Structuur Van Computerprogramma's II

D. Vermeir

2008-2009

Outline

Goals

- Study language closer to hardware and operating system (than Scheme): e.g. memory management.
- Introduce different programming paradigms:
  - Functions.
  - Object oriented (ADT, inheritance)
  - Generic (templates, STL).
- Learn an industrially relevant language (`Java == ( C++ )--`).

Language Popularity

Mention in on-line job offers (US, june 2003)

C++	55.5%	C	43.0%
Java	42.3%	C/C++	31.2%
VBasic	17.5%	.Net	15.2%
J*script	9.6%	Ada	7.1%
C#	5.8%	Cobol	2.6%
Fortran	1.1%	Rpg	1.2%
Pascal	0.2%		

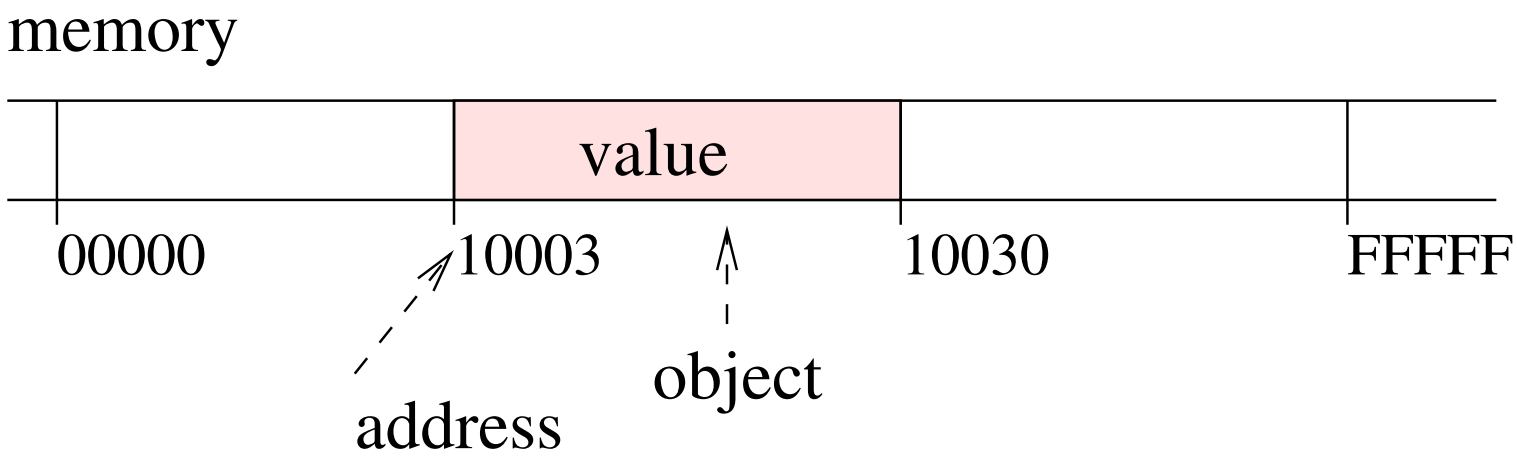
T. Plum, Quantifying Popular Programming Languages, DDJ Oct 2003.

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. Introduction to program design.

Outline

objects, values, types



- An **object** is a memory area (with an **address**), its contents is a **value**.
- An object has a **type** which implies a **size** (e.g. **int**)
- A type supports a set of **operations** (e.g. **+**) $\Rightarrow$ ADT
- To create an object of a certain type, you need to:
  1. Allocate a piece of memory of the correct size.
  2. Fill memory with initial value.

variables, constants, expressions

Ways to **refer** to a value:

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

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

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

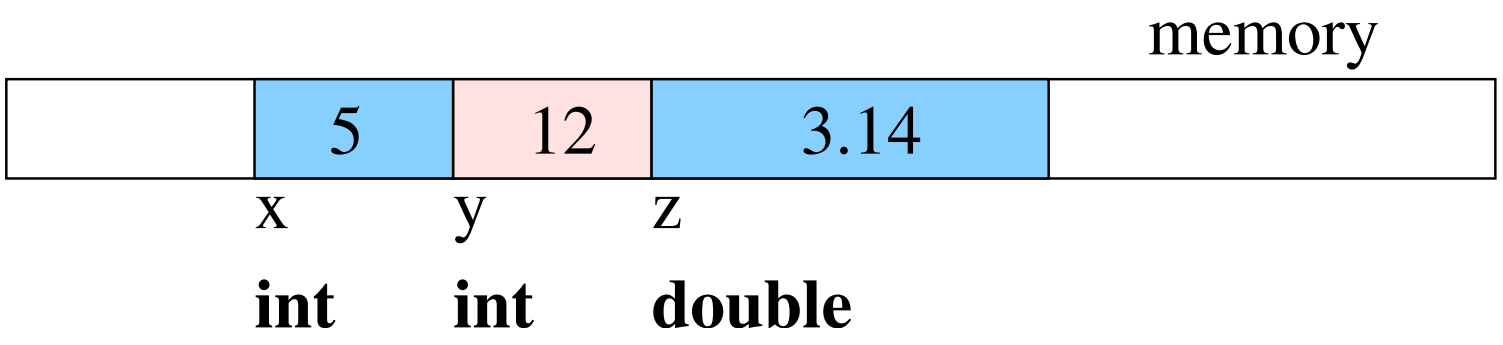
A value with an address is an lvalue, otherwise it is an **rvalue**.

x	lvalue
123	rvalue
x+123	rvalue

defining objects

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

```
1   int   x(5);
2   int   y(x+7); // initial value is specified by expression
3   double   z(3.14);
```



Alternative syntax: (see website for subtle difference):

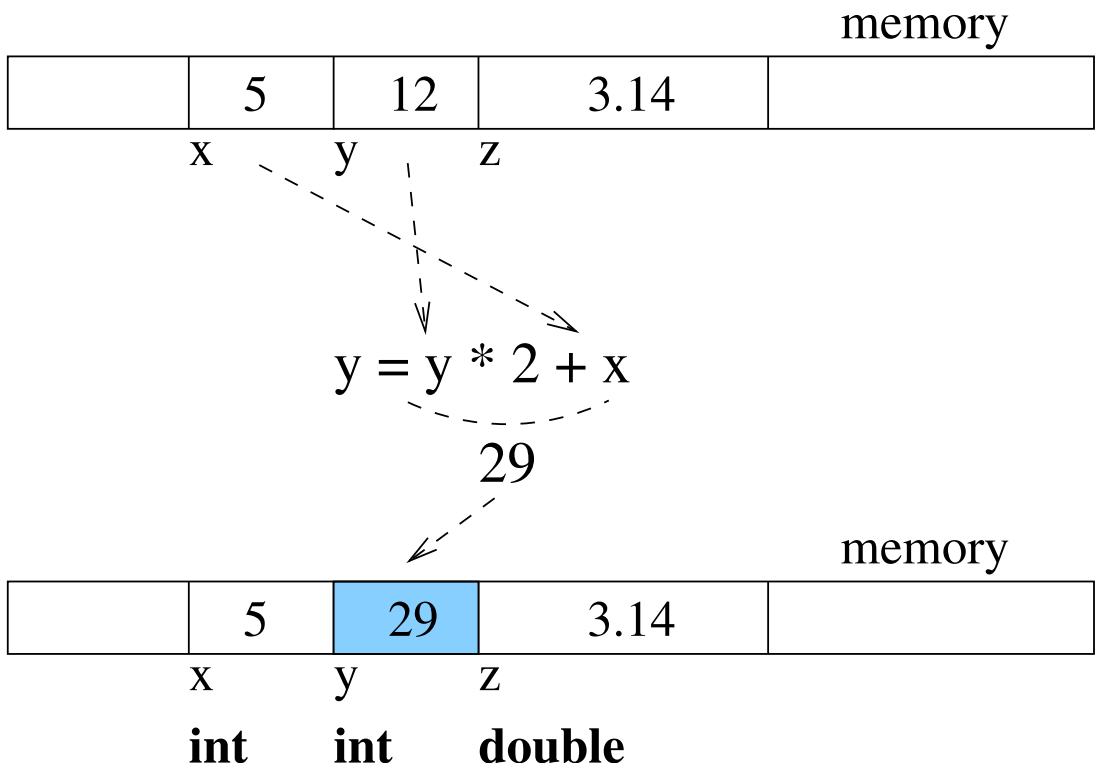
```
1   int   x(5), y(12);
2   double   z = 3.14;
```

manipulating objects: assignment

```
LeftExpression = RightExpression
```

LeftExpression must evaluate to **lvalue**.

```
1  int    x(5);
2  int    y(x+7);
3  double   z(3.14);
4
5  y = y * 2 + x;
```

