
```

arrays vs pointers

A pointer can be made to point to an array; a pointer can also be indexed.

```
#include <iostream>

void
f(int x[]) // can be called with an array or a pointer parameter
{
x[0] = 1;
}

int
main() {
int     a[] = { 0, 2, 3 };
int*    p(a);
int*    q(&a[0]); // exactly the same
f(p); // passing a pointer or an array is the same
cout << *p << ", " << a[0] << endl; // prints 1,1
for (unsigned int i=0;(i<sizeof(a)/sizeof(int));++i)
  cout << p[i] << ", " << a[i] << endl; // prints 1, 1\n 2, 2\n 3, 3
}
```

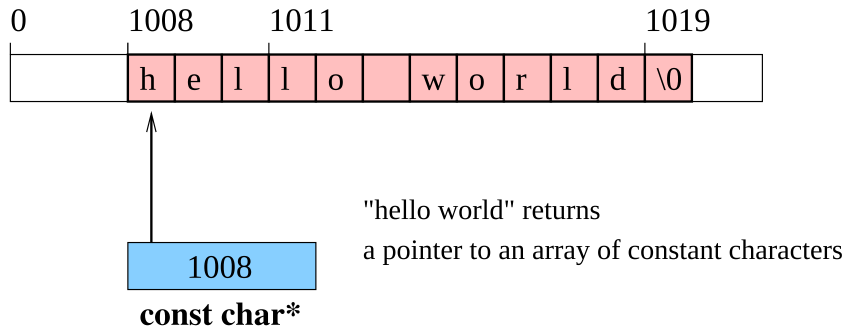
pointer arithmetic



Pointers can be assigned, integers can be added (subtracted).

```
#include <iostream>

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

int
main() {
int* p(a);
int* q(p+2);
int* s(q−2);
for (unsigned int i=0;(i<3);++i)
  cout << *p++ << "\n"; // what happens?
cout << q − p << endl; // prints -1
}
```

C-strings



C strings (literal strings) are arrays of characters with a 0 after the last character.

```
#include <iostream>
void
print(ostream& os,const char*p) {
while (*p)
  os << *p++;
os << endl;
}

int
main() {
const char* s("hello world");
print(cout,s);
}
```

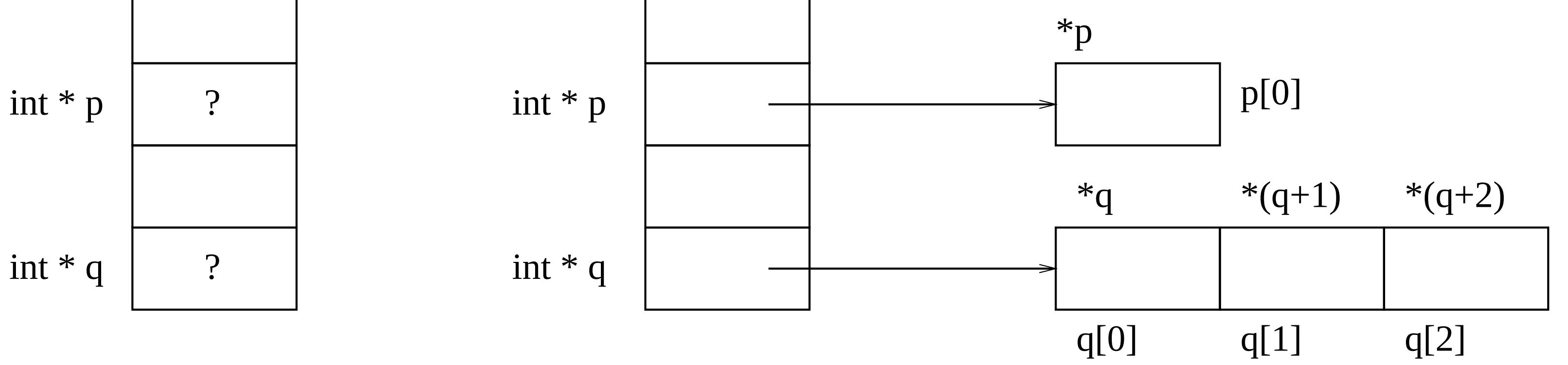
comparing C-strings

```
#include <iostream>
int
strcmp(const char*s1,const char* s2) {
// returns
//      0      if s1 and s2 are (lexicographically) equal
//      >0     if s1 is lexicographically larger than s2
//      <0     if s1 is lexicographically smaller than s2
while ((*s1) && (*s2) && (*s1==*s2)) {
  ++s1; ++s2;
  }
return *s1 - *s2;
}

int
main() {
const char*     s1("abc");
const char*     s2("abcde");
cout << strcmp(s1,s2) << endl; // prints -100
}
```

command line processing

```
#include    <iostream>
#include    <stdlib.h> // for atoi()

// sum: this program adds computes the sum of its commandline arguments
// usage: sum int..

int
main(unsigned int argc,char* argv[]) {
// argv is an array of pointers to (arrays of) char, one for each argument
// argv[0] is the name of the program, i.e. the first word in the command line
// argc is the number of arguments
int    sum(0);

for (unsigned int i=1;(i<argc);++i)
  sum += atoi(argv[i]); // atoi(const char*) converts a string to an int

cout << sum << endl;
}
```

explicit memory allocation

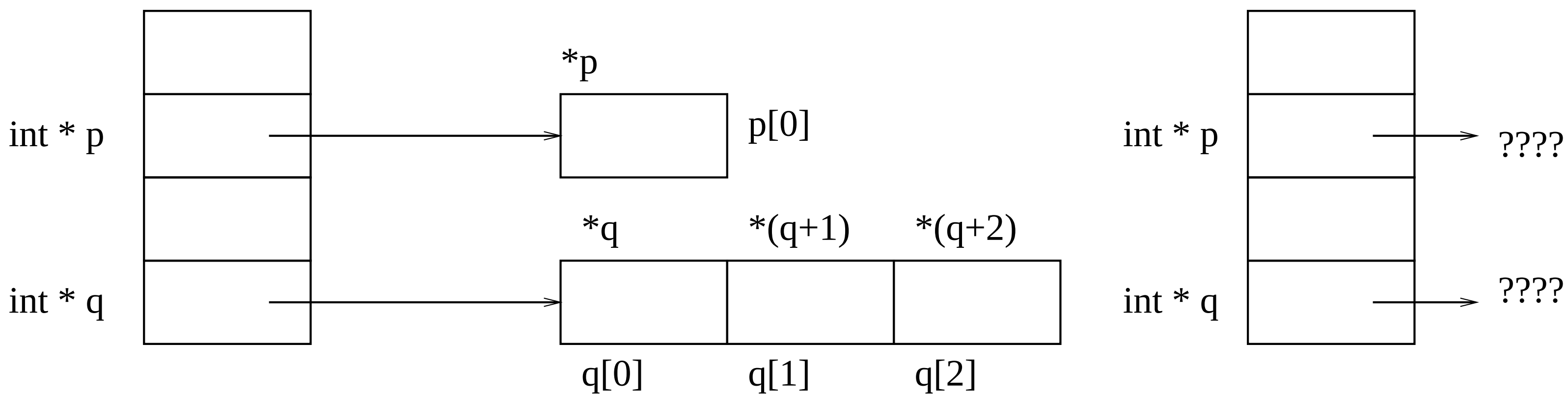


int* p; // *not initialized*
int* q; // *not initialized*

p = **new int**; // *allocate memory for 1 new integer*
q = **new int**[3]; // *allocate memory for 3 new integers*

The explicitly allocated memory does not go away until the programmer explicitly deallocates it.

explicit memory deallocation



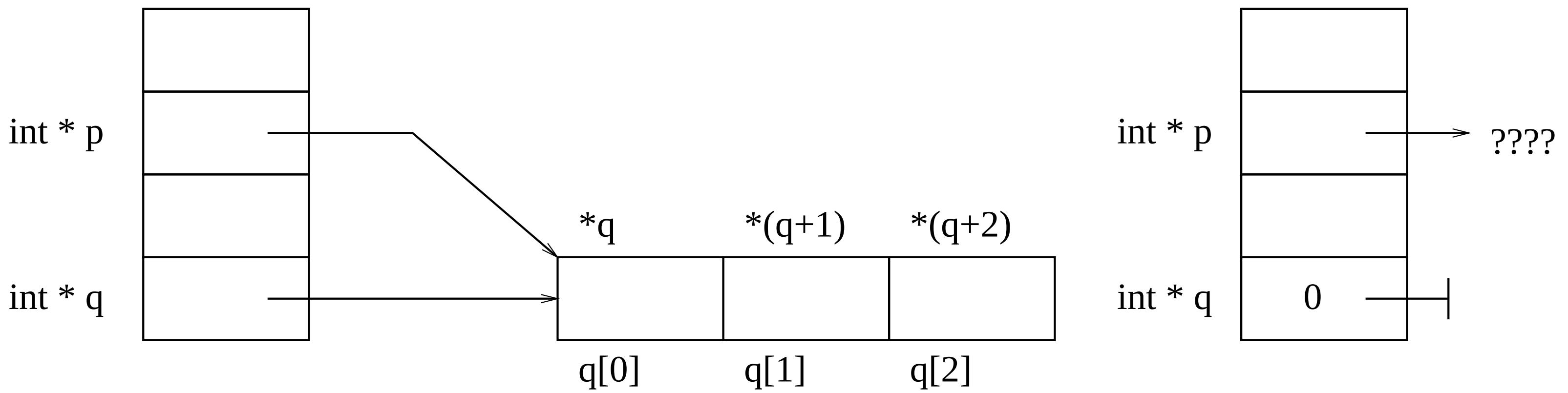
int* p; // *not initialized*
int* q; // *not initialized*

p = **new int**; // *allocate memory for 1 new integer*
q = **new int**[3]; // *allocate memory for 3 new integers*

delete p; // *deallocate memory that was allocated using new*
delete [ ] q; // *deallocate memory that was allocated using new [ ]*

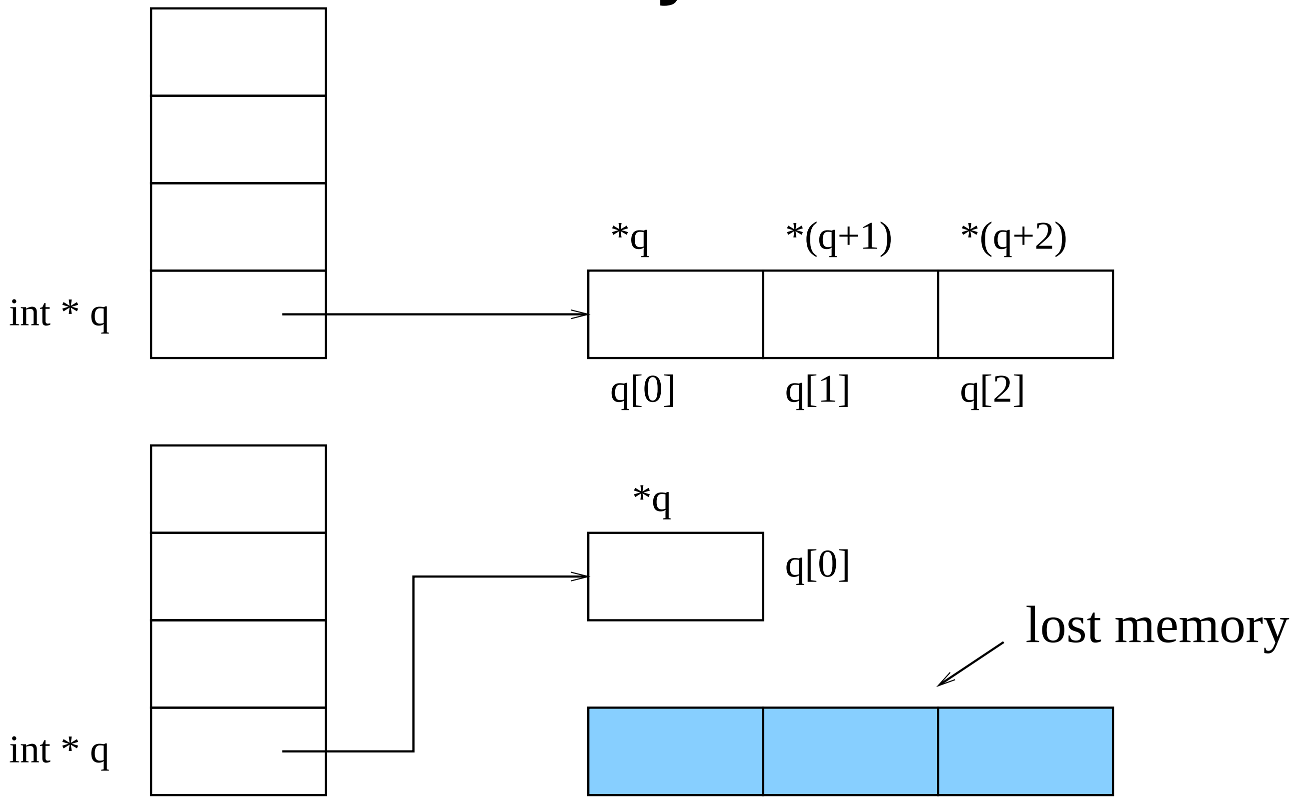
After delete pointers are left “**dangling**”

the dangling pointer syndrome