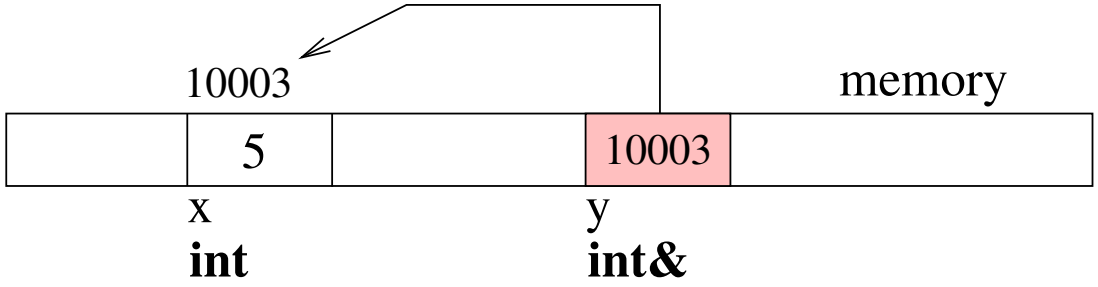


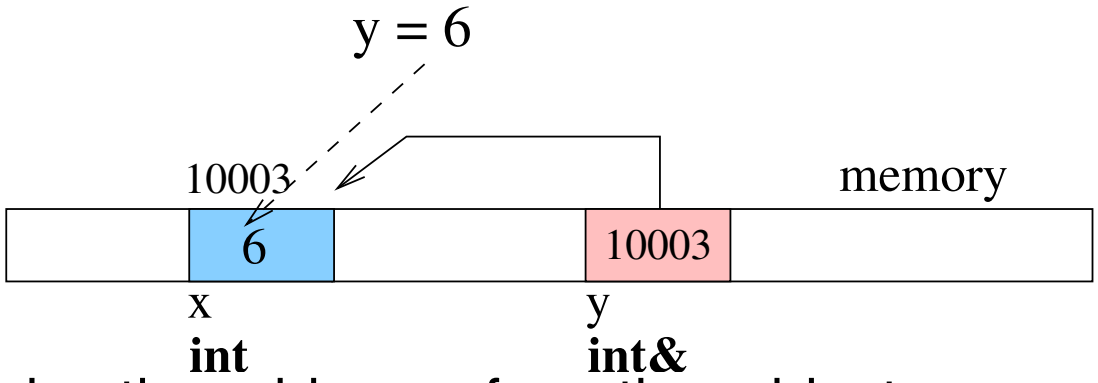
reference variables

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

Expression must yield **lvalue** of type NameOfType

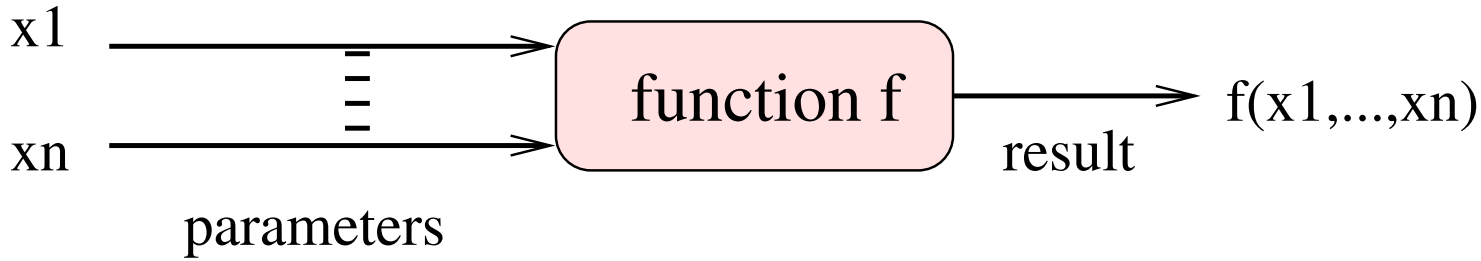
```
1  int  x(5);
2  int&  y(x);
3  // type of y is int&
4
5  y = 6;
```





Reference variable contains the address of another object.

functions




```
1  int  x(3);
2  int  y(0);
3  int  z(5);
4
5
6  y = x * x;
7  y = y + z * z;
```

⇒

```
1  int  // return type
2  square(int u) {
3    // u is formal parameter
4    return u*u; // return value
5  }
6
7  y = square(x) + square(z);
```

function definition syntax

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

```
1  int  // return type
2  square(int u) { // u is formal parameter
3    return u*u; // return value
4  }
```

calling a function

```
1  int  // return type
2  square(int u) { // u is formal parameter
3    return u*u; // return value
4  }
```

```
y = square(x+1); //x = 3
```

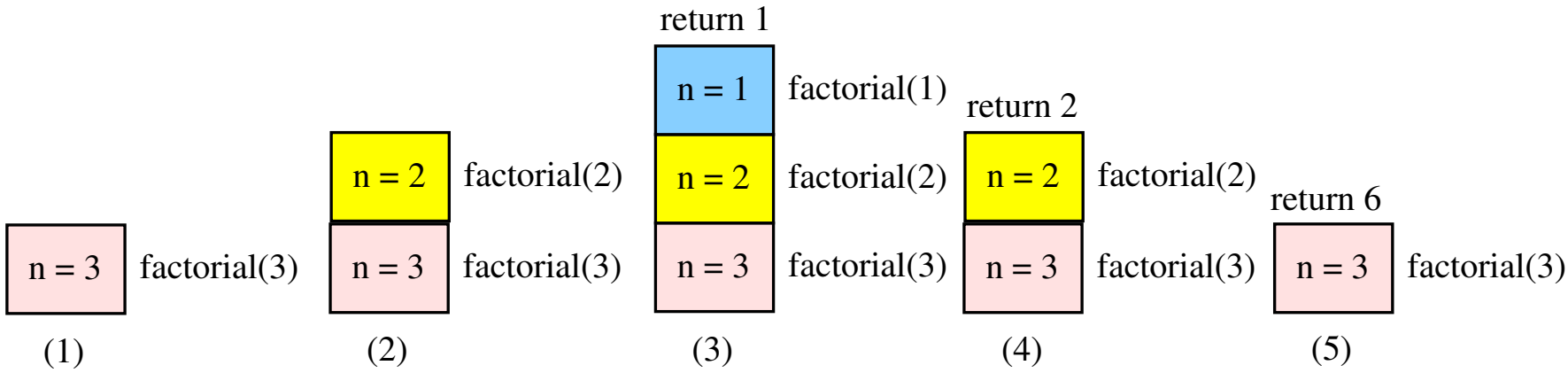
1. Initialize fresh **local parameter object** with the value of the parameter expression corresponding tot the **formal parameter**:

   ```
   int u(x+1);
   ```

2. Execute statements in the function **body**. This occurs in an **environment** containing local (`u`) and non-local names.
3. To evaluate `return exp`: create new object initialized with value of `exp` $\Rightarrow$ `int tmp(u*u);`
4. Caller processes new object $\Rightarrow$ `y = tmp;`
5. Deallocate memory of function call **frame**.

the frame stack

```
1  int
2  factorial(int n) {
3    if (n<2)
4      return 1;
5    else
6      return n * factorial(n-1);
7  }
8
9  factorial(3);
```



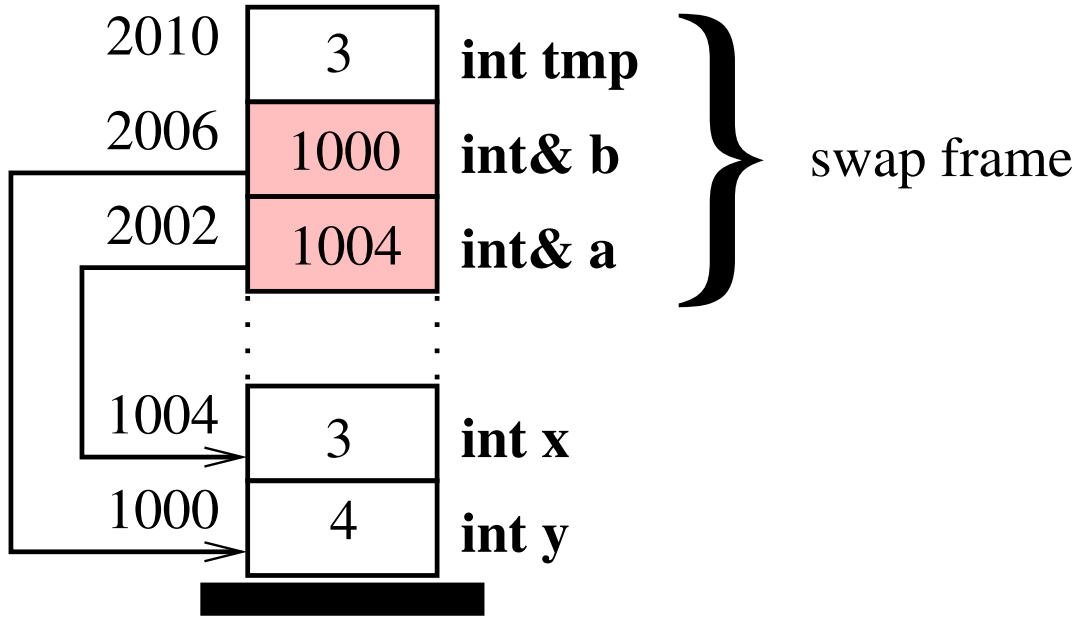
parameter passing

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) {
  // for swap(x,y):
  // int& a(x)
  // int& b(y)
  int  tmp(a);
  a = b;
  b = tmp;
}

swap(x, y);
```



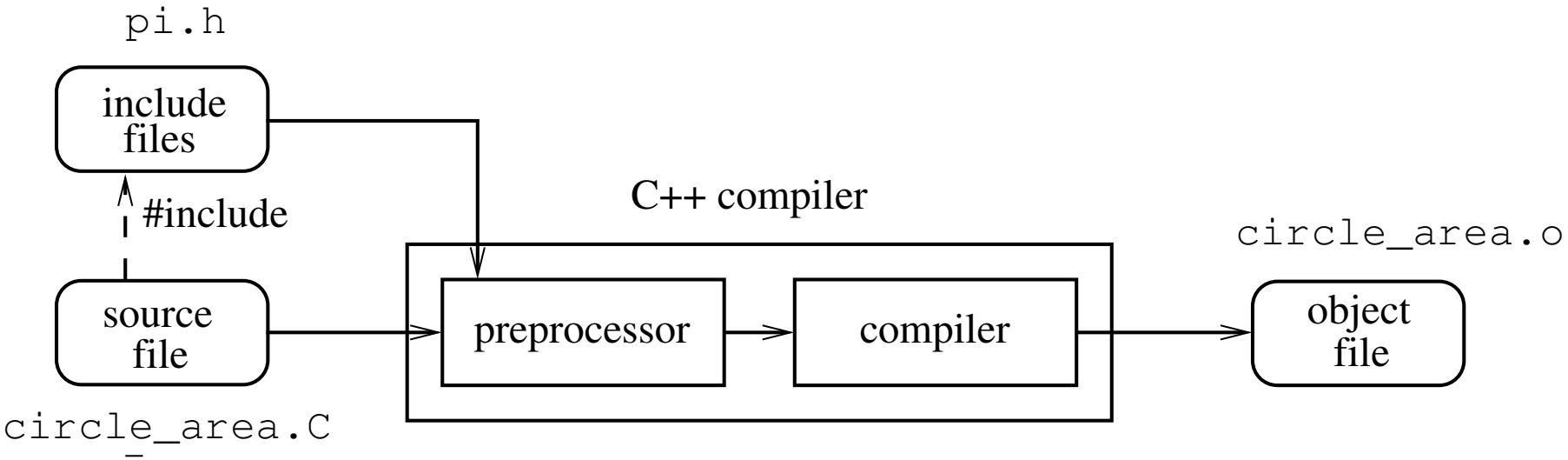
program structure

- 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.

```
1 extern double pi; // variable
2 double circle_area(double radius); // 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

```
1  #ifndef PI_H // begin conditional inclusion:
2  // read stuff until endif ONLY if PI_H is not defined
3
4  #define PI_H // assign 1 to the preprocessor
5  // variable PI_H, next time this file is read..
6
7
8  extern double pi; // declaration of variable pi
9
10
11 #endif // end of conditional inclusion
```

pi.C

```
1  #include "pi.h" // include declaration to ensure that
2                  // declaration and definition of pi
3                  // are consistent
4
5  double pi(3.14); // definition of variable pi
```

circle.h

```
1  #ifndef CIRCLE_H
2  #define CIRCLE_H
3  // The ifndef define .. endif sequence ensures
4  // that the following declaration is not read twice
5  // (see also pi.h)
6
7  // declaration of function 'circle_circumference'
8  double circle_circumference(double radius);
9
10 // declaration of function 'circle_area'
11 double circle_area(double radius);
12
13 #endif
```

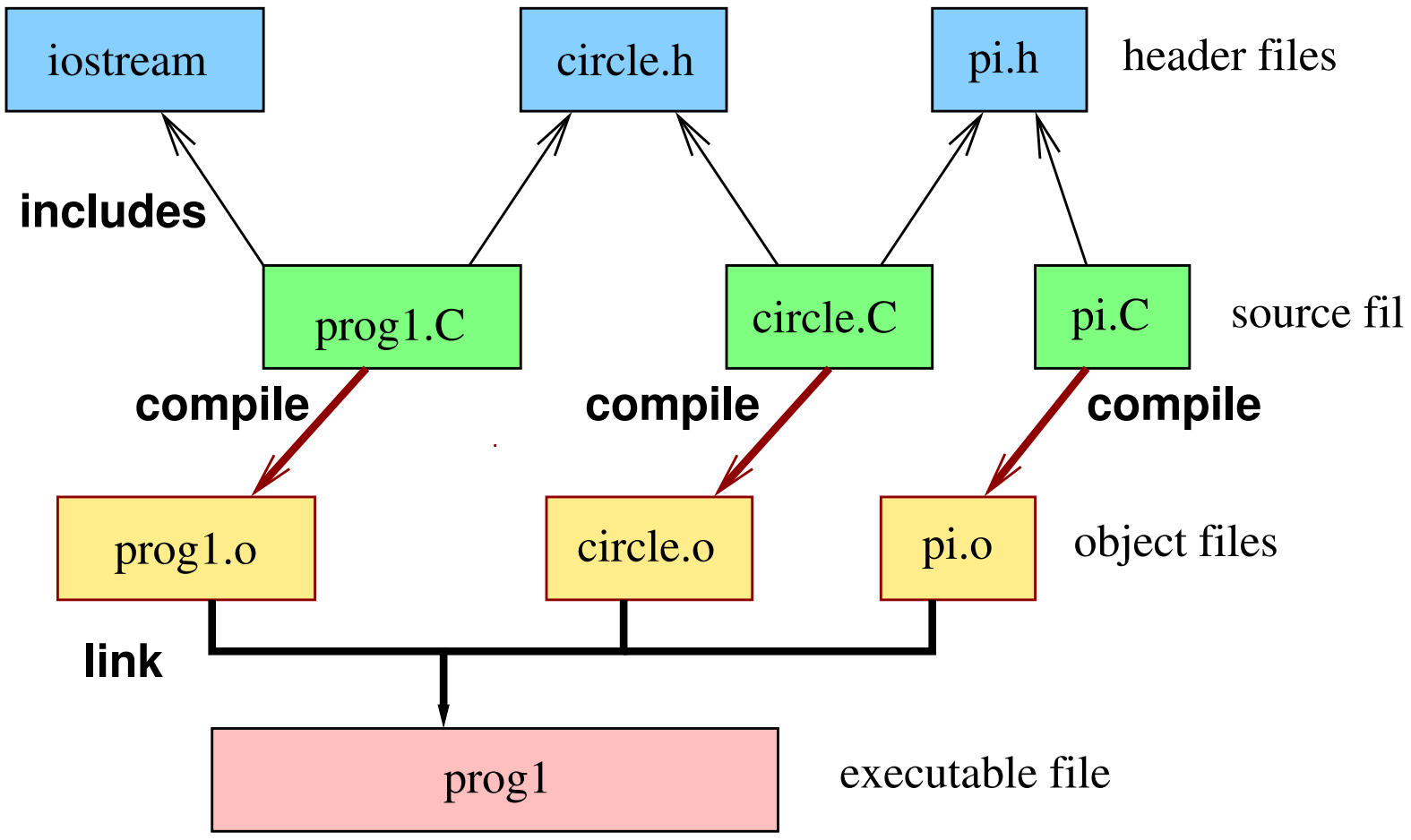
circle.C

```
1  #include "pi.h" // declaration of pi
2  #include "circle.h" // declaration of functions
3  // that are defined in this file:
4  // including them ensures the consistency of
5  // the declarations with the following definitions
6
7  double // function definition
8  circle_circumference(double radius) {
9    return 2 * pi * radius;
10 }
11
12 double // function definition
13 circle_area(double radius) {
14   return pi * radius * radius;
15 }
```

prog1.C

```
1  #include <iostream> // contains needed declarations,
2  // e.g. for operator<<(ostream&, X)
3
4  #include "circle.h" // contains needed declaration
5  // of circle_area
6
7  int // function definition
8  main() {
9    double r(35); // local variable definition
10   // r will live in the call frame
11
12   // the next expression writes the area of the
13   // circle to standard output
14   std::cout << circle_area(r) << std::endl;
15
16   return 0; // all is well: return 0
17 }
```

prog1 structure



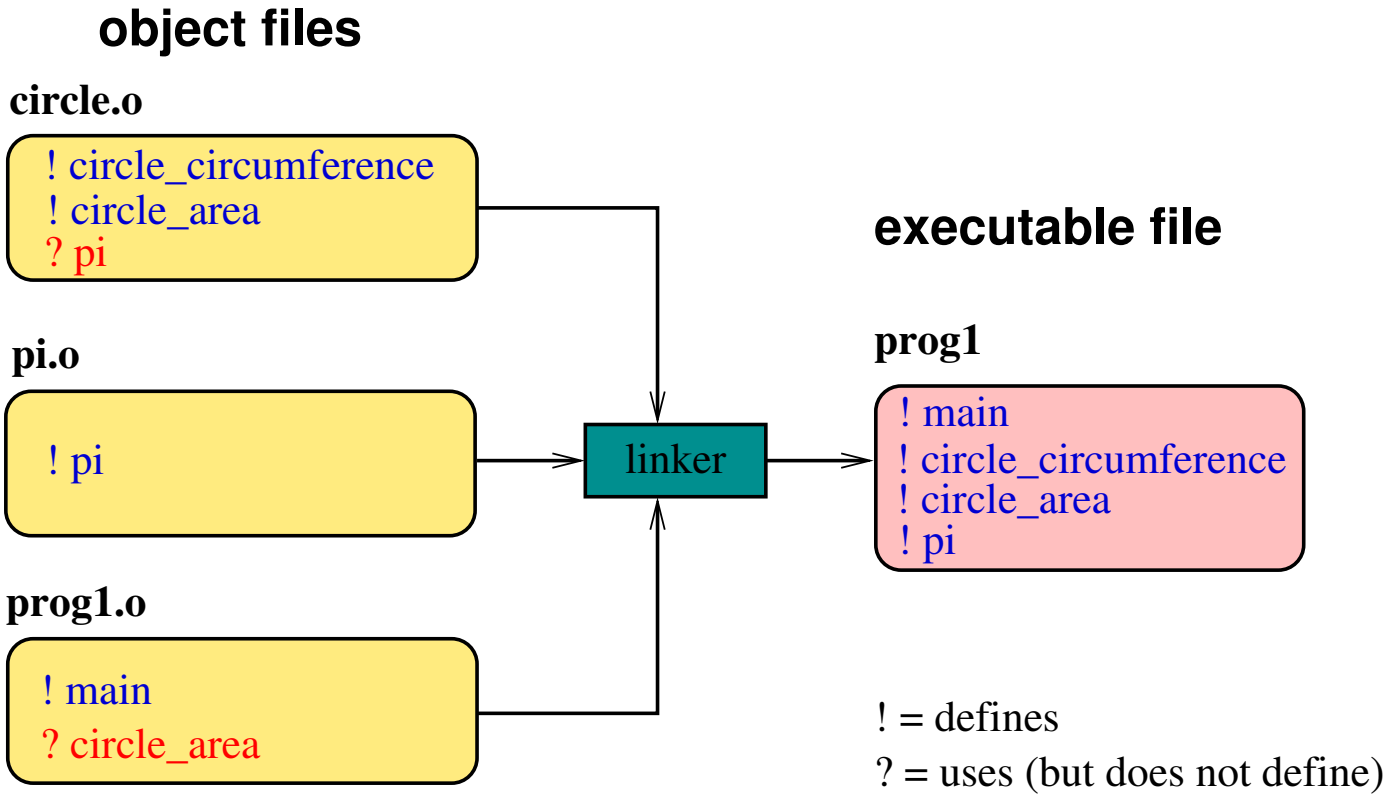
Executing a C++ program

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.

```
1  wilma% prog1 || echo "error"
```