```
int*     q(new int[3]);
int*     p(q);
delete [] q;
q = 0; // p is left dangling!
```

memory leaks



int$*$ q(**new** **int**[3]);
q = **new** **int**;

The originally allocated memory cannot be referenced anymore!

object taxonomy w.r.t. memory management

name	how defined	when initialized	when destroyed
static	static var in function body	first call of function	program exit
	static class member	program startup	program exit
	global variable	program startup	program exit
automatic	local var in function body	when definition is executed	exit scope
member	data member of class	together with owner	together with owner
free	using new/delete	determined by programmer	determined by programmer

example: a Ztring class

```
#ifndef MYSTRING_H
#define MYSTRING_H
#include <iostream>

class Ztring {
public:
        Ztring(const char* cstring=0);
        Ztring(const Ztring&); // copy constructor
        ~Ztring(); // destructor

        Ztring& operator=(const Ztring&);

        const char*     data() const; // why return const char*?
        unsigned int    size() const;
        void            print(ostream&) const;
        void            concat(const Ztring&); // concatenates to *this
private:
        char*           data_; // a C string, i.e. a 0-terminated array of char
        char*           init(const char* string);
};
```

Ztring overloaded operators

```
// Auxiliary functions: overloaded operators
// E.g.
// Ztring x("abc");
// Ztring y("def");
// cout << x + y << endl;

Ztring
operator+(const Ztring& s1,const Ztring& s2);
ostream&
operator<<(ostream& os,const Ztring& s);
#endif
```

Ztring implementation: constructors and destructor

```
#include      "ztring.h"
#include      <strings.h>

// constructors and destructor

Ztring::Ztring(const char* cstring): data_(init(cstring)) {
}

Ztring::Ztring(const Ztring& s): data_(init(s.data())) {
}

Ztring::~Ztring() {
delete [] data_; // very important: avoid memory leak
}
```

Ztring implementation: assignment operator

```
// assignment operator
Ztring&
Ztring::operator=(const Ztring& s) {
if (this==&s) // why?
  return *this;
delete [] data_; // avoid memory leak
data_ = init(s.data());
return *this;
}
```

Ztring implementation: inspectors

```
// constant public member functions
const char*
Ztring::data() const {
return data_;
}

unsigned int
Ztring::size() const {
if (data_==0)
  return 0;
return strlen(data_);
}

void
Ztring::print(ostream& os) const { // can be made shorter; how?
for (unsigned int i=0;(i<size());++i)
  os << data_[i];
}
```

Ztring implementation: concatenation

```
void
Ztring::concat(const Ztring& s) {
// save old data of this ztring
unsigned int      old_size(size());
char*    old(data_);
// allocate buffer large enough to hold both + trailing '\0'
data_     = new char[old_size+s.size()+1];

unsigned int j(0);
for (unsigned int i=0;(i<old_size);++i)
  data_[j++] = old[i]; // copy original string
for (unsigned int i=0;(i<s.size());++i)
  data_[j++] = s.data_[i]; // after that the argument string
data_[j] = '\0'; // don't forget trailing '\0' character

delete [] old; // avoid memory leak
}
```

Ztring implementation: private member functions

```
// private member functions

char*
Ztring::init(const char* s) {
if (s==0)
  return 0;
else {
  unsigned int len(strlen(s)+1); // why +1?
  char* p(new char[len]);
  for (unsigned int i=0;(i<len);++i)
    p[i] = s[i];
  return p;
  }
}
```

Ztring implementation: auxiliary functions

```
// auxiliary functions (overloaded operators)

Ztring
operator+(const Ztring& s1,const Ztring& s2) {
Ztring    s(s1);
s.concat(s2);
return s;
}

ostream&
operator<<(ostream& os,const Ztring& s)
{
s.print(os);
return os;
}
```

gang of three rule

If a class C contains dynamically allocated data members, it should have:

- A copy-constructor `C::C(const C&)` (to avoid unwanted sharing of data).
- A tailored assignment operator `C& C::operator=(const C&)` (to avoid unwanted sharing of data).
- A destructor `C::~C()` (to avoid memory leaks).

overloading new, delete

```
class Rational { // ADT representing rational numbers
// ..
public:
  void*        operator new(size_t) { return pool_.alloc(); }
  void         operator delete(void* p,size_t size) {
                 assert(size==sizeof(Rational)); // sanity check
                 if (p) // do not attempt to delete a null pointer
                   pool_.dealloc(p);
                 }
// ..
private:
  static Pool pool_;
// ..
};

Rational*      p(new Rational(1,3)); // will allocate from Rational::pool_
```

Pool for allocating free Rational objects

```
#ifndef POOL_H
#define POOL_H
#include "rational.h"
class Pool { // a pool of reusable areas, each of sizeof(Rational)
public:
  Pool(unsigned int size); // size is number of areas in Pool
  ~Pool();
  bool is_full() const { return free_<0; }
  Rational*     alloc(); // return pointer to area available for Rational
  void          dealloc(void* p); // deallocate a Rational area
private:
  Pool(const Pool&); // forbidden
  Pool& operator=(const Pool&); // forbidden
  Rational*     slots_;
  int*          next_; // if i is the index of a free slot, then next_[i] is the
                       // index of another free slot or -1
  int           free_; // index of first free slot, <0 if the pool is full
};
#endif
```

pool constructor, destructor

```
#include <stdlib.h> // for abort()
// note: Rational::operator new[ ] is not overloaded
Pool::Pool(unsigned int size): slots_(new Rational[size]),
  next_(new int[size]), free_(0) {
// initially, the free list is 0, 1, 2, ..
for (unsigned int i=0;(i<(size-1));++i)
  next_[i] = i+1;
next_[size-1] = -1; // end of free list
}

// note: Rational::operator delete[ ] is not overloaded
Pool::~Pool() {
delete [] next_;
delete [] slots_;
}
```

(de)allocation from a Pool

```
Rational*
Pool::alloc() {
if (is_full())
  abort();
Rational* r(&slots_[free_]); // address of first free Rational area
free_ = next_[free_]; // update start of free list
return r;
}

void
Pool::dealloc(void* p) { // deallocate a Rational area
// compute index of pointer p in slots_ array
int index(static_cast<Rational*>(p)-slots_);
        // static_cast converts ''related'' types, e.g. void* to Rational*
next_[index] = free_; // add index of deallocated area to front of free list
free_ = index;
}
```

smart pointers

```
class Url; // defined elsewhere
class HtmlPage { // a page is uniquely identified by its URL
friend class Proxy;
public:
  string title() const;
private:
  static HtmlPage* fetch(const Url&); // retrieve the page corresponding with a Url
  HtmlPage(const HtmlPage&); // forbid copy constructor
  HtmlPage& operator=(const HtmlPage&); // forbid assignment
};
class Proxy {
public:
  Proxy(const string& url): key_(url) {}
  HtmlPage* operator->() const { return HtmlPage::fetch(url()); }
  const Url&   url() const { return key_; }
private:
  Url    key_;
};
Proxy              proxy("http://tinf2.vub.ac.be/index.html");
proxy->title();
```

why templates

Suppose we have written a function

```
void sortints(int a[]);
```

and now we need a function

```
void sortstrings(string a[]);
```

Most of sortstrings can be duplicated from sortints (only the type of the things we compare, move will be different).
We want to be able to write 1 function

```
void sort(T a[])
```

which will work for **any** type T.

```
strings  sa[];   sort(sa); // should work
int      ia[];   sort(ia); // should work
```

template functions

```
#include <iostream>
#include <string>
template<typename T>
void
swap(T& x,T& y) {
T tmp(x);
x = y;
y = tmp;
}

template <typename T>
void
bubble_sort(T a[],unsigned int total_size) {
for (unsigned int size=total_size-1;(size>0);--size)
  // find largest element in 0..size range and store it at size
  for (unsigned int i=0;(i<size);++i)
    if (a[i+1]<a[i])
      swap(a[i+1],a[i]);
}
```

What are the (hidden) requirements on T?

template functions

```
int
main(unsigned int argc,char* argv[]) {
unsigned int      size = argc-1;
cerr << "size=" << size << endl;
string    *args = new string[size];
for (unsigned int i=0;(i<size);++i)
  args[i] = string(argv[i+1]); // why i+1?
bubble_sort<string>(args,size);
bubble_sort(args,size); // will also work
for (unsigned int i=0;(i<size);++i)
  cerr << "args[" << i << "] = " << args[i] << endl;
}
```

overloading template functions

```
template <typename T>
T
maximum(T x1,T x2) { return (x1 > x2 ? x1 : x2 ); }

template <typename U>
U*
maximum(U* p1,U* p2) { return (*p1 > *p2 ? p1 : p2 ); }

const char*
maximum(const char* s1,const char* s2) {
return (strcmp(s1,s2)>0 ? s1 : s2 );
}

double d1(1.23);
double d2(4.5);

maximum(d1,d2);
maximum(&d1,&d2);
maximum("abc","abcd");
```

overloading and specialization of template functions

To resolve an overloaded function call, more specialized functions (or function templates) that better match the actual call's parameters are to be preferred.

In a readable program, a call's resolution should be clear from this principle alone.
Explicit specialization: cfr. text (not supported in egcs-2.91.66)

template classes

```
#ifndef ARRAY_H
#define ARRAY_H

#include        <assert.h>

// a safe array: subscripts are checked
template <class T>
class Array {
public:
  Array(unsigned int size);
  Array(const Array&); // gang of three
  ~Array();

  unsigned int   size() const;

  // overloaded operator[ ], will check legality of index
  T& operator[](unsigned int i); // why two versions?
  const T& operator[] (unsigned int i) const;
private:
  T*             data_;
  unsigned int   size_;
  Array& operator=(const Array&); // we forbid assignment!
};
```

Array implementation: constructors

```
// constructors

template <class T>
Array<T>::Array(unsigned int size): data_(new T[size]), size_(size) {
}

template <class T>
Array<T>::Array(const Array& a): data_(new T[a.size()]), size_(a.size()) {
for (unsigned int i=0;(i<size_);++i)
  data_[i] = a[i];
}

// destructor
template<class T>
Array<T>::~Array() {
delete [] data_;
}
```

Array implementation: inspectors

```
// inspector functions

template <class T>
unsigned int
Array<T>::size() const {
return size_;
}

template <class T>
const T&
Array<T>::operator[](unsigned int i) const {
assert(i<size_);
return data_[i];
}
```

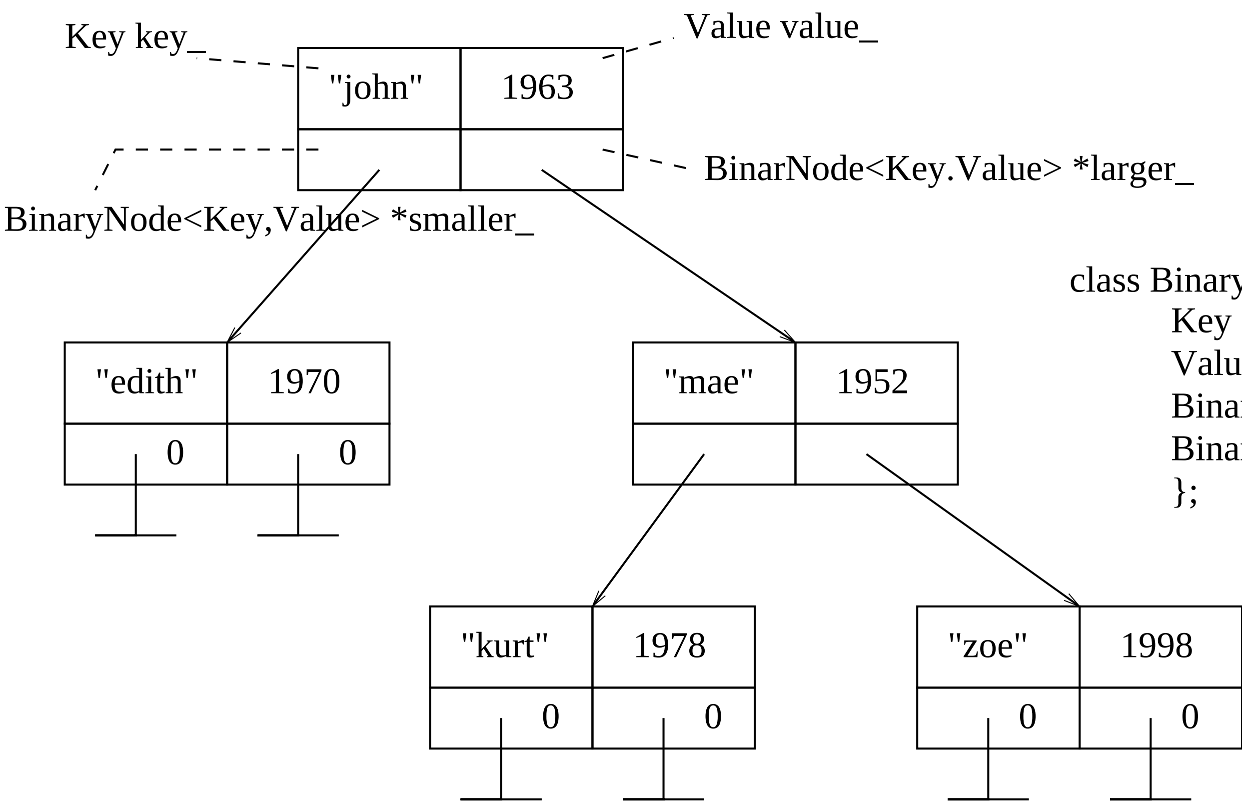
Array implementation: indexing for assignment

```
// non-const operator[ ]: can be used as in a[i] = ..

template <class T>
T&
Array<T>::operator[ ](unsigned int i) {
assert(i<size_);
return data_[i];
}
#endif
```

Array usage example

```
#include <string>
#include "array.h"
Array<string>
f(const Array<string> a) { // to test copy-ctor
return a;
}

int
main(int argc,char *argv[]) {
unsigned int    size = argc-1;

Array<string>   a(size);

for (unsigned int i=0;(i<size);++i)
   a[i] = string(argv[i+1]);

for (unsigned int i=0;(i<size);++i)
   cout << f(a)[i] << endl;
}
```

Binary search trees




```
class BinaryNode<Key,Value> {
        Key       key_;
        Value     value_;
        BinaryNode<Key,Value> *smaller_;
        BinaryNode<Key,Value> *larger_;
        };
```

The BinaryTree template class

```
#ifndef BINTREE_H
#define BINTREE_H

template <class Key,class Value>
class BinTree {
private:

  // Node represents a node in the binary search tree
  // There is one Node class for every instantiation of BinTree
  struct Node { // everything is public, but the whole class is private
    Node(Key k,Value v,Node* smaller=0,Node *larger=0): key_(k),val_(v), smaller_(smaller),
        larger_(larger) {}

    Key     key_;
    Value   val_;
    Node    *smaller_;
    Node    *larger_;
    };

public:
  BinTree(): root_(0) {}
  ~BinTree() { zap(root_); root_ = 0; }
```

```
// find returns true and fills in Value if the node with key can be found
bool    find(const Key& key,Value& val) const {
  Node*  node = find_node(root_,key); // find a node containing key
  if (node) { // we found the node, get the value
    val = node->val_;
    return true;
    }
  else  return false; // no such node, bad luck,
  }

// insert inserts (key,val) in tree, replacing old value for key is necessary
void    insert(const Key& key,const Value& val) { insert_node(root_,key)->val_ = val; }

// nice shorter way to insert: t[key] = val, so operator[ ] should return a reference
Value&  operator[ ](const Key& key) { return insert_node(root_,key)->val_; }

private:
  BinTree(const BinTree&); // forbid copy constructor
  BinTree&  operator=(const BinTree&); // forbid assignment
```

```
// zap(node) deallocates memory used by subtree starting at node
void   zap(Node* node) {
  if (node==0)
    return;
  zap(node->smaller_);
  zap(node->larger_);
  delete node;
  }

// find_node returns a pointer to a node containing key, 0 if such a node cannot be found
Node*  find_node(Node* node,const Key& key) const {
  if (node==0)
    return 0;
  if (node->key_ == key)
    return node; // got it
  else
    if (key<node->key_)
      return find_node(node->smaller_,key);
    else
      return find_node(node->larger_,key);
  }
```

```
// insert_node returns node where key should be; if key cannot be found, it returns
// a (pointer to a) fresh node
// insert_node will also properly update the tree if it needs to create a new
// node; this is why the first parameter is a reference to a pointer
Node*  insert_node(Node*& node,const Key& key) {
  if (node==0)
    // don't forget: where there was a 0-pointer,
    // there will now be pointer to a fresh node
    return (node = new Node(key,Value())); // note default constructor for Value
  if (key==node->key_)
    return node;
  if (key<node->key_)
    return insert_node(node->smaller_,key);
  else
    return insert_node(node->larger_,key);
  }

// data member: the root of the binary search tree
Node*  root_;
};
#endif
```

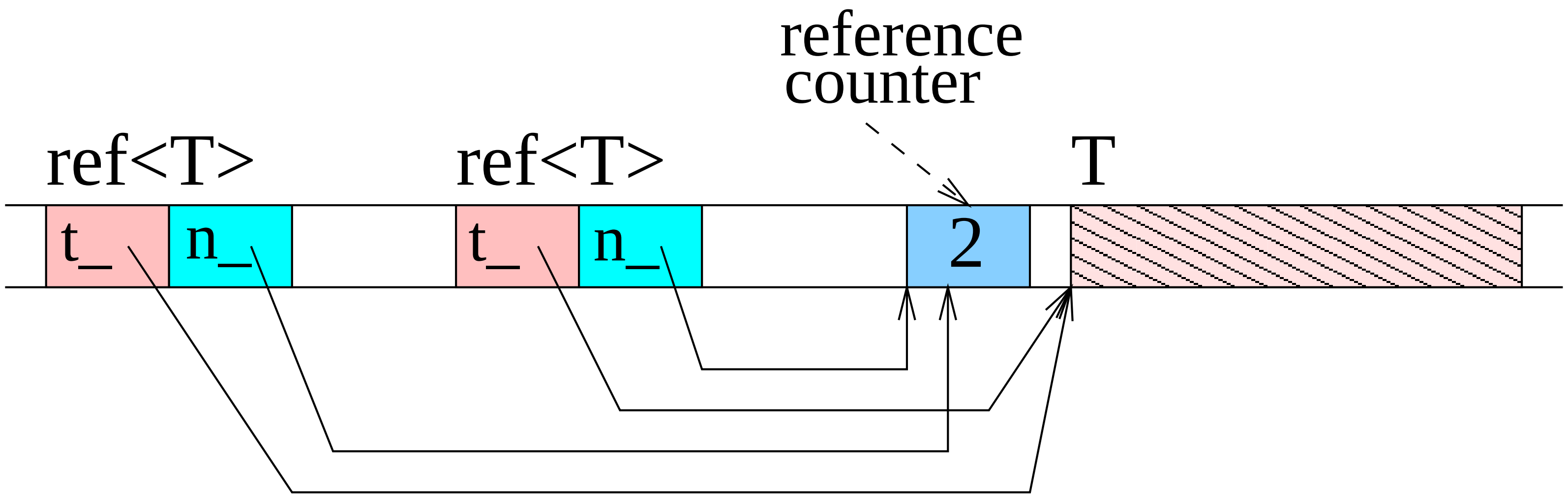
example program using BinaryTree

```
#include     <string>
#include     <iostream>

#include     "bintree.h"

int
main(int argc,char* argv[]) {
BinTree<string,unsigned int>     birth_year;

birth_year.insert("john",1936);          birth_year.insert("edit",1970);
birth_year.insert("mae",1952);           birth_year.insert("kurt",1978);
// alternative way to insert
birth_year["zoe"] = 1998;                birth_year["john"] = 1963;

unsigned int     year(0);

if (birth_year.find("john",year))
  cout << "birth year of john = " << year << endl;
}
```

reference-counted pointers

```
#ifndef REF_H
#define REF_H

template<typename T>
class Ref { //  a reference counting encapsulation of a pointer.
friend class T;
public:
  Ref(): t_(0), n_(new unsigned int(1)) {} // a null pointer
  Ref(const Ref& r): t_(r.t_), n_(r.n_) { ++*n_; } // copy ctor increments shared counter *n_
  ~Ref() { if (--*n_) return; delete t_; delete n_; } // destructor decrements shared counter *n_
  operator        bool() const      { return (t_!=0); }
  T&              operator*() const { return *t_; }
  T*              operator->() const { return t_; }
  Ref&            operator=(const Ref& r); // similar to copy ctor
  bool            operator==(const Ref& r) const { return t_ == r.t_; }
  bool            operator<(const Ref& r) const { return t_ < r.t_; }
private:
  Ref(T* t): t_(t), n_(new unsigned int(1)) {} // only T can produce references to itself
  T*              t_; // the encapsulated pointer
  unsigned int*   n_; // reference count for *t_
};
```

```
template <typename T>
inline Ref<T>&
Ref<T>::operator=(const Ref& r) {
if (this=&r) // check for self-assignment
  return *this;
if (--*n_==0) {  // target is last pointer to *t_: delete it
  delete t_; delete n_;
  }
t_ = r.t_;  n_=r.n_; ++*n_;
return *this;
}
#endif
```



ref-counted pointers example

```
#include        <iostream>
#include        <string>
#include        "refcnt.h"
class Huge {
friend class Ref<Huge>; // why needed?
public: // lots of stuff omitted
  // only private constructors; the following FACTORY METHOD returns
  //    a reference-counted pointer to a Huge object
  static Ref<Huge> create(const char *s) { return Ref<Huge>(new Huge(s)); }
private:
  string                  data_;
  Huge(const char* s): data_(s) {}
  ~Huge()         { cerr << "Huge::~Huge()" << endl; }
};

int
main(int,char **) {
Ref<Huge>       r = Huge::create("c++");
Ref<Huge>       p(r);
p = r; // copies only reference and increments reference count
}
```

ref-counted pointers pro and con

+ automatic memory management à la Java: just create, don't worry about deletion
+ low overhead (compared to some garbage collection algorithms)
- not for circular structures (why?); but these do not occur often in practice

motivation: sequential search

```
template <typename T>
const T*
find(const T* first,const T* last,const T& value) {
// sequential search for value in [*first..*last[; return last if not found
while (first!=last && *first!=value)
  ++first;
return first;
}

extern int a[SIZE];
find(a,a+SIZE,20); // return pointer to 20 or a+SIZE
```

```
template <typename T>
class Node { // Node* is a (too) simple linked list
public:
  Node(const T& t): value_(t), cdr_(0) {} // you cannot make an empty list
  Node* cons(const T& t) { return new Node(t,this); } // prepend
  const T&      car() const { return value_; } // non const version missing
  Node* cdr() const { return cdr_; } // non const version missing
private:
  Node(const T& t,Node* cdr): value_(t), cdr_(cdr) {}
  T             value_;
  Node*         cdr_;
};
template <typename T>
Node<T>*
find(Node<T>* first, Node<T>* last,const T& value) {
// sequential search for value in linked list [first .. last[
while (first!=last && first->car()!=value)
  first = first->cdr();
return first;
}
extern Node<int> *list;
find(list,static_cast<Node<int>*>(0),20);
```

the essence of sequential search

```
Cursor
find(Cursor start,Cursor end, T value) {
while (start!=end && (object-pointed-to-by-start != value) )
  advance start-cursor to next position in container;
return start
}
```

Requirements for this to work (in addition to constructors):

```
class Cursor { // + constructors
public:
  T operator*(); // dereference to obtain an object from the container
  Cursor operator++(); // return cursor that refers to the next object in the container
  bool operator!=(Cursor); // compare cursors for (non)equality
  Cursor& operator=(const Cursor&); // cursors must be assignable
};
```

a generic find function

A *generic* algorithm is "pure", i.e. independent of data types on which it operates.

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

This works

- For arrays, **`InputIterator`** is **`T*`**
- For linked list, we must implement a **`Node<T>::Cursor`** class that implements the requirements.

linked list iterator

```
template <typename T>
class Node { // Node* is a (too) simple linked list
public:
  Node(const T& t): value_(t), cdr_(0) {} // you cannot make an empty list
  Node*        cons(const T& t) { return new Node(t,this); } // prepend
  const T&     car() const { return value_; } // non const version missing
  Node*        cdr() const { return cdr_; } // non const version missing
  class Cursor { // satisfies requirements for find
  public:
    Cursor(Node* node=0): node_(node) {}
    const T&   operator*() const { assert(node_); return node_->car(); }
    Cursor&    operator++() { assert(node_); node_ = node_->cdr(); return *this; }
    Cursor     operator++(int) { Cursor tmp(*this); ++*this; return tmp; }
    bool       operator==(const Cursor& c) const { return node_ == c.node_; }
  private:
    Node*      node_;
  };
private:
  Node(const T& t,Node* cdr): value_(t), cdr_(cdr) {}
  T            value_;
  Node*        cdr_;
};
```

linked list iterator example

```
extern Node<int> *list;
find(Node<int>::Cursor(list),Node<int>::Cursor(),20);
```

If we define

```
Node<T>::Cursor
Node<T>::begin() { return Cursor(this); }
Node<T>::Cursor
Node<T>::end() { return Cursor(); }
```

then