linking a program

- resolve symbols (e.g. circle_area in prog1.o)
- relocate code



linking example: prog1.o

```
1  00000000 <main>:
2     0:   55                          pushl  %ebp
3    ...
4    1a:   dd 1c 24                    fstpl  (%esp,1)
5    1d:   e8 fc ff ff ff              call   1e <main+0x1e>
6    ...
```

linking example: prog1

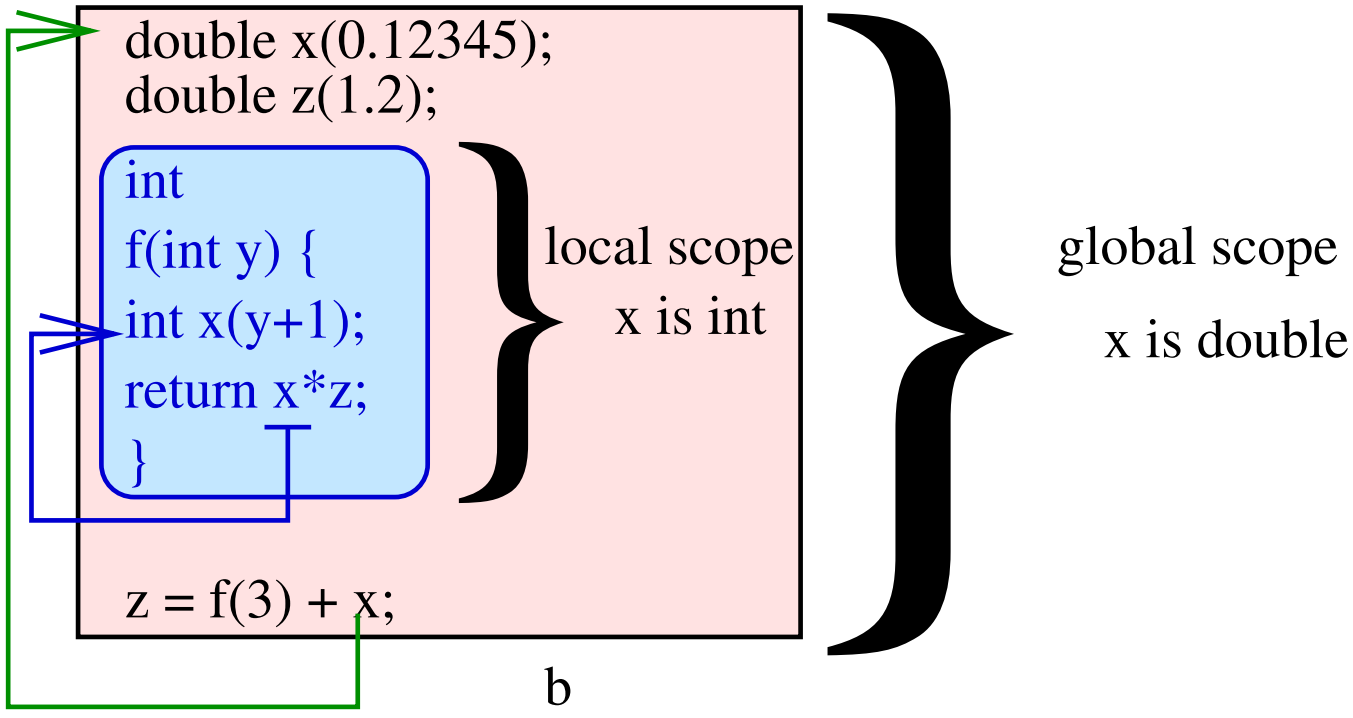
```
1  08048710 <main>:
2   8048710:  55                pushl  %ebp
3   ..
4   804872a:  dd 1c 24          fstpl  (%esp,1)
5   804872d:  e8 36 00 00 00    call   8048768 <circle_area(double)>
6   ..
7  08048768 <circle_area(double)>:
8   8048768:  55                pushl  %ebp
```

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

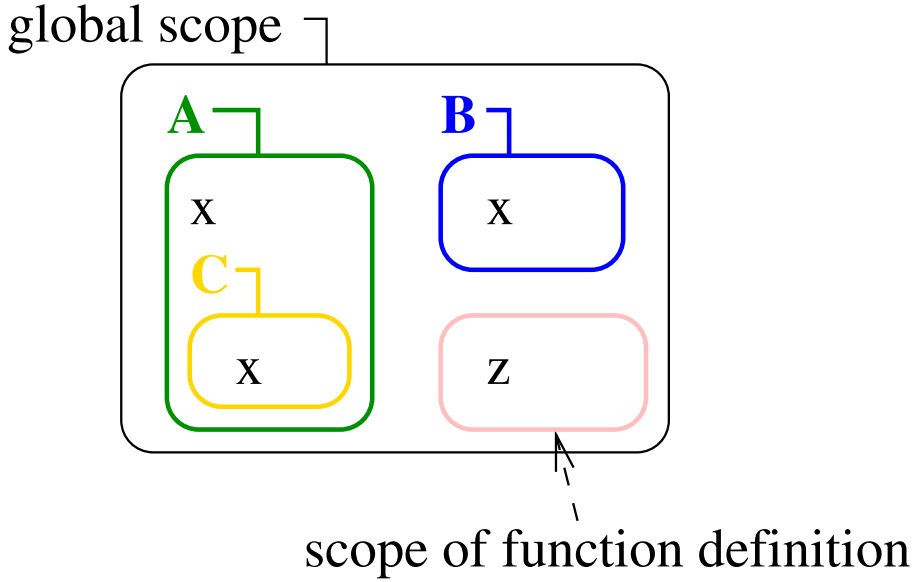
scopes

- 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

```
1  namespace A {
2    int x;
3    namespace C { int x; }
4    }
5  namespace B { int x; }
6
7  int
8  main() {
9    int z(2);
10   // error: which x?
11   z = x;
12 }
```



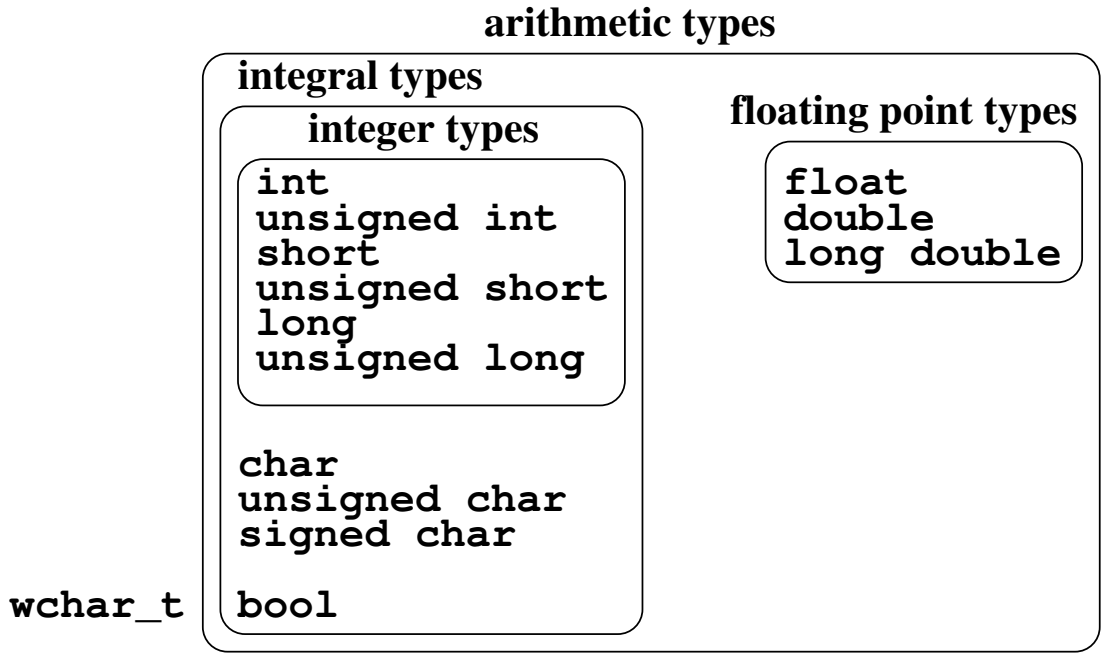

```
1  // possible correction for z=x
2  z = A::C::x; // ok
3  // other possibility:
4  using A::C::x;
5  z = x;
```

using namespaces

```
1  namespace A {
2    namespace C { extern int x; } // only a declaration
3    }
4  namespace B { int x; }
5
6  int A::C::x; // definition of name "x" declared in ::A::C
7
8  namespace A { int x; } // continuation of namespace A
9
10 int
11 main() {
12   int z(2);
13   using namespace A::C;
14   z = x; // thus "x" refers to A::C::x
15 }
```