```
find(list->begin(),list->end(),20);
```

will work.

containers and algorithms

The STL provides many container template classes:

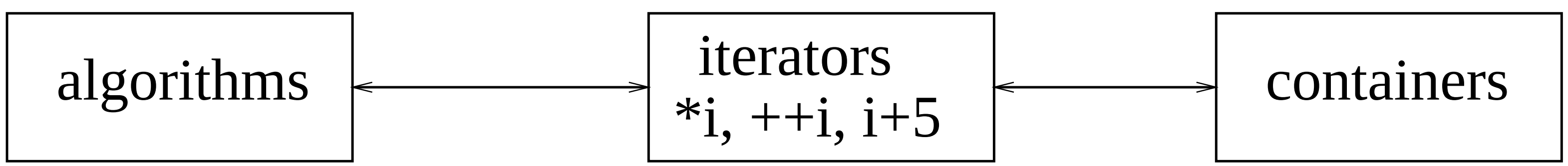
- **sequences**: vector, list, dequeue
- **associative**: set, multiset, map, multimap, hash table
- **adapters**: stack, queue, priority queue

It also supports more than 50 algorithms that use these containers:

- non-mutating sequence operations: e.g. find, find_if, for_each, ...
- mutating sequence operations: e.g. copy, replace, erase, ...
- sorting operations
- ...

why iterators?

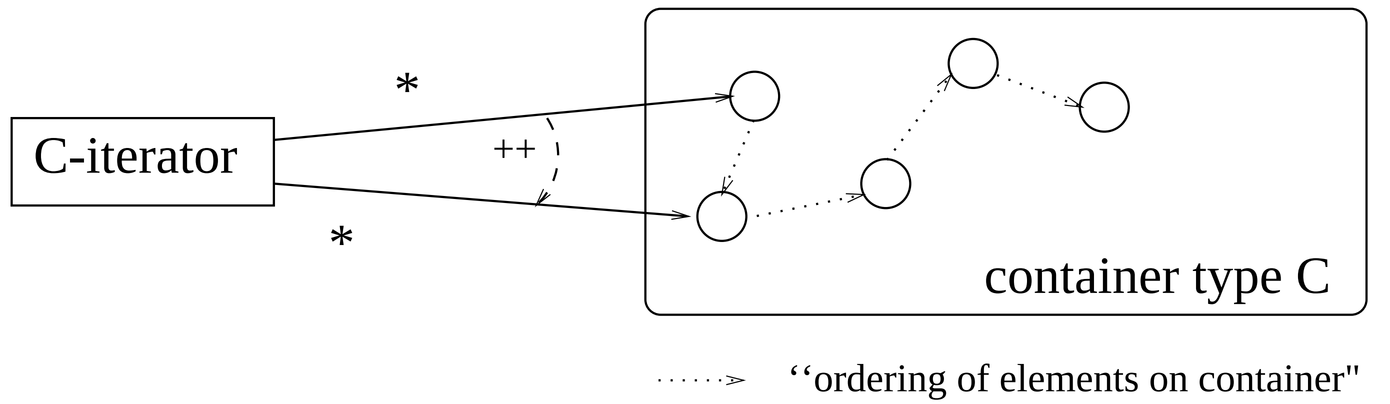
- To implement each algorithm for each container would require hundreds of implementations.
- **Why not write algorithms that work for many containers?**
- This is possible by putting a new abstraction between algorithms and containers (data structures): **iterators**



An iterator is like an **abstract pointer**: it may support

- dereference (using `operator*()`): `*it`
- increment, decrement: `++it`, `--it`, `it++`, `it--`
- random access: `it+5`

a container and its iterator



template <**class** Element>
class C {
    CIterator   begin(); // *‘‘pointer” to first element in C*
    CIterator   end(); // *‘‘pointer” ‘‘beyond last” element in C*
    . .
};
template <**class** Element>
class CIterator {
public:
    Element   **operator**$*$();
    CIterator   **operator++**();
};

a simple algorithm using containers

```
template <class Iterator,class T> // works for any iterator on any container
Iterator // return iterator pointing to cell containing value or last
find(Iterator first,Iterator last,const T& value) {
        while (first != last && *first != value)
                ++first;
        return first;
}

C<string>       container;
string          s("abc");
CIterator       it;

if ((it=find(container.begin(),container.end(),s))==container.end())
        cout << s << " not found" << endl;
else
        cout << s << " found: " << *it << endl;
```

kinds of iterators

- Some containers (e.g. singly-linked list) do not support random access, others do.
- STL considers 5 kinds of iterators with increasing functionality:
  - input iterator: read-only access (**`x=*it`**), **`it++`**,**`++it`**
  - output iterator: write-only access (**`*it=x`**), **`it++`**, **`++it`**
  - forward iterator: read-write access, **`it++`**, **`++it`**
  - bidirectional iterator: forward + **`it--`**, **`--it`**
  - random access iterator: bidirectional + **`it[4]`**, **`it+14`**
- Some algorithms require a certain kind of iterator: what kind is required by **`find()`**?
- pointers are random access iterators!

```
extern string a[];          extern unsigned int a_size;
find(a,a+a_size,string("abc"));
```

example algorithm: sum

```
template <class InputIterator,class T>
T
sum(InputIterator first,InputIterator last) {
assert(first!=last);
T       result(*first++);
while (first!=last)
  result += *first++;
return result;
}
```

does not work:

```
extern Node<int>*       l;
typedef Node<int>::Cursor       l_iterator;
sum(l->begin(),l->end()); // error; compiler cannot deduce T=int
sum<l_iterator,int>(l->begin(),l->end()); // ok
```

ugly fix

```
template <class InputIterator,class T>
T
sum(InputIterator first,InputIterator last,T& result) {
if (first==last)
  return result;
do
  result += *first++;
while (first!=last);
return result;
}

int r;
extern Node<int>*        l;
sum(l->begin(),l->end(),r); // ok
```

Now it works (why?)

associating types with iterators

```
template <typename T>
class Node { // Node* is a (too) simple linked list
public:
  // ...
  class Cursor {
  public:
    typedef T value_type; // type to which cursor refers
    Cursor(Node* node=0): node_(node) {}
    const T&    operator*() const { assert(node_); return node_->car(); }
    Cursor&     operator++() { assert(node_); node_ = node_->cdr(); return *this; }
    Cursor      operator++(int) { Cursor tmp(*this); ++*this; return tmp; }
    bool        operator==(const Cursor& c) const { return node_ == c.node_; }
  private:
    Node*       node_;
  };
private:
  // ...
};
```

sum revisited

Needs only 1 (deducible) template parameter:

```
template <class InputIterator>
typename InputIterator::value_type // what is "typename"?
sum(InputIterator first,InputIterator last) {
assert(first!=last);
typename InputIterator::value_type result(*first++);
while (first!=last)
  result += *first++;
return result;
}

extern Node<int>*        l;
sum(l->begin(),l->end()); // ok
extern int          a[10];
sum(a,a+10); // error! why?
```

iterator types problem

Q Iterators may not be classes: e.g. what is **`value_type`** for a pointer type?

A Use a compile-time function to compute **`T::value_type`** from **`T`**. This can be done using template classes and partial specialization.

```
template <class Iterator>
struct iterator_traits { // default; ok for iterator class types
  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;
};
```

specializing iterator traits

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

template <class InputIterator>
typename iterator_traits<InputIterator>::value_type
sum(InputIterator first,InputIterator last) {
assert(first!=last);
typename iterator_traits<InputIterator>::value_type result(*first++);
while (first!=last)
  result += *first++;
return result;
}
int       a[10];
sum(a,a+10);  // ok; why?
```

dispatching on iterator category

```
template <class InputIterator>
inline void // version for input iterators and forward iterators
advance(InputIterator& i,
   typename iterator_traits<InputIterator>::difference_type n,
   input_iterator_tag) {
for (; n>0 ; --n)
  ++i;
}

template <class BidirectionalIterator>
inline void // version for bidirectional iterator
advance(BidirectionalIterator& i,
    typename iterator_traits<BidirectionalIterator>::difference_type n,
    bidirectional_iterator_tag) {
if (n>=0)
  for (; n>0 ; --n) ++i;
else
  for (; n<0 ; ++n) --i;
}
```

```
template <class RandomAccessIterator>
inline void // version for random access iterators
advance(RandomAccessIterator& i,
    typename iterator_traits<RandomAccessIterator>::difference_type n,
    random_access_iterator_tag) {
i += n;
}

template <class InputIterator>
inline void         // the general version dispatches to a more specialized one,
                    //   depending on the iterator kind
advance(InputIterator& i,typename iterator_traits<InputIterator>::difference_type n) {
advance(i,n,typename iterator_traits<InputIterator>::iterator_category());
}
```

“advance” call resolution

```
advance(I& i,typename iterator_traits<I>::difference_type n)
```

calls an overloaded function with an extra `I::iterator_category` argument:

```
advance(i,n,typename iterator_traits<I>::iterator_category())
```

which will resolve, depending on the type of

```
iterator_traits<I>::iterator_category()
```

to the most efficient implementation. E.g.

```
int*     a;
advance(a,10);
```

will eventually resolve (explain!) to

```
a += 10
```

pair

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair() : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b) : first(a), second(b) {}
};
template <class T1, class T2>
inline bool
operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {
return x.first == y.first && x.second == y.second;
}
template <class T1, class T2>
inline bool
operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {
return x.first < y.first || (!(y.first < x.first) && x.second < y.second);
}
template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y) {
return pair<T1, T2>(x, y);
}
```

example container: map

```
template <class Key, class T, class Compare=less<Key>, class Alloc = alloc> class map {
public: // a map implements a [Key->T] finite function
  typedef Key key_type;              typedef rep_type::iterator iterator;
  typedef T data_type;               typedef rep_type::const_iterator const_iterator;
  typedef pair<const Key, T> value_type; // *iterator is value_type&
  // constructors
  map();
  template <class InputIterator> map(InputIterator first, InputIterator last);// [*first,..,*last[
  // inspectors
  iterator begin();                  const_iterator begin() const;
  iterator end();                    const_iterator end() const;
  size_type size() const;
  iterator find(const key_type& x);  const_iterator find(const key_type& x) const;
  size_type count(const key_type& x);
  // mutators; insert().first = where inserted, insert().second = true iff ok
  pair<iterator,bool> insert(const value_type& x);
  T& operator[](const key_type& k); // can be assigned to for insert/replace
  template <class InputIterator> void insert(InputIterator first, InputIterator last);
  void erase(iterator position);
  size_type erase(const key_type& x); // find and erase, return 1 iff ok
  void erase(iterator first, iterator last);
  void clear(); // erase [begin(),..,end()[
};
```

map usage example

```
// $Id: mapex.C,v 1.2 2000/11/08 06:41:02 dvermeir Exp $
#include        <string>
#include        <map>
typedef map<string,int>  EXAMEN;

int
main(int,char**) {
EXAMEN scores;
// insert
scores[string("john")] = 18;
scores.insert(make_pair(string("fred"),5));
// retrieve
for (EXAMEN::const_iterator i=scores.begin();(i!=scores.end());++i)
  cout << (*i).first << ": " << (*i).second << endl;
// update
scores[string("fred")] = 11; // 2de zittijd
EXAMEN::iterator i = scores.find(string("john"));
if (i!=scores.end()) {
  (*i).second = 13; // another way to update
  cout << (*i).first << ": " << (*i).second << endl;
  }
return 0;
}
```

example container: set (specialization of map)

```
template <class Key, class Compare = less<Key>, class Alloc = alloc> class set {
public: // elements are kept in sorted order, less<Key>=Key::operator<
  typedef Key key_type;
  typedef Key value_type;
  typedef rep_type::const_iterator iterator;
  // constructors
  set();
  set(const set<Key, Compare, Alloc>& x);
  template <class InputIterator> set(InputIterator first, InputIterator last);// [*first,..,*last[
  // inspectors
  iterator begin();
  iterator end();
  size_type size(); // cardinality
  iterator find(const key_type& x); // end() if not found
  size_type count(const key_type& x); // 1 or 0
  // mutators; insert().first = where inserted, insert().second = true iff ok
  pair<iterator,bool> insert(const value_type& x);
  void erase(iterator position); // erase at position
  size_type erase(const key_type& x); // 1 if ok, 0 if not
  void erase(iterator first, iterator last); // erase [*first,..,*last[
  void clear(); // erase(begin(),end());
};
```

example algorithm: for_each, find, find_if, copy

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f) {
  for ( ; first != last; ++first)
    f(*first);
  return f;
}
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
  while (first != last && *first != value) ++first;
  return first;
}
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
template <class InputIterator, class OutputIterator>// code is a simplification of the real thing
inline OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result) {
for ( ; first != last; ++result, ++first) // how do we copy to a set container?
  *result = *first; // how do we copy to a vector that is too small?
  return result;
  }
}
```

insert_iterator

```
template <class Container>
class insert_iterator { // an iterator that translates *it = v to a container insert operation
protected: // e.g. copy(c1.begin(),c1.end(),inserter(c2,c2.begin())) will work fine
  Container* container;
  typename Container::iterator iter;
public:
  insert_iterator(Container& x, typename Container::iterator i) : container(&x), iter(i) {}
  insert_iterator<Container>& operator=(const typename Container::value_type& value) {
                iter = container->insert(iter, value);
                ++iter;
                return *this;
                }
  insert_iterator<Container>& operator*() { return *this; } // do nothing
  insert_iterator<Container>& operator++() { return *this; } // do nothing
  insert_iterator<Container>& operator++(int) { return *this; } // do nothing
};

template <class Container, class Iterator> // makes it easy to use an insert iterator
inline insert_iterator<Container> inserter(Container& x, Iterator i) {
  typedef typename Container::iterator iter;
  return insert_iterator<Container>(x, iter(i));
}
```

istream_iterator

```
template <class T, class Distance = ptrdiff_t>
class istream_iterator { // an input iterator that reads values from a stream
  friend bool
  operator== (const istream_iterator<T, Distance>& x, const istream_iterator<T, Distance>& y);
protected: // e.g. value = *it++; will read value from stream of it
  istream* stream; // stream from which data is read
  T value;
  bool can_read; // true iff not yet at end
  void read() { can_read = (*stream) ? true : false;
        if (can_read) *stream >> value; can_read = (*stream) ? true : false;
        }
public:
  typedef T                 value_type;
  typedef const T*          pointer;
  typedef const T&          reference;
  istream_iterator() : stream(&cin), can_read(false) {}
  istream_iterator(istream& s) : stream(&s) { read(); }
  reference operator*() const { return value; }
  pointer operator->() const { return &(operator*()); }
  istream_iterator<T, Distance>& operator++() { read(); return *this; }
  istream_iterator<T, Distance> operator++(int)  {
        istream_iterator<T, Distance> tmp = *this; read(); return tmp;
        }
};
```

ostream_iterator

```
template <class T>
class ostream_iterator { // an output iterator that writes to a stream
protected: // e.g. *it = value will write value on the stream of it
  ostream* stream;
  const char* string; // what is this used for?
public:
  typedef void                    value_type;
  typedef void                    difference_type;
  typedef void                    pointer;
  typedef void                    reference;

  ostream_iterator(ostream& s) : stream(&s), string(0) {}
  ostream_iterator(ostream& s, const char* c) : stream(&s), string(c)  {}
  ostream_iterator<T>& operator=(const T& value) {
    *stream << value;
    if (string) *stream << string;
    return *this;
    }
  ostream_iterator<T>& operator*() { return *this; }
  ostream_iterator<T>& operator++() { return *this; }
  ostream_iterator<T>& operator++(int) { return *this; }
};
```

sample application: telephone directory

```
1 tinf2$ tel fred "x2356 02-6124441" # insert data for fred
2 tinf2$ tel jane 092−6124441 # insert data for jane
3 tinf2$ tel fred # retrieve data for fred
fred     x2356 02−6124441
4 tinf2$ tel fred ""  # delete fred
5 tinf2$ tel fred
"fred" not found
6 tinf2$ tel # show all data in the file
jane 092−6124441
```

```
#ifndef TEL_H
#define TEL_H
// $Id: tel.h,v 1.2 1999/12/05 13:27:14 dvermeir Exp $
#include        <string>
#include        <iostream>

class TelRecord { // associates a string info_ with a string name_
public:
        TelRecord(const string& name="",const string& info=""): name_(name), info_(info) {}
        TelRecord(const TelRecord& r): name_(r.name_), info_(r.info_) {}

        // two records are the same if their name_'s match
        bool    operator==(const TelRecord& r) const { return name_ == r.name_; }
        bool    operator<(const TelRecord& r) const { return name_ < r.name_; }

        friend ostream& operator<<(ostream& os,const TelRecord& r);
        friend istream& operator>>(istream& os,TelRecord& r);
private:
        string  name_; // unique key, no two records have the same name
        string  info_; // name_ may not contain white space, info_ can
};
```

```
ostream&
operator<<(ostream& os,const TelRecord& r) {
os << r.name_ << '\t' << r.info_ << endl;
return os;
}

istream&
operator>>(istream& is,TelRecord& r) {
is >> r.name_ ; // line starts with name
is.ignore(); // ignore \t following name
getline(is,r.info_); // rest of line is info
return is;
}

#endif
```

```
// $Id: tel.C,v 1.3 2000/11/08 06:41:04 dvermeir Exp $

// usage:
//          tel name             – query: retrieve info for name from database
//          tel name info        – insert: store/replace info for name in database
//          tel name “”          – delete: remove name from database
//          tel                  – list whole database

// error return codes:
//
//          1                    – query: not found
//          2                    – insert: cannot write database file
//          3                    – delete: not found
//          4                    – too many arguments

// only 1 info string can be associated with a name in the database

#include <iostream>
#include <fstream.h>
#include <set>
#include "tel.h"
```

```
// note that to use set<T>, T needs a default and a copy-constructor as well as an operator<
typedef set<TelRecord>                  record_set;
typedef istream_iterator<TelRecord>     record_input_it;

const char*     database_name = "tel.data";

int
main (int argc, char *argv[]) {
record_set      dir;
{
// Load the database into a record_set

ifstream        data_file(database_name);

if (data_file)
   copy(record_input_it(data_file),record_input_it(),insert_iterator<record_set>(dir,dir.begin()));
// data_files is automatically closed by the ifstream destructor
}
```

```
switch (argc) {
  case 1: // no arguments: just show all records on cout
    copy(dir.begin(),dir.end(),ostream_iterator<TelRecord>(cout));
    return 0;
  case 2: // 1 argument: retrieve info associated with argv[1]
    { // use the find() algo with dummy TelRecord with name_= key
    string          key(argv[1]);
    record_set::iterator              r = dir.find(TelRecord(key,""));

    if (r!=dir.end()) { // success: iterator points to found record
      cout << *r; // output it
      return 0;
      }
    else {
      cerr << "\"" <<  key << "\" not found" << endl;
      return 1;
      }
    }
    break;
```

```
case 3: { // 2 arguments, insert, replace or delete
  string       key(argv[1]);
  string       info(argv[2]);
  TelRecord    r(key,info);

  if (info.size()==0) { // delete
    if (dir.erase(r)!=1) {
      // return value of set::erase() is number of els deleted
      cerr << "\"" << key << "\" not found; cannot erase" << endl;
      return 3;
      }
    }
  else { // record_set::insert(r) returns a pair of an iterator and a bool
    pair<record_set::iterator,bool>     pair = dir.insert(r);
    if (!pair.second) { // insert failed: duplicate key: replace
      dir.erase(pair.first); // erase,
      dir.insert(r); // then insert again
      }
    }
```

```
      // save file
      ofstream        data_file(database_name);
      if (!data_file) {
        cerr << "Cannot open \"" << database_name << "\" for writing" << endl
        return 2;
        }
      copy(dir.begin(),dir.end(),ostream_iterator<TelRecord>(data_file));
      return 0;
      }
      break;
    default:
      cerr << "Usage: " << argv[0] << " [name [info|""]] " << endl;
      return 4;
      break;
    }

return 0;
}
```

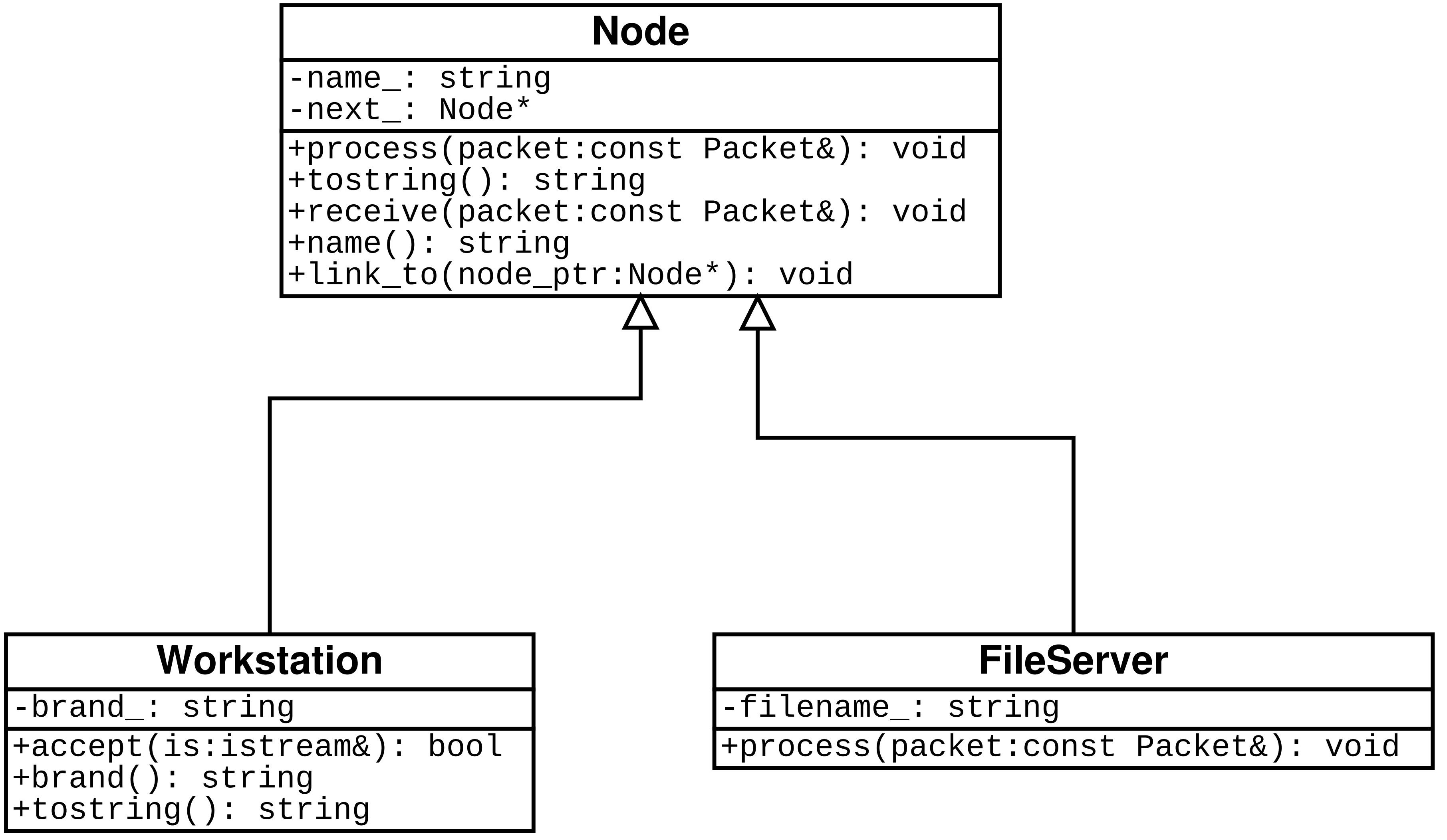
problem description

A circular LAN consists of **nodes**. Nodes process **packets** that are addressed to them; and may pass other packets on to the next node. Besides "simple nodes" there are several more sophisticated types of nodes: e.g. **workstations** may generate new packets, **fileservers** save their packets in a file.