Outline

primitive types




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

conversions

```
1  long  l('a'); // char fits into l, no problem; l = 97
2  int   i(3.14); // compiler will warn; i will be 3
3  double  d(0.5);
4  float  f(0.6);
5  int   i(0);
6
7  i = d + f; // i will become 1
```

conversions

```
1  long  l('a'); // char fits into l, no problem; l = 97
2  int   i(3.14); // compiler will warn; i will be 3
3  double  d(0.5);
4  float  f(0.6);
5  int  i(0);
6
7  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).

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

operations on primitive types

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

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

- Assignment is expression and can be compounded.

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

operations on primitive types

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

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

- Assignment is expression and can be compounded.

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

- I/O expressions

```
1  std::ostream& operator<<(std::ostream&, int);
2  std::istream& operator>>(std::istream&, int&);
3
4  std::cin >> x >> y;
5  std::cout << x << y;
```

operations on arithmetic types

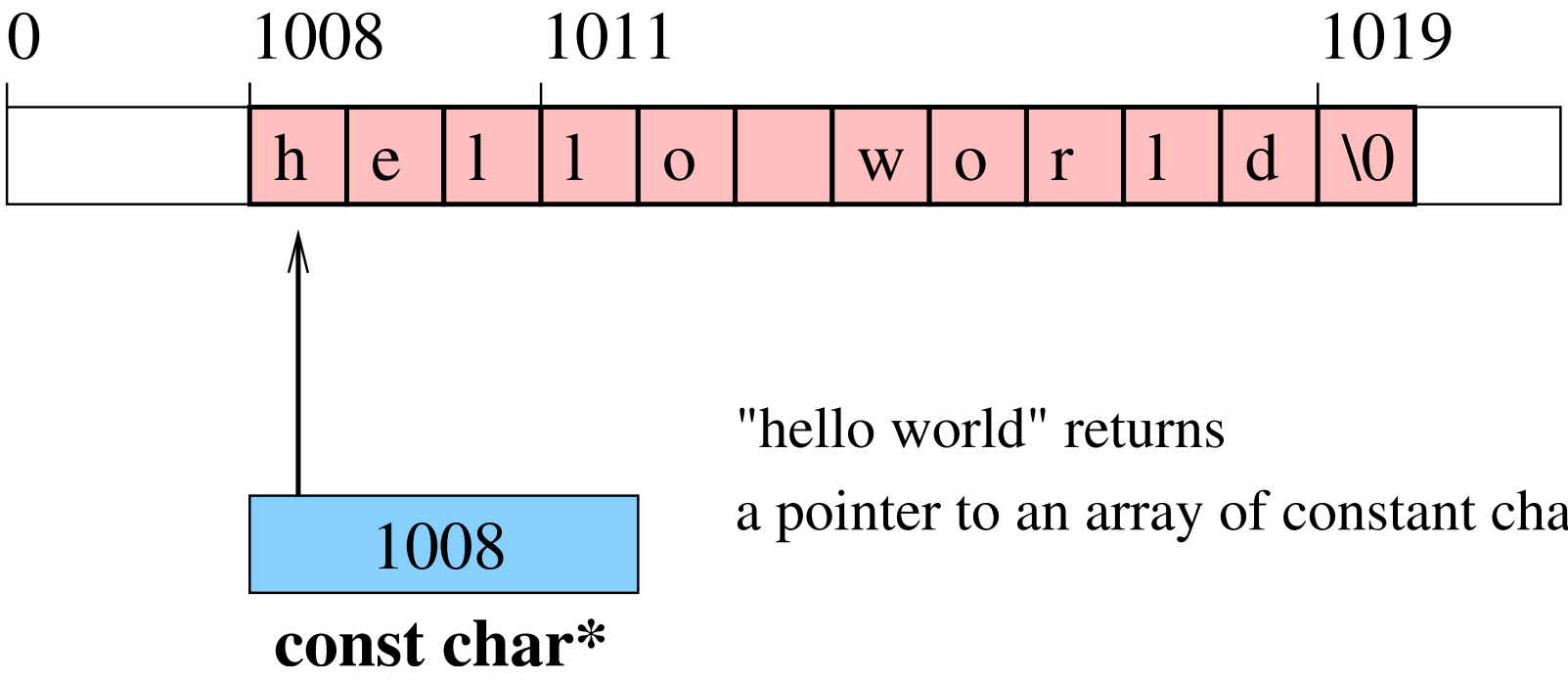
```
1  ++li; --li;  // increment, decrement
2  la++; la--; // return, then increment
3  sizeof(x); // number of bytes used by x
4  ~i; // bitwise complement
5  i & j | k ^ l; // bitwise and/or/xor
6  a + b - c * d / e; // arithmetic
7  -a; // unary minus
8  i % j; // remainder (modulo)
9  x << 4; // shift left
10 x >> 4; // shift right
11 std::cin >> x; // input
12 std::cout << x; // output
13 x < y > z <= u >= v; // comparison
14 !i; // logical complement
15 i && j || k; // logical and/or
16 x = y; // assign
17 x += y; // x = x + y, also -= etc.
```

initialisation of primitive types

If no initial value is specified:

- global or locally static variables of primitive type are automatically initialized to 0
- automatic variables are **not** initialized

(C) string literals




```
1  std::ostream& operator<<(std::ostream&, const char*);
2  const char* hi("hello world");
3
4  std::cout << "hello world"; // hello world
5  std::cout << "hello" "world"; // helloworld
6  std::cout << "hello world\n";
```

Outline

function declarations

$$\boxed{\texttt{T}_0\ \ \texttt{f(T}_1\texttt{,..,T}_n\texttt{)}}$$

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

```
1  double
2  my_function(int a, bool b) {
3    // function body
4  }
```

type of my_function	$[int \times bool \rightarrow double]$
signature of my_function	$int \times bool$
return type of my_function	$double$

function may have side effects

Side effect: when two identical function calls return different answers.

```
1  int x(0); // global variable
2
3  int f() {
4    return x++; // change and return global
5  }
6
7  int
8  main() {
9    f(); // will return 0
10   f(); // will return 1
11 }
```

default parameters

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

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

```
1  // print i1 separator i2, use base for representing number
2  void printline(int i1, int i2,
3    char separator = '\t', int base = 10);
4  // output?
5  printline(10,12,',',8);
6  printline(10,12);
7  printline(10,12,'|');
8  printline(10,12,8);
```

default parameters example

```
1  printline(10,12,',',8); // 12,14
2  printline(10,12); // 10 12
3  printline(10,12,'|'); // 10|12
4  printline(10,12,8); // same as printline(10,12,'\b',10);
5
6  print(const Student&, std::ostream& os = std::cout);
7  print(fred); // print(fred, std::cout);
8  print(lisa, std::cerr);
```

unspecified number of parameters

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

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

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

inline functions

```
1  inline int
2  maximum(int i1, int i2) {
3    // return the largest of two integers
4    if (i1>i2)
5      return i1;
6    else
7      return i2;
8  }
```

Compiler will replace calls “in line”:

```
1  i = maximum(12,x+3)
```

$\Rightarrow$

```
1  int tmp(x+3);
2
3  if (12>tmp)
4    i = 12;
5  else
6    i = tmp;
```

Where to put definitions of inline functions? why?

overloading function definitions

Functions with the same name but different signatures are different.

```
1  int
2  sum(int i1, int i2, int i3) {
3    return i1+i2+i3;
4  }
5
6  int
7  sum(int i1, int i2, int i3, int i4) {
8    return i1+i2+i3+i4;
9  }
10
11 int
12 main() {
13   sum(1, 2, 3, 4); // sum(int,int,int,int)
14 }
```

overloading function definitions

```
1  #include <math.h> // contains declaration for rint()
2
3  int round(double a) {
4    return int(rint(a));
5    // rint will round 'a' to the nearest integer
6    // (as a double) see also 'man rint'
7  }
8
9  int round(int i) {
10   return i;
11 }
12
13 int
14 main() {
15   round(1.1); // will call round(double)
16   round('a'); // error?
17 }
```

determining which function is called

$\boxed{\mathbf{f}(\mathbf{e_1}, \ldots, \mathbf{e_n})}$ unqualified, e.g. not for `N::f`

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.
   - same for namespaces where types of arguments of f are declared (Koenig lookup)
2. Find the **best match** in F for the call, i.e. the declaration f' whose *signature* best matches the call's signature $(T_{e_1}, \ldots, T_{e_n})$.

notes on lookup

- Definitions in a closer scope hide definitions in a wider scope.

```
1  namespace N {
2    void f(int);
3    namespace M {
4      int f;
5      namespace K {
6        void g() {
7          f(4); // compile error: why?
8          }
9      }
10   }
11 }
```

- Default and unspecified arguments are taken into account when determining F.

overloaded example

```
1  #include <iostream>
2  std::ostream& std::operator<<(std::ostream&, char c);
3  std::ostream& std::operator<<(std::ostream&, unsigned char c);
4  std::ostream& std::operator<<(std::ostream&, signed char c);
5  std::ostream& std::operator<<(std::ostream&, const char *s);
6  std::ostream& std::operator<<(std::ostream&, const unsigned ch
7  std::ostream& std::operator<<(std::ostream&, const signed char
8
9  int f(int i, int j=0) { return 1; }
10 int f(int k) { return 2; }
11
12 int
13 main() {
14   f(3); // which f?
15   // Next line illustrates Koenig lookup:
16   //   std::operator<<(ostream&,const char *s);
17   std::cout << "hello world";
18 }
```

function definition

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

kinds of statements

- expression statement, incl. function call, assignment.

  `OptionalExpression ;`

- definition statement, e.g. for local variables.

  `ObjectDefinition ;`

- control flow statement, some have an expression analogue

Example:

```
1  x = f(y); // expression statement
2  double d(3.14); // definition statement
3
4  while (x>0) // control flow statement
5    x = f(f(x));
```

the compound statement

A compound statement defines a new scope.