Thus a workstation **is-a** node and a fileserver **is-a** node as well. In OO jargon, we say that the class **Workstation** and the class **Fileserver** are **subclasses** of the class **Node**: they **inherit** (data and function) members from the class Node. In C++ we say that **Workstation** and **Fileserver** are **derived from Node**.

```
class Node;
class Workstation: public class Node; // Workstation is derived from Node
class Fileserver: public class Node; // Fileserver is derived from Node
```

uml class diagram



class Packet

```
#ifndef PACKET_H
#define PACKET_H
#include <string>
#include <iostream>
class Packet { // "bare bones" implementation
public:
  Packet(const string& destination,const string& contents):
        destination_(destination), contents_(contents) {}
  string    destination() const { return destination_; }
  string    contents() const { return contents_; }
private:
  string    destination_;
  string    contents_;
};

inline ostream&
operator<<(ostream& os,const Packet& p) {
return os << "[" << p.destination() << ", \"" << p.contents() << "\"]";
}
#endif
```

class Node

```
#ifndef NODE_H
#define NODE_H

#include <string>
#include "packet.h"

class Node {
public:
  Node(const string& name,Node* next=0): name_(name), next_(next) {}
  void receive(const Packet&); // what we do with a packet we receive
  void process(const Packet&); // what we do with a packet destined for us
  void link_to(Node* node_ptr) { next_ = node_ptr; }
  string         name() const { return name_; }
  string         tostring() const; // nice printable name for Node
private:
  string         name_; // unique name of Node
  Node*          next_; // in LAN
};
#endif
```

class Node

```
#include "node.h"

void
Node::receive(const Packet& packet) {
cout << tostring() << " receives packet " << packet << endl;
if (packet.destination()==name())
  process(packet);
else
  if (next_)
    next_->receive(packet);
}

void
Node::process(const Packet& packet) {
cout << tostring() <<  " processes " << packet << endl;
}

string
Node::tostring() const {
return string("node ")+name_;
}
```

class Workstation

```
#ifndef WORKSTATION_H
#define WORKSTATION_H

#include        <iostream>
#include        "node.h"

class Workstation: public Node { // Workstation is publicly derived from Node
public:
  Workstation(const string name,const string brand, Node* next=0):
                Node(name,next), brand_(brand) {}

  bool          accept(istream& is); // extra: accept packet from cin and send it to next
  string        tostring() const; // override Node::tostring()
  string        brand() const { return brand_; } // extra function
private:
  string        brand_; // extra data member
};
#endif
```

class Workstation

```
#include "workstation.h"

bool
Workstation::accept(istream& is) { // packet format: destination contents \n
string destination;
string contents;
cout << tostring() << " accepts a packet.." << endl;
is >> destination;
getline(is,contents);
if (destination.size()==0) {
  cerr << "error: name of destination not specified" << endl;
  return false;
  }
Packet packet(destination,contents);
receive(packet); // ``send'' the packet to myself
return true;
}

string
Workstation::tostring() const {
return Node::tostring()+" ("+brand()+")";
}
```

program lan

```
#include        "node.h"
#include        "workstation.h"

int
main(int,char**)  {
Workstation     wendy(string("wendy"),string("Sun ultra"));
Node            node1(string("fred"),&wendy); // node1 -> wendy
Node            node2(string("jane"),&node1); // node2 -> node1
wendy.link_to(&node2); // wendy -> node2
// now we have a circular net: wendy -> node1 -> node1 -> wendy
while (wendy.accept(cin))
  ;
}
```

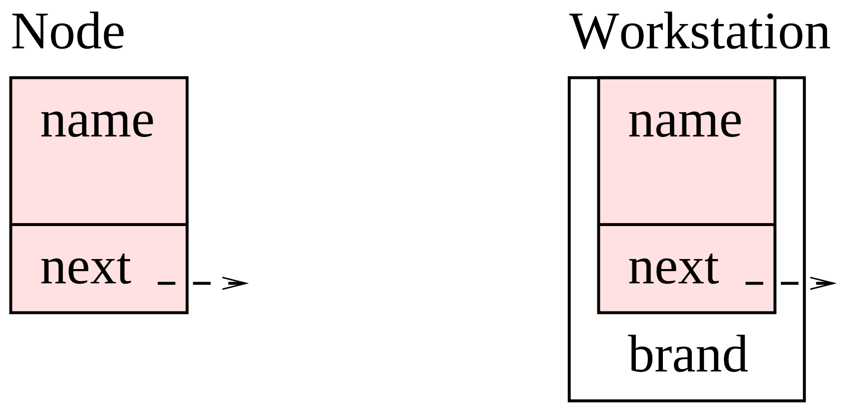
running lan

```
3039 dv2:~/courses/structuur2/slides$ lan
node wendy (Sun ultra) accepts a package..
fred hi there, fred
node wendy receives packet [fred, " hi there, fred"]
node jane receives packet [fred, " hi there, fred"]
node fred receives packet [fred, " hi there, fred"]
node fred processes [fred, " hi there, fred"]
node wendy (Sun ultra) accepts a package..
3040 dv2:~/courses/structuur2/slides$
```

Explain these lines:

> node wendy (Sun ultra) accepts a package..
> node wendy receives packet [fred, ” hi there, fred”]

derived object layout



- A **Workstation** contains a "**Node** part" which is initialized using a Node constructor.
- The address of a **Workstation** is also the address of a **Node**.
- The compiler can convert automatically

Workstation*	$\rightarrow$	Node*
Workstation&	$\rightarrow$	Node&
Workstation	$\rightarrow$	Node *how?*

virtual member functions

```
Workstation*      ws;
Node*             nodeptr(ws); // ok

nodeptr->tostring(); // we would like this to call Workstation::tostring()
```

This can be achieved by declaring

```
virtual string Node::tostring() const;
```

which will cause the compiler to generate code such that which function is actually called for

```
nodeptr->tostring()
```

is determined at run time (**late binding**) and depends on the “**real**” class of **`*nodeptr`**.

class Node with virtual function members

```
#ifndef NODE2_H
#define NODE2_H

#include        <string>
#include        "packet.h"

class Node { // version with virtual functions
public:
  Node(const string& name,Node* next=0): name_(name), next_(next) {}
  virtual ~Node() {} // virtual destructor, see later
  void            receive(const Packet&); // what we do with a packet we receive
  virtual void    process(const Packet&); // what we do with a packet destined for us
  void            link_to(Node* node_ptr) { next_ = node_ptr; }
  string          name() const { return name_; }
  virtual string tostring() const; // nice printable name for Node
private:
  string          name_; // unique name of Node
  Node* next_; // in LAN
};
#endif
```

class Node with virtual function members

```
#include "node2.h"

void
Node::receive(const Packet& packet) {
cout << tostring() << " receives packet " << packet << endl;
if (packet.destination()==name())
  process(packet);
else
  if (next_)
    next_->receive(packet);
}

void
Node::process(const Packet& packet) {
cout << tostring() <<  " processes " << packet << endl;
}

string
Node::tostring() const {
return string("node ")+name_;
}
```

class Workstation with virtual function members

```
#ifndef WORKSTATION2_H
#define WORKSTATION2_H

#include <iostream>
#include "node2.h"

class Workstation: public Node { // Workstation is publicly derived from Node
public:
  Workstation(const string name,const string brand, Node* next=0):
        Node(name,next), brand_(brand) {}
  virtual ~Workstation() {} // virtual destructor, see later

  bool        accept(istream& is); // extra: accept packet from cin and send it on
  string      tostring() const; // override virtual Node::tostring()
  string      brand() const { return brand_; } // extra function
private:
  string      brand_; // extra data member
};
#endif
```

class Workstation with virtual function members

```
#include        "workstation2.h"

bool
Workstation::accept(istream& is) {
string     destination;
string     contents;
cout << tostring() << " accepts a package.." << endl;
is >> destination;
getline(is,contents);
if (destination.size()==0) {
  cerr << "error: name of destination not specified" << endl;
  return false;
  }
Packet    packet(destination,contents);
receive(packet);
return true;
}

string
Workstation::tostring() const {
return Node::tostring()+" ("+brand()+")";
}
```

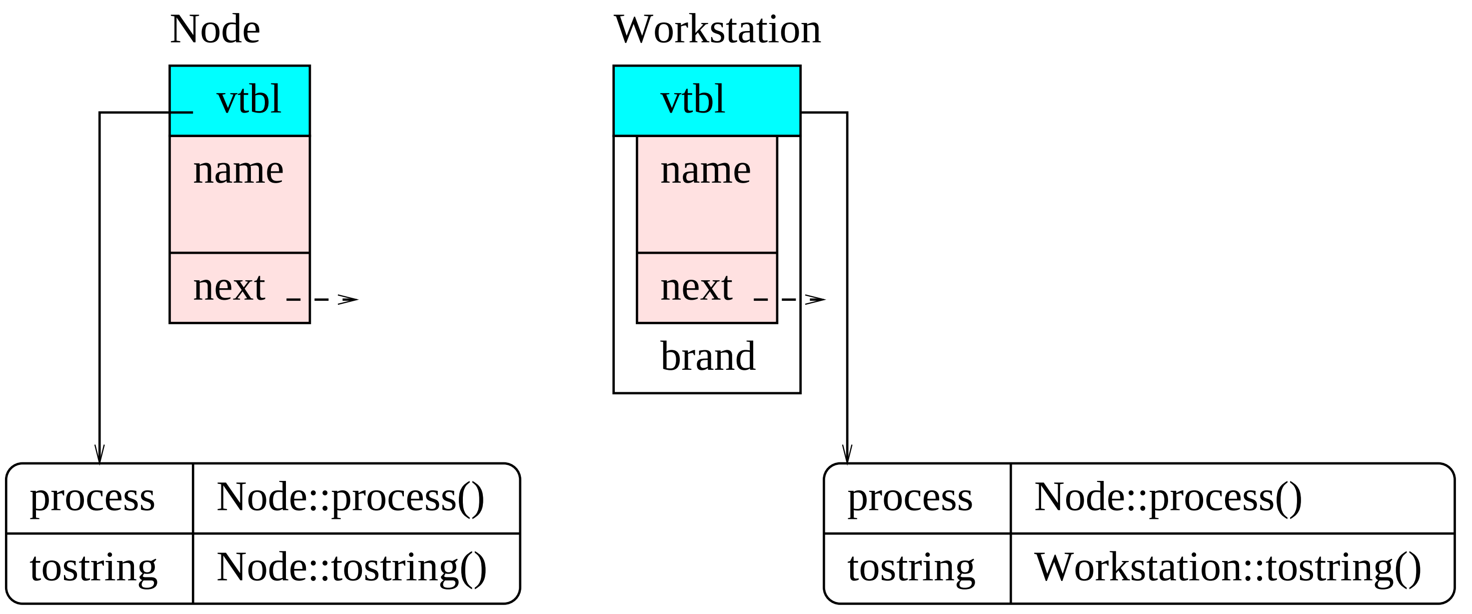
running lan2

```
3089 dv2:~/courses/structuur2/slides$ lan2
node wendy (Sun ultra) accepts a package..
fred hi there,fred
node wendy (Sun ultra) receives packet [fred, " hi there,fred"]
node jane receives packet [fred, " hi there,fred"]
node fred receives packet [fred, " hi there,fred"]
node fred processes [fred, " hi there,fred"]
node wendy (Sun ultra) accepts a package..
3090 dv2:~/courses/structuur2/slides$
```

Explain this line:

node wendy (Sun ultra) receives packet [fred, " hi there,fred"]

derived object layout with virtual functions: vtbl's



Node∗ nodeptr;

nodeptr−>tostring()

is translated to

nodeptr−>vtbl[1]()

virtual functions without refs or pointers

```
Workstation     ws("wendy","Sun");
Node            node(ws);

node.tostring();
```

Which function is executed? why?

pure virtual functions

```
class Animal {
public:
        virtual string sound() const = 0;
};

class Dog: public Animal {
        string sound() const { return "woef"; }
};
class Cat: public Animal {
        string sound() const { return "miauw"; }
};

set<Animal*>   pets;
for (set<Animal*>::const_iterator a=pets.begin();(a!=pets.end());++a)
        cout << (*a)->sound() << endl;
Animal  fido; // error, why?
```

Abstract classes

Abstract classes are **specifications**; they define an **interface** that subclasses will have to implement.

```
#ifndef STACK_H
#define STACK_H
template<class T>
class Stack { // an abstract class, aka interface in java
public:
  virtual T                  pop() = 0;
  virtual void               push(const T&) = 0;
  virtual unsigned int       size() const = 0;
  void                       clear() {  // clear() only uses pure virtual functions
                               while (size()>0)
                                 pop();
                             }
  virtual                    ~Stack() {} // see further
};
#endif
```

Implementing abstract classes

```
#ifndef LISTACK_H
#define LISTACK_H
#include        "stack.h"
#include        <list>
// We implement a stack using a list; the top of the stack
// is represented by the front of the list.
template <class T>
class ListStack: public Stack<T> {
private:
  list<T>        list_;
public:
  ListStack(): list_() {}

  T              pop() { T t = list_.front(); list_.pop_front(); return t; }
  void           push(const T& t) { list_.push_front(t); }
  unsigned int   size() const { return list_.size(); }
};
#endif
```

Using liststack1

```
#include "liststack.h"

int
main(int,char**) {
Stack<int>* s = new ListStack<int>();
for (int i=0;(i<10);++i)
  s->push(i);
while (s->size()>0)
  cout << s->pop() << endl;
}
```

virtual destructors

```
class Person {
public:
  Person(const string& name): name_(name) {}
  string        name() const { return name_; }
  void          set_name(const string& name) { name_ = name; }
private:
  string        name_;
};
class Image { // something *BIG*
};
class FamilyMember: public Person {
public:
  FamilyMember(const string& name,Image* im=0): Person(name), image_(im) {}
  ~FamilyMember() { if (image_) delete image_; }
  Image*        image() const { return image_; }
  void          set_image(Image* im) { image_ = im; }
private:
  Image*        image_;
};
```

virtual destructors

```
#include "person.h"

Person* p(new FamilyMember("fred",im));
delete p; // which destructor will be called? why?
```

To fix:

```
class Person {
public:
  Person(const string& name): name_(name) {}
  virtual ~Person() {} // how will this fix the problem?
  string        name() const { return name_; }
  void          set_name(const string& name) { name_ = name; }
private:
  string        name_;
};
```

private derivation and multiple inheritance

```
#ifndef LISTACK_H
#define LISTACK_H
#include        "stack.h"
#include        <list>
// inheriting privately from list<T> ensures that users of ListStack
// cannot access the underlying list<T> operations; this makes
// ListStack a properly encapsultaed Abstract Data Type
template <class T>
class ListStack: public Stack<T>, private list<T> {
public:
  ListStack(): list<T>() {}

  T               pop() { T t = front(); pop_front(); return t; }
  void            push(const T& t) { push_front(t); }
  unsigned int    size() const { return list<T>::size(); } // what's this?
};
#endif
```