```
{
   OptionalStatementList
}
```

Example:

```
1  {
2    int tmp(x);
3    x = y;
4    y = tmp;
5  }
```

the sequence operator

Expression analogue of compound statement.

Expression$_1$, Expression$_2$, ..., Expression$_n$

Example:

```
1  x = (std::cin >> y, 2 * y);
```

≈

```
1  std::cin >> y;
2  x =  2 * y;
```

the if statement

if (Expression) ** StatementIfTrue**	**if (Expression)** ** StatementIfTrue** **else** ** StatementIfFalse**

Example:

```
1  if (a>b) // StatementIfTrue is if statement
2    if (c>d) // a>b && c>d; StatementIfTrue is
3      x = 1; // expression statement
4    else
5      x = 2; // a>b && c<=d
6
7  if (x>10) { // StatementIfTrue is compound statement
8    int tmp(y);
9    y = x;
10   x = tmp;
11   }
```

the ? operator

Expression analogue of if statement.

Condition ? ExpressionIf : ExpressionElse

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

≈

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

the while statement

```
while (Expression)
  Statement
```

Same effect as

```
1  if (Expression) {
2    Statement;
3    while (Expression)
4      Statement;
5    }
```

while statement example

```
1  int
2  factorial(int n) {
3    int result(1);
4    while (n>1)  {
5      result *= n; // can be done in 1 line
6      --n;
7    }
8    return result;
9  }
```

the do statement

```
do
  Statement
while (Expression)
```

is equivalent to

```
Statement
while (Expression)
  Statement
```

```
1  int sum(0);
2  do {
3    int i(0);
4    std::cin >> i;
5    sum += i;
6  }
7  while (std::cin); // while state of input stream is ok
```

the for statement

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

is equivalent to

```
1  ForInitialization;
2  while (ForCondition) {
3    Statement
4    ForStep;
5    }
```

for statement example

```
1  int
2  factorial(int n) {
3    // make a copy of the input, but 0 becomes 1
4    int  result(n?n:1);
5    // if n <=2, the result is n and we are done
6    if (result>2)
7      // multiply with every number < n, except 1
8      for (int j=2;(j<n);++j)
9        // use compound assignment for multiplication
10       result *= j;
11   return result;
12 }
```

the switch statement

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

is equivalent to

```
1  int tmp(Expression); // Must be integral: why?
2  if (tmp==Constant1)
3    Statement1
4  else if (tmp==Constant2)
5    Statement2
6  else if ...
7  else Statement // default
```

switch example 1

```
1  #include <stdlib.h> // for the random() functions
2  #include <time.h> // for the time() function
3
4  int
5  throwdice(int score) { // throw dice and return new score
6    // use current time() value as seed for the
7    srandom(time(0)); // random number generator
8    int result(random() % 6 + 1); // throw dice
9    switch (result) {
10     // update score, depending on result
11     case 1: score += 1; break;
12     case 2: score += 3; break;
13     case 3: score += 4; break;
14     case 4: score += 6; break;
15     case 5: score += 6; break;
16     case 6: score += 8; break;
17     default:
18       abort(); // should not happen
19   }
20   return score;
21 }
```

switch example 2

```
1  bool
2  // return true iff c is a vowel letter
3  isvowel(char c) {
4    switch(c) {
5      case 'a': case 'e': case 'i':
6      case 'o': case 'u':
7      case 'A': case 'E': case 'I':
8      case 'O': case 'U':
9            // no need for break; we exit immediately
10           // from the function
11           return true;
12     default:
13           return false;
14   }
15   abort(); // should never get here
16 }
```

the return and break statements

```
return Expression;
break;
```

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

```
1  #include <iostream>
2
3  void
4  break_demo() { // output?
5    for (int j=0; (j<10); ++j) {
6      for (int i=0; (i<5); ++i) {
7        if (i>3)
8          break;
9        std::cout << i << " ";
10     }
11     std::cout << "\n";
12   }
13 }
```

the continue statement

The **continue** statement interrupts the current iteration and immediately starts the next iteration of the enclosing loop

```
1  #include <iostream>
2
3  void process(int); // declaration
4
5  int
6  main() {
7    int  i(0);
8
9    while (std::cin>>i) {
10     // an input stream can be converted to a bool;
11     // .. the result is true iff input ok
12     if (i<=0) // i not positive, throw away
13       continue;
14     if (i%2) // i odd, throw away
15       continue;
16     process(i); // i ok, process it
17   }
18 }
```

automatic local objects

Automatic local objects for local variables are defined in a function body or compound statement scope and are destroyed when control leaves the scope (pop frame stack).

```
1  void
2  f() {
3    { // Start compound statement.
4      // It has its own nested scope.
5      int x(3); // local object definition
6      x *= 2;
7    } // End compound statement and scope.
8    std::cout << x; // ???
9  }
```

static local objects

Static local objects are **persistent** across function calls or scope entries (but the same accessibility rules as for other local variables apply).

```
1  int
2  counter() {
3    static int n(0); // Created when function is first called.
4    return ++n;
5  }
6
7  std::cout << counter() << counter() << std::endl;
```

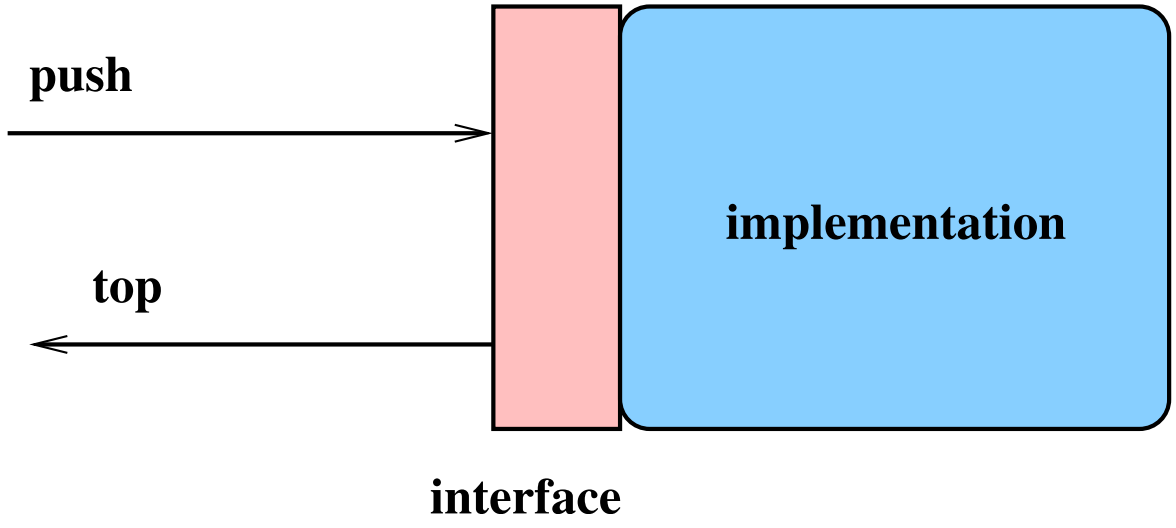
return reference to static object

It is safe to return a reference to a **static** local object.

```
1  int&
2  silly() {
3    static int n(0);
4    return n; // ok, n is static
5  }
6
7  silly() += 3; std::cout << silly(); // output?
```

Outline

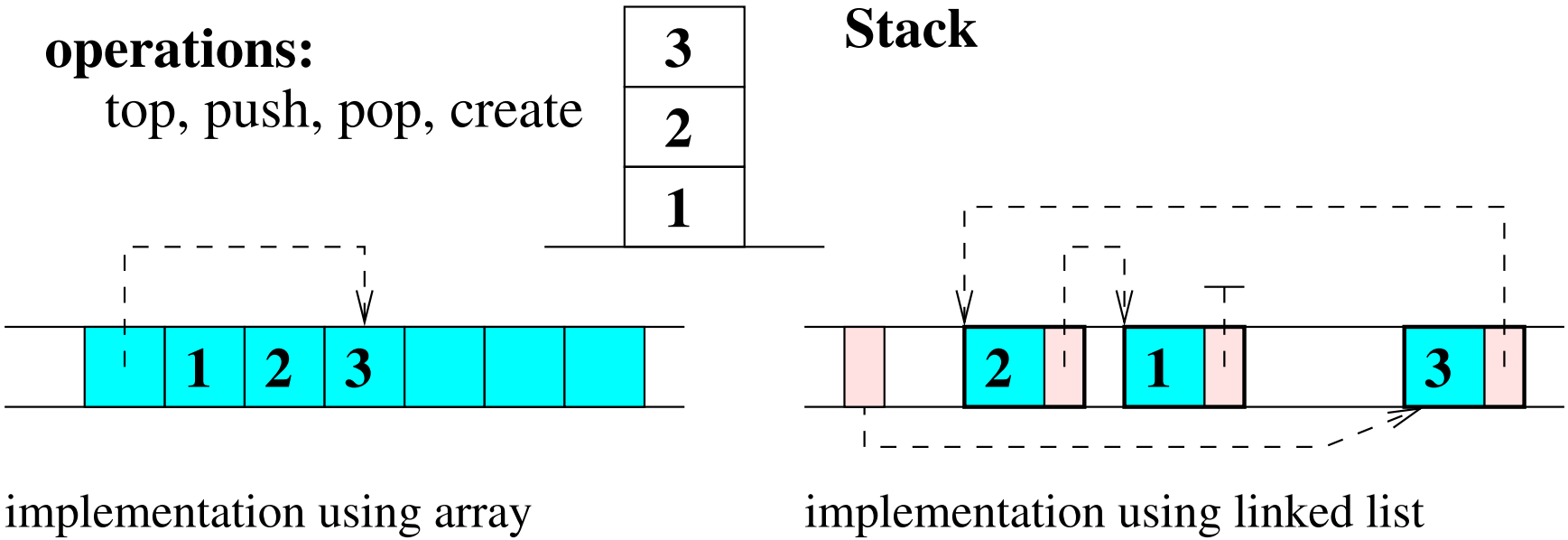
abstract data types (adt's)



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



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

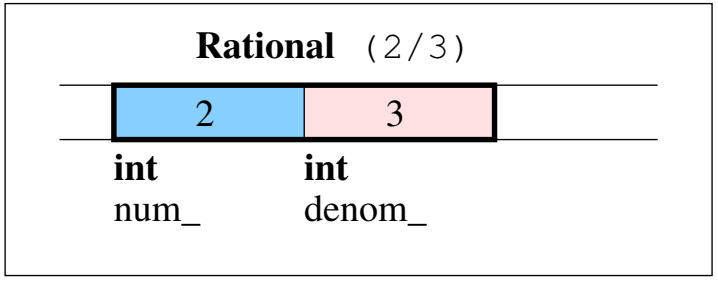
C++ ADTs:

```
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

```
1  #ifndef RATIONAL_H
2  #define RATIONAL_H
3  class Rational { // defines a scope called Rational
4    public:
5      // public interface to be supplied
6    private:
7      // implementation part
8      int num_;
9      int denom_; // must not be 0!
10 };
11 #endif
```




```
1  Rational r; // just another object definition
```

member function declarations

```
1  #ifndef RATIONAL_H
2  #define RATIONAL_H
3  class Rational {
4    public: // public member function declarations
5      Rational multiply(Rational r);
6      Rational add(Rational r);
7    private: // implementation part
8      int num_;
9      int denom_; // must not be 0!
10 };
11 #endif
```

Calling a member function: specify target class object

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

member functions may be overloaded

```
1  class Rational {
2    public: // public member function declarations
3      Rational multiply(Rational r);
4      Rational add(Rational r);
5      Rational add(double d);
6    private: // implementation part
7      int num_;
8      int denom_; // must not be 0!
9  };
10
11 Rational r1;
12 Rational r2;
13 r1.add(r2);
14 r1.add(3.3);
```

constructors: initializing a class object

```
1  #ifndef RATIONAL_H
2  #define RATIONAL_H
3  class Rational {
4    public:
5      // constructor member function declaration
6      Rational(int num, int denom);
7      Rational multiply(Rational r);
8      Rational add(Rational r);
9    private:
10     int num_;
11     int denom_; // must not be 0!
12 };
13 #endif
```

Constructors are more flexible than just giving an initial value.

```
1  Rational r(2,3); // initialize r with Rational::Rational(2,3)
```

overloading constructors

```
1  class Rational {
2    public: // overloaded constructors
3      Rational(int num, int denom); // initializes to num/denom
4      Rational(int num); // initializes to num/1
5      Rational(); // default constructor; initializes to 0/1
6      Rational(const Rational& r); // copy constructor:
7        // initializes to copy of r
8
9      Rational multiply(Rational r);
10     Rational add(Rational r);
11   private:
12     int num_;
13     int denom_; // must not be 0!
14 };
15
16 // why not Rational r1();?
17 Rational r1; // calls Rational::Rational();
18 Rational r2(r1); // calls Rational::Rational(r1)
19 Rational r3(5); // calls Rational::Rational(5)
```

the default constructor

```
1  class Rational {
2    public:
3      Rational(int num, int denom);
4    private:
5      int num_;
6      int denom_;
7  };
8
9  Rational r;
10 // compile error; why?
```

```
1  class Rational {
2    private:
3      int num_;
4      int denom_;
5  };
6
7
8
9  Rational r;
10 // what exactly happens?
```

the copy constructor (cctor)

```
1  class Rational {
2    public:
3      Rational(int num, int denom);
4    private:
5      int num_;
6      int denom_;
7  };
8
9  int f(Rational r) {
10   ...
11 }
```

Cctor is used for passing class objects by value

```
1  Rational x;
2  f(x); // copy x using Rational::Rational(x);
```

The compiler will provide a default cctor (which does bitwise copy) if it is not explicitly defined.

member function definition

```
1  #include          "rational.h"
2
3  Rational // note scope operator: multiply in scope Rational
4  Rational::multiply(Rational r) { // a/b * c/d = (a*c)/(c*d)
5    int num(num_ * r.num_);
6    int denom(denom_ * r.denom_);
7    // return object is initialized using ctor
8    return Rational(num, denom);
9  }
10
11 Rational
12 Rational::add(Rational r) { // a/b + c/d = (a*d + c*b)/(b*d)
13   int num = num_ * r.denom_ + r.num_ * denom_;
14   int denom = denom_ * r.denom_;
15   return Rational(num, denom);
16 }
```

ctor definition

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

```
1  Rational::Rational(int num, int denom):
2    num_(num), denom_(denom) /* member initialization  */ {
3      // check that denom is not 0, if it is we abort
4      if (denom==0)
5        abort();
6  }
```

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

```
1  Rational::Rational(int num, int denom) {
2    num_ = num;
3    denom_ = denom;
4    if (denom==0)
5      abort();
6  }
```

member initialization and ctors

Data members are initialized **before** constructor body is executed:

1. Initialize members, using the member type's default ctor if no member initialization was specified in the ctor definition (if the member type is not primitive and has no default ctor: compile error). Note that primitive types do not have a default ctor.
2. Execute body of the constructor.

To initialize the "whole", first initialize the parts, then do additional work, if any.

inline member function definition

```
1  #ifndef RATIONAL_H
2  #define RATIONAL_H
3  class Rational {
4    public:  // interface
5      Rational(int num, int denom): num_(num), denom_(denom) {
6        if (denom==0)
7          abort();
8      }
9      Rational multiply(Rational r) {
10       return Rational(num_ * r.num_, denom_ * r.denom_);
11     }
12     Rational add(Rational r) {
13       return Rational(num_ * r.denom_ + r.num_ * denom_,
14           denom_ * r.denom_);
15     }
16   private: // implementation part
17     int num_;
18     int denom_; // must not be 0!
19 };
20 #endif
```

members with default parameters

```
1  #ifndef RATIONAL_H
2  #define RATIONAL_H
3  class Rational {
4    public:
5      // using default pars saves 2 overloaded
6      // ctor functions
7      Rational(int num=0, int denom=1);
8
9      Rational multiply(Rational r);
10     Rational add(Rational r);
11   private:
12     int num_;
13     int denom_; // must not be 0!
14 };
15 #endif
```

user-defined conversions

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

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

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

```
1  class Rational {
2    public:
3      ... // explain definition below
4      operator double() { return double(num_)/denom_; }
5    ..
6  };
7
8  Rational r(1, 3);
9  // the following prints 0.333; it first converts:
10 // double tmp(r.operator double()); operator<<(cout,tmp);
11 std::cout << r;
```

operator overloading

```
1  #ifndef RATIONAL_H
2  #define RATIONAL_H
3  class Rational {
4    public:
5      Rational(int num=0, int denom=1);
6      Rational operator+(Rational r) { return add(r); }
7      Rational operator*(Rational r) { return multiply(r); }
8      Rational multiply(Rational r);
9      Rational add(Rational r);
10   private:
11     int num_;
12     int denom_; // must not be 0!
13 };
14 #endif
```

Example:

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

overloading by non-member functions

problem:

```
1  Rational r;
2  r+2; // Rational tmp(2); r.operator+(tmp);
3  2+r; // compile error: no function operator+(int,Rational)
```

solution:

```
1  inline Rational operator+(Rational r1, Rational r2) {
2    return r1.add(r2);
3  }
```

now, normal conversion strategy will work:

```
1  2+r; // Rational tmp(2); operator+(tmp, r);
```

operators that can be overloaded

```
1  [], (), ++, --, !, -, +, *, /, %,
2  new, delete, new[], delete[], /
3  -> , <<, >>, < , <=, >, >=, == ,
4  =, !=, &&, ||, &, |, *=, +=, ..
```

- `=`, `[]`, `()` 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

```
1  #include <math.h> // for rint(double)
2  // which rounds a double to the nearest integer
3
4  class Rational {
5    public:
6      Rational(int num=0, int denom=1);
7      ..
8      Rational& operator=(double d) {
9        int units(rint(d));
10       int hundreds(rint((d - units) * 100));
11       num_ = units * 100 + hundreds;
12       denom_ = 100;
13       return *this;
14     }
15   private:
16     ..
17 };
18
19 Rational r;
20 r = 1.3; // sets r to 130/100
```

default assignment operator

A default implementation of `C& operator=(const C&)` is available for each class `C`. It performs a bitwise copy of the data members.

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

overloading ++, -- operators

```
1  class Rational {
2    public:
3    ..
4      Rational operator++() { // prefix version, e.g. ++r
5        Rational r(num_+denom_, denom_);
6        num_ += denom_;
7        return r;
8      }
9      Rational operator++(int) { // postfix version, e.g. r++
10       Rational r(num_, denom_);
11       num_ += denom_;
12       return r;
13     }
14   private:
15     ..
16 };
17
18 Rational r(1, 2);
19 r1 = ++r; // r1 = r = 3/2
20 r2 = r++; // r2 = 3/2, r = 4/2
```

forbidding operators

```
1  class Server {
2    public:
3      Server(std::ostream& log, int port);
4      ~Server();
5      // lots of stuff omitted
6    private:
7      // we forbid making copies of a Server object by
8      // declaring the copy constructor and assignment
9      // operators to be private (no definition is
10     // needed, by the way).
11     Server(const Server&);
12     Server& operator=(const Server&);
13     std::ostream&  log;
14     // stuff omitted
15 };
16
17 void start_protocol_bad(Server s); // error: why?
18 void start_protocol_ok(Server& s); // ok
```

finalizing objects using destructors

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

Called by the system before the object is destroyed.