- Inherit publicly from the abstract base class (**interface**)
- Inherit privately for the **implementation**.

multiple and virtual inheritance

```
#include <string>

struct Person {
string name;
};

struct Staff: public virtual Person {
int salary;
};

struct Student: public virtual Person {
int rolnr;
};

struct Tutor: public Student, public Staff {
}; // how many name's does a Tutor have?
```

A derived class contains only **a single copy** of any **virtual base class**.

iostreams: protected members

```
class streambuf; // manages buffer and communication with device
class ios {
// format state, status info, streambuf etc.
protected: // only derived classes can create an ios
  ios(streambuf*);
};
class istream: virtual public ios {
public:
  istream(streambuf* buf): ios(buf) { /* .. */ }
  int read(char* buffer,unsigned int size);
};
class ostream: virtual public ios {
public:
  ostream(streambuf* buf): ios(buf) { /* .. */ }
  int write(char* buffer,unsigned int size);
};
class iostream: public istream, public ostream {
public:
  iostream(streambuf* buf): istream(buf), ostream(buf), ios(buf) { /* .. */ }
  // why explicit call to ios(buf)?
};
class fstream: public iostream { // a stream connected to a file
// ...
};
```

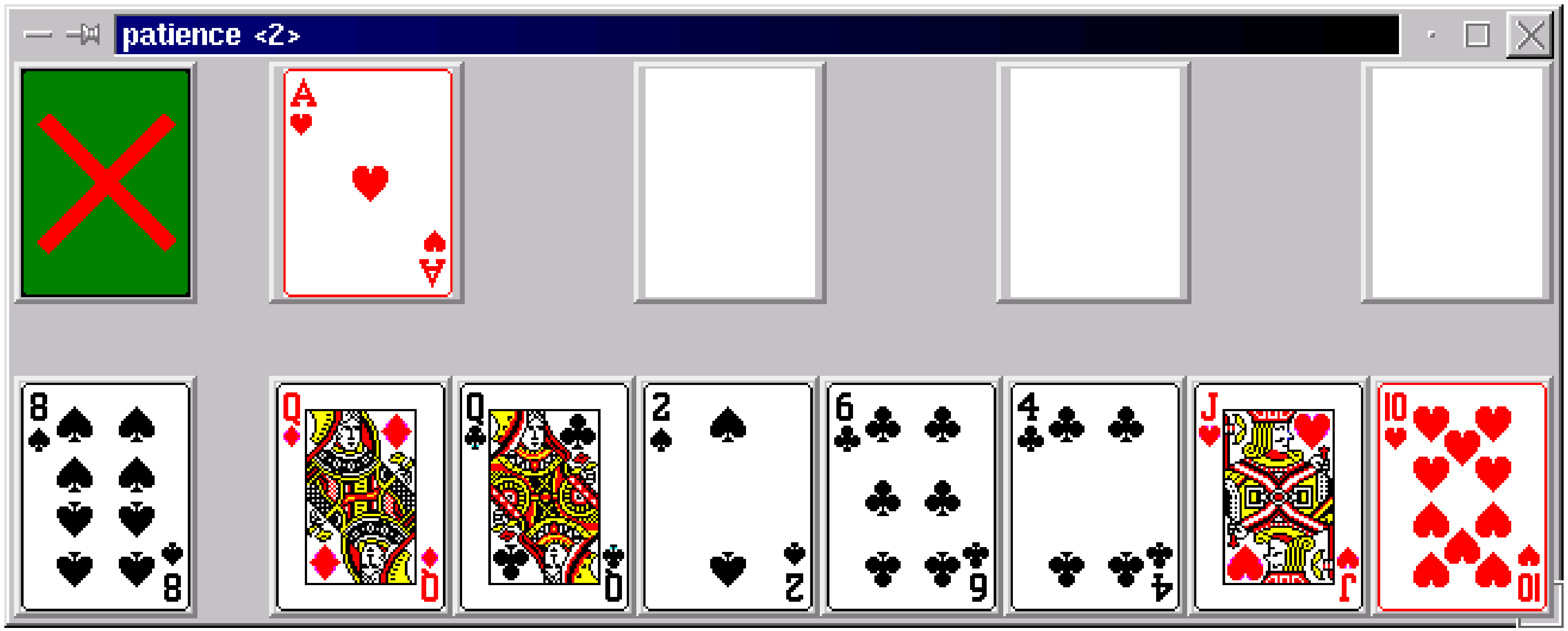
inheritance and operators

- All operators except (*why?*) constructors, destructors and assignment are inherited.
- Be careful with inherited **new** and **delete** operators: use **size_t** argument.
- What is the order of initialization for an object of a derived class?
- What if a derived class's constructor does not mention a base class constructor?
- What is the order of finalization for an object of a derived class?

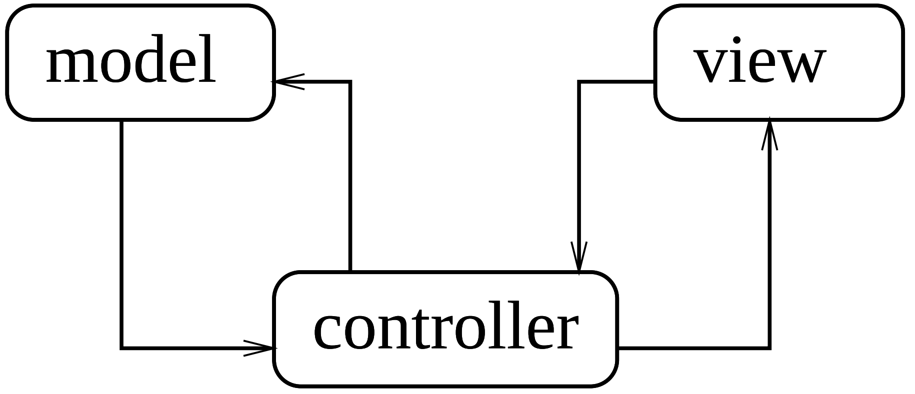
inheritance and arrays

```
struct Base { // 4 bytes
  Base(int i=0): i(i) {}
  int i;
};

struct Derived: public Base { // 8 bytes
  Derived(int i=0,int j=0): Base(i), j(j) {}
  int j;
};

void
f(Base a[]) { cout << a[0].i << ", " << a[1].i << endl; }

int
main() {
Derived da[] = { Derived(1,2), Derived(3,4) };
f(da); // prints 1,2 instead of the expected 1,3
}
```

patience snapshot

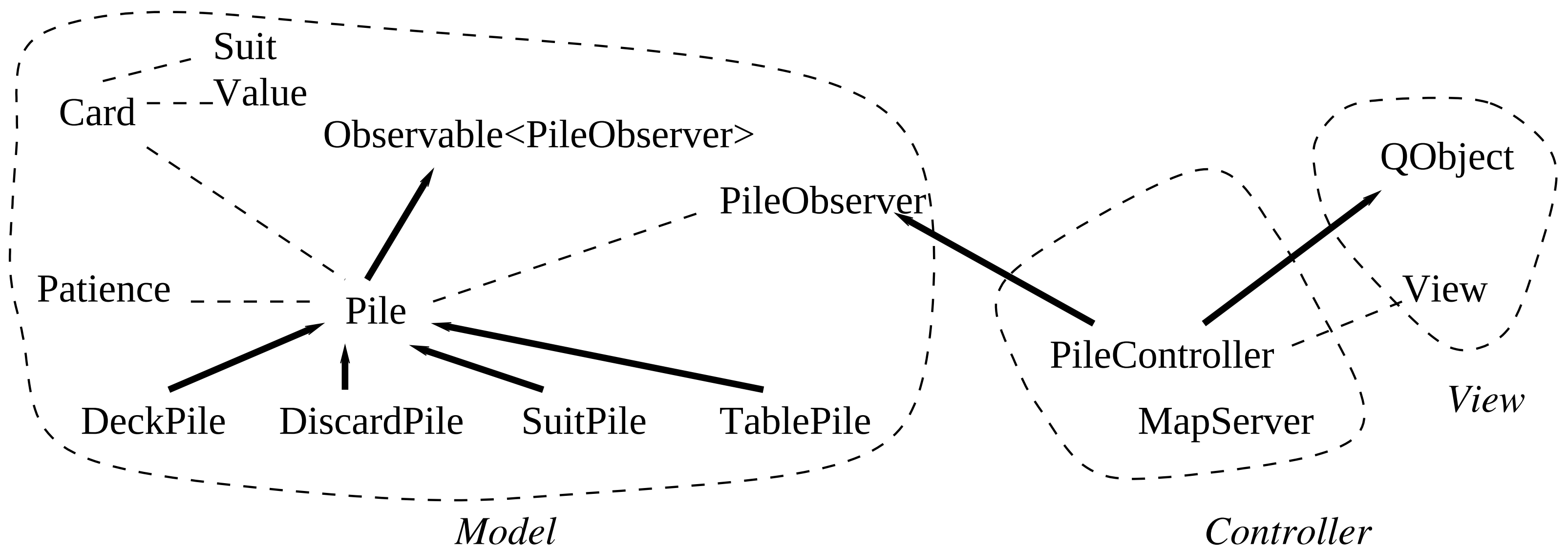


architecture



- **model**: Card, Pile etc., Patience
- **view**: View, (QPushButton, QHBox, QVBox, ..)
- **controller**: PileController, MapServer

classes



- Pile:
  - `void select()`: What pile does when activated, may try to transfer card(s) to connected piles.
  - `bool accept(Card)`: May accept card (from another pile).
- specialized Piles: DeckPile, DiscardPile, TablePile, SuitPile

card suits

```
#ifndef CARD_H
#define CARD_H
// $Id: card.h,v 1.3 2000/12/20 10:44:09 dvermeir Exp $
#include        <vector>
#include        <string>
#include        <assert.h>

class Suit {
public:
  static const unsigned int N_SUITS = 4;
  enum { HEART, DIAMOND, CLUB, SPADE };
  enum Color { RED, BLACK };

  Suit(unsigned int i): suit_(i) { assert(i<N_SUITS); }

  string        str() const;
  Color         color() const;
  /// the following operator takes care of == etc.
  operator unsigned int() const { return suit_; }
private:
  unsigned int  suit_;
};
```

card values

```
class Value {
public:
  static const unsigned int N_VALUES = 13;
  enum { ACE=1, JACK=11, QUEEN=12, KING=13 };

  Value(unsigned int i): value_(i) { assert(i>0 && i<= N_VALUES); }

  string        str() const;
  /// the following operator takes care of == etc.
  operator unsigned int() const { return value_; }
private:
  unsigned int  value_;
};
```

cards and decks

```
class Card {
public:
  Card(Suit suit,Value value): suit_(suit), value_(value) {}

  Suit          suit() const { return suit_; }
  Value         value() const { return value_; }
  string        str() const;

  bool          operator<(const Card& c) const { // e.g. for map<Card,T>
                  return suit() < c.suit() ||
                         ( suit()==c.suit() && value() < c.value() );
                }

  Suit::Color   color() const { return suit().color(); }
  // a Deck is just a vector containing all cards
  typedef vector<Card>  Deck;
  static Deck           deck(bool shuffled=false);
private:
  Suit   suit_;
  Value  value_;
};
#endif
```

observable and observers

```
#ifndef OBSERVER_H
#define OBSERVER_H
// $Id: observer.h,v 1.3 2000/12/20 10:44:10 dvermeir Exp $
#include <list>
#include <functional>
#include <algorithm>

// An Observable has a number of associated Observers that will be
// warned using Observer::notify() each time the Observable changes state
template<class Observer>
class Observable {
public:
  void add_observer(Observer& observer) { observers_.push_back(&observer); }
  void remove_observer(Observer& observer) { observers_.remove(&observer); }
protected: // an observable that changes state will call notify()
  void notify() {
    for_each(observers_.begin(), observers_.end(), mem_fun(&Observer::notify));
    }
private:
  list<Observer*>        observers_;
};
#endif
```

diversion: mem_fun_t

```
// Example use:
//
//      class C {
//              int f();
//              }
//      C*      p;
//      mem_fun_t<C,int>        F(C::f);
//      F(p) will call p->f()
//
// Useful, e.g. to use a member function in an algorithm like for_each
template <class S, class T>
class mem_fun_t : public unary_function<T*, S> {
public: // explicit: the constructor cannot be used for automatic conversion
  explicit mem_fun_t(S (T::*pf)()) : f(pf) {}
  S operator()(T* p) const { return (p->*f)(); }
private:
  S (T::*f)();
};
template <class S, class T> // convenience function to make mem_fun_t
inline mem_fun_t<S,T> mem_fun(S (T::*f)()) { return mem_fun_t<S,T>(f); }
```

card pile

```
class PileObserver;
class Pile: public Observable<PileObserver> {
public:
  Pile(bool visible=true): pile_(), visible_(visible) {}
  virtual ~Pile() {} // possibly needed for subclasses
  // inspectors
  unsigned int   size() const { return pile_.size(); }
  bool           empty() const { return size()==0; }
  bool           visible() const { return visible_; }
  Card           top() const;
  // mutators, will call notify() to warn any observers
  Card           pop();
  void           add(Card card);
  // behaviour of pile:
  // selected() reacts to the user clicking on the pile, e.g.
  //     the pile may try to move its top card somewhere else
  // accept() checks whether a card can be put on top of this pile
  //     and does so, if possible
  virtual void   selected() {}
  virtual bool   accept(Card c) { return false; }
private:
  vector<Card>   pile_;
  bool           visible_;
};
```

card pile observer

```
class PileObserver { // this class connects the game with the controller
public: // a particular GUI will subclass PileObserver
  PileObserver(Pile& pile): pile_(pile) { pile_.add_observer(*this); }
  virtual ~PileObserver() { pile_.remove_observer(*this); }
  void              notify() { changed(); }
  // inspector
  Pile&             pile() { return pile_; }
protected: // for a given GUI, changed() will generate the visual response
  virtual void      changed() = 0;
private:
  Pile&             pile_;
};
#endif
```

tablepile

```
#ifndef TABLEPILE_H
#define TABLEPILE_H
// $Id: tablepile.h,v 1.4 2000/12/20 10:44:10 dvermeir Exp $

#include        "pile.h"
#include        <vector>

class TablePile: public Pile {
public:
  TablePile(vector<Pile*>& receivers);
  void          selected();
  bool          accept(Card c);
private:
  vector<Pile*>&        receivers_;
};
#endif
```

```
// $Id: tablepile.C,v 1.5 2000/12/20 10:44:10 dvermeir Exp $
#include    "tablepile.h"
/////////////// Constructor

TablePile::TablePile(vector<Pile*>& receivers):
         Pile(), receivers_(receivers) {
}
/////////////// behaviour
void
TablePile::selected() {
if (empty())
  return;
Card  c(pop());
for (vector<Pile*>::iterator i=receivers_.begin();(i!=receivers_.end());++i)
  if ((*i)!=this)
    if ((*i)->accept(c))
      return;
add(c);
}
```

```
bool
TablePile::accept(Card c) {
if (empty())
   if (c.value()==Value::KING) {
      add(c); return true;
      }
   else
      return false;
else
   if (c.color()!=top().color())
      if (c.value()==top().value()−1) {
         add(c); return true;
         }
      else
         return false;
   else
      return false;
}
```

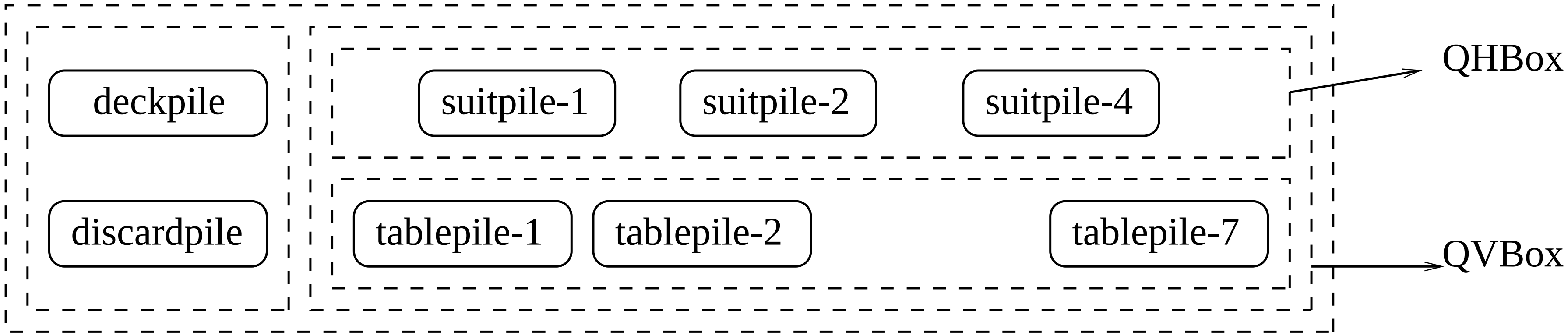
game class

```
class Patience {
public:
  static const unsigned int      N_TABLES = 7;

  Patience();
  ~Patience();

  void         deal(); // distributes cards over deck and table piles
  // inspectors
  Pile&        deck() { return deck_; }
  Pile&        discard() { return discard_; }
  Pile&        suitpile(unsigned int i) { assert(i<Suit::N_SUITS);
                       return suitpile_[i]; }
  Pile&        tablepile(unsigned int i) { assert(i<N_TABLES);
                       return *tablepile_[i]; }
private:
  SuitPile     suitpile_[Suit::N_SUITS];
  TablePile*   tablepile_[N_TABLES];
  DiscardPile  discard_;
  DeckPile     deck_;
  vector<Pile*> receivers_; // all but deck
};
```

```
//////////////// Constructor

Patience::Patience(): discard_(receivers_), deck_(discard_) {
for (unsigned int i=0;(i<N_TABLES);++i)
        tablepile_[i] = new TablePile(receivers_);
// set up empty piles
for (unsigned int i=0;(i<Suit::N_SUITS);++i)
  receivers_.push_back(suitpile_+i);
for (unsigned int i=0;(i<N_TABLES);++i)
  receivers_.push_back(tablepile_[i]);
}

Patience::~Patience() {
for (unsigned int i=0;(i<N_TABLES);++i)
  delete tablepile_[i];
}
```

```
//////////////// deal
void
Patience::deal() {
// deal
Card::Deck                      cards(Card::deck(true));
Card::Deck::const_iterator      ic(cards.begin());
// put some cards int the table piles
for (unsigned int i=0;(i<N_TABLES);++i)
  for (unsigned int j=i;(j<N_TABLES);++j)
    tablepile_[j]->add(*ic++);
while (ic!=cards.end())
  deck_.add(*ic++);
}
```

View classes



- Each pile is represented by a QPushButton with a QPixmap representing the top card (if visible).
- The buttons are organized into horizontal or vertical boxes.

view

```
#ifndef VIEW_H
#define VIEW_H
// $Id: view.h,v 1.5 2000/12/20 10:44:10 dvermeir Exp $
#include        <qhbox.h>
#include        <qvbox.h>
#include        <qpushbutton.h>
#include        <qpixmap.h>
#include        <assert.h>

// This class defines the game's GUI; it consists of a number of
// buttons, one for each pile of card in the game, that have a
// variable (card) image on top. The buttons are organized in horizontal
// and vertical boxes ('S' = suit, 'T' = table):
//
//      Deck    S1 S2 S3 S4
//      Discard T1 T2 T3 T4 T5 T6 T7
//
// There are 5 boxes grouping suit piles, table piles, etc.
```

```
class View {
public:
  View(unsigned int table_size,unsigned int suits_size,QPixmap& default_pixmap);
  ~View();
  /// inspectors
  QWidget&       top() { return *top_; }
  QPushButton&   deck() { return *deck_; }
  QPushButton&   discard() { return *discard_; }
  QPushButton&   table(unsigned int i) { assert(i<table_size_); return *table_[i]; }
  QPushButton&   suits(unsigned int i) { assert(i<suits_size_); return *suits_[i]; }
private:
  QPushButton*   deck_;
  QPushButton*   discard_;
  QPushButton**  table_;          // array of pointers to QPushButtons
  QPushButton**  suits_;          // array of pointers to QPushButtons
  QHBox*         table_box_;      // table piles
  QHBox*         suits_box_;      // suit piles
  QVBox*         left_;           // deck- and discard piles
  QVBox*         right_;          // table_box_ and suits_box
  QHBox*         top_;            // left_ and right_ box

  unsigned int   table_size_;     // number of table piles
  unsigned int   suits_size_;     // number of suit piles
};
#endif
```

```
// $Id: view.C,v 1.5 2000/12/20 10:44:10 dvermeir Exp $
#include        "view.h"

//////////////// constructor
View::View(unsigned int table_size,unsigned int suits_size,
        QPixmap& default_pixmap): table_size_(table_size), suits_size_(suits_size) {
top_            = new QHBox();
left_           = new QVBox(top_);
right_          = new QVBox(top_);
suits_box_      = new QHBox(right_);
table_box_      = new QHBox(right_);
deck_           = new QPushButton(left_);
deck_->setPixmap(default_pixmap);
discard_        = new QPushButton(left_);
discard_->setPixmap(default_pixmap);
```

```
suits_     = new   QPushButton*[suits_size];
for (unsigned int i=0;(i<suits_size);++i) {
   suits_[i] = new QPushButton(suits_box_);
   suits_[i]->setPixmap(default_pixmap);
   }

table_     = new QPushButton*[table_size];
for (unsigned int i=0;(i<table_size);++i) {
   table_[i] = new QPushButton(table_box_);
   table_[i]->setPixmap(default_pixmap);
   }

top_->setSpacing(30);
left_->setSpacing(30);
right_->setSpacing(30);
suits_box_->setSpacing(
   ((table_size-suits_size)*default_pixmap.width())/(suits_size-1));
}
```

PileController

```
#ifndef CONTROLLER_H
#define CONTROLLER_H
// $Id: controller.h,v 1.6 2000/12/20 10:44:09 dvermeir Exp $
#include        <qobject.h>
#include        <qpixmap.h>
#include        <qpushbutton.h>
#include        "pile.h"
// A PileController acts as a bridge between the game and its GUI: it receives
// signals from clicked cards and passes them on to the game; conversely, if a
// pile changes, it calls changed(). It also serves up QPixmap's.
class PileController: public QObject, public PileObserver {
Q_OBJECT // QT macro, needed because this class defines a ‘‘slot’’
public: PileController(Pile& pile,QPushButton& button);
public slots: // clicked() is called when button_ is pressed
  void          clicked();
protected: // changed() is called when a pile changes state
  void          changed();
private:
  QPushButton&  button_;
};
#endif
```

```
// $Id: controller.C,v 1.6 2000/12/20 10:44:09 dvermeir Exp $
#include        "controller.h"
#include        "mapserver.h"
#include        <map>
//////////////// constructor
PileController::PileController(Pile& pile,QPushButton& button):
                QObject(), PileObserver(pile), button_(button) {
// the connection from the pile to the controller was established
//       by the PileObserver constructor
// the following establishes the connection from the QPushButton
//       representing the pile to the controller: clicking the button_
//       will activate PileController::clicked()
QObject::connect(&button_,SIGNAL(clicked()),this,SLOT(clicked()));
}
```

```
//////////////// notification function called from Pile
void
PileController::changed() {
if (pile().empty())
  button_.setPixmap(PixMapServer::clear_pixmap());
else
  if (pile().visible())
    button_.setPixmap(PixMapServer::pixmap(pile().top()));
  else
    button_.setPixmap(PixMapServer::back_pixmap());
}
//////////////// slot function called from QPushButton's signal
void
PileController::clicked() {
pile().selected();
}
```

MapServer

```
#ifndef MAPSERVER_H
#define MAPSERVER_H
// $Id: mapserver.h,v 1.2 2000/12/20 10:44:10 dvermeir Exp $
#include <qpixmap.h>
#include "card.h"
// a simple class that only has static methods to return
// a pixmap, possibly associated with a Card
class PixMapServer {
public:
  static QPixmap&      pixmap(Card card);
  static QPixmap&      back_pixmap();
  static QPixmap&      clear_pixmap();
};
#endif
```

```
// $Id: mapserver.C,v 1.2 2000/12/20 10:44:10 dvermeir Exp $
#include "mapserver.h"
#include <map>
/////////////// static function associating QPixmap with card
QPixmap&
PixMapServer::pixmap(Card card) {
static map<Card,QPixmap*> maps;
if (maps.size()==0) {
  Card::Deck deck(Card::deck());
  for (Card::Deck::const_iterator i=deck.begin();(i!=deck.end());++i) {
    string fn((*i).str()+".bmp");
    maps[*i] = new QPixmap(fn.c_str());
    if (maps[*i]->isNull()) {
      cerr << "Cannot load bitmap " << fn << endl;
      exit(1);
      }
    }
  }
return *maps[card];
}
```

main

```
// $Id: main.C,v 1.5 2000/12/20 10:44:10 dvermeir Exp $
#include        <qapplication.h>
#include        "view.h"
#include        "controller.h"
#include        "mapserver.h"
#include        "patience.h"
int
main(int argc, char* argv[]) {
QApplication    application(argc,argv);
Patience        game;
View            view(Patience::N_TABLES,Suit::N_SUITS, PixMapServer::clear_pixmap());
application.setMainWidget(&view.top());
PileController  deck(game.deck(),view.deck());
PileController  discard(game.discard(),view.discard());
for (unsigned int i=0;(i<Suit::N_SUITS);++i)
  new PileController(game.suitpile(i),view.suits(i));
for (unsigned int i=0;(i<Patience::N_TABLES);++i)
  new PileController(game.tablepile(i),view.table(i));
game.deal();
view.top().show();
return application.exec();
}
```