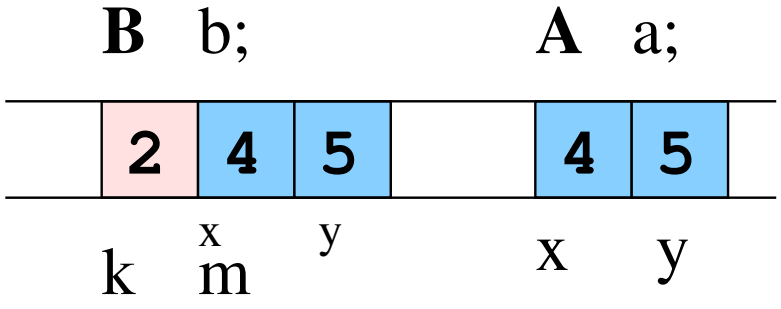
```
1  #include  <unistd.h> // for close(int)
2  class File {
3    public:
4      File(const char* filename);
5      ~File() { // destructor; why no parameters?
6        close(fd_); // close file descriptor
7      }
8      ..
9    private:
10     // file descriptor corresponding to open file
11     int  fd_;
12     ..
13 };
```

destructors example

```
1  #include  <unistd.h> // for close(int)
2
3  class File {
4    public:
5      File(const char* filename);
6      ~File() { // destructor; why no parameters?
7        close(fd_); // close file descriptor
8      }
9      ..
10   private:
11     // file descriptor corresponding to open file
12     int  fd_;
13     ..
14 };
15
16 void
17 process_file(const char* name) {
18   File f(name);
19   .. // f automatically destroyed
20 }
```

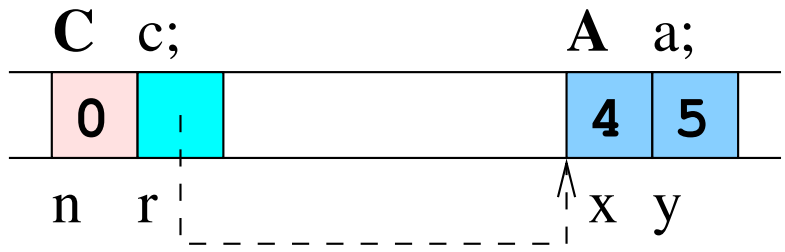
member objects

```
1  class A { // ...
2    public:
3      A(int i, int j): x(i), y(j) {}
4    private:
5      int x;
6      int y;
7  };
8
9  class B { // ...
10   public:
11     B(int i, A& a): k(i), m(a) {}
12   private:
13     int   k;
14     A     m; // a member object
15 };
16
17 A a(4, 5);
18 B b(2, a); // what's happening?
```



member references

```
1  class A {
2    public:
3      A(int i, int j): x(i), y(j) {}
4    private:
5      int x;
6      int y;
7  };
8
9  class C {
10   public:
11     C(A& a, int i): r(a), n(i) {}
12     // ...
13   private:
14     int n;
15     A& r; // not a member object, *must* be initialized
16 };
17
18 A a(4, 5);
19 C c(a, 0); // what's happening?
```



the life of a server class object

```
1  #include <fstream>
2
3  class Server {
4    public:
5      Server(const char* logfilename, int port):
6        log_(logfilename), port_(port) {
7          // set up server
8          log_ << "server started\n"; // why does this work/
9      }
10     ~Server() { // close down server
11       log_ << "server quitting\n"; // why does this work?
12     }
13     void serve() {  // handle requests
14     }
15     // lots of stuff omitted
16   private:
17     Server(const Server&);
18     Server& operator=(const Server&);
19     std::ofstream  log_;
20     int port_;
21 };
```

the life of a class object

1. (allocate memory)
2. Construction using a (possibly default) constructor function:
   - 2.1 Construct member objects in the order of their declaration (which should match the order in the initialization list of the constructor).
   - 2.2 Execute the body of the constructor.
3. Provide services via member function calls, or as parameter to ordinary functions.
4. Destruction:
   - 4.1 Execute code of destructor body, if there is a destructor.
   - 4.2 Destroy member objects.
5. (deallocate memory)

friends

```
1  class Rational {
2    public:
3      Rational(int num=0, int denom=1);
4      Rational multiply(Rational r);
5      Rational add(Rational r);
6      // non-member function operator<< has
7      // access to private members of Rational
8      friend std::ostream& operator<<(std::ostream&, Rational);
9    private:
10     int num_;
11     int denom_; // must not be 0!
12 };
13 // definition of operator<<
14 inline std::ostream& operator<<(std::ostream& os, Rational r)
15   return os << r.num_ << "/" << r.denom_;
16 }
17
18 Rational r(2, 3);
19 std::cout << r; // what happens?
```

more friends

```
1  class Node {
2    friend class IntStack; // everything is private
3    // but IntStack is a friend so only IntStack
4    // can use Node objects
5    private: // default, so this line could be dropped
6      Node(int, Node* next=0);
7      ~Node();
8      Node*  next() { return next_; }
9      int    item;
10     Node* next_;
11 };
12
13 class IntStack { // stack of int
14   public:
15     IntStack();
16     ~IntStack();
17     IntStack& push(int);
18     int  top();
19     bool  empty();
20   private:
21     Node*  top_; // pointer to topmost node
22 };
```

nested classes

```
1  class IntStack { // stack of integers
2    public:
3      IntStack();
4      ~IntStack();
5      IntStack& push(int);
6      // ..
7    private:
8      class Node { // why is this solution better?
9        public:
10         Node(int, Node* next=0);
11         ~Node();
12         Node* next();
13       private:
14         int item;
15         Node* next_;
16     };
17   Node*  top_; // pointer to topmost node
18 };
19
20 inline Node* // also illustrates scope (::) operator
21 IntStack::Node::next() { return next_; }
```

static members: declaration (point.h)

```
1  #ifndef POINT_H
2  #define POINT_H
3  class Point {
4  public:
5    Point(int X,int Y): x(X), y(Y) { ++count; }
6    Point(const Point&p): x(p.x), y(p.y) { ++count; }
7    ~Point() { --count; }
8    // declaration (and definition) of a static member function
9    static int get_count() { return count; }
10   ...
11 private:
12   // declaration of a static data member
13   static int count;
14   int x; // x-coordinate of point
15   int y; // y-coordinate of point
16 };
17 #endif
```

static members: definition (point.C)

point.C should contain definition of static member:

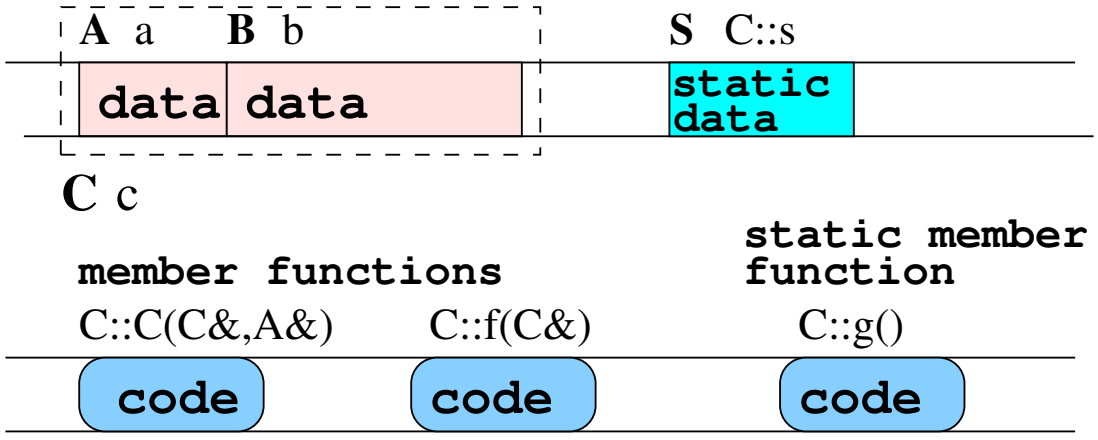
```
1  #include "point.h"
2
3  // definition of static data member
4  int Point::count(0);
```

Example

```
1  #include "point.h"
2
3  Point p1(1, 2);
4  {
5    Point p1(p);
6    p.get_count(); // print 2
7  }
8  // no target needed:
9  Point::get_count();  // print 1
```

implementing classes

```
1  class C {
2    public:
3      C(A& a);
4      A f();
5      static S g();
6    private:
7      A a;
8      B b;
9      static S s;
10 };
```



- 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

```
1  class File {
2    public:
3      // defines 4 names in scope File
4      enum Mode { READ, WRITE, APPEND};
5      File(const char* filename, Mode mode=READ);
6      ~File();
7      Mode mode() { return mode_ ; }
8    private:
9      Mode  mode_;
10     // ...
11 };
```

`enum NameOfType { EnumeratorList };`

```
1  File f("book.tex");
2  if (f.mode()==File::WRITE) {
3    // ...
4  }
```

overriding enumerated type values

```
1  class Http {
2    public:
3      enum Operation { GET, HEAD, PUT };
4      enum Status { OK = 200, CREATED = 201, ACCEPTED = 202,
5        PARTIAL = 203, MOVED = 301, FOUND = 302, METHOD = 303,
6        NO_CHANGE = 304, BAD_REQUEST = 400,
7        UNAUTHORIZED = 401, PAYMENT_REQUIRED = 402,
8        FORBIDDEN = 403, NOT_FOUND = 404,
9        INTERNAL_ERROR = 500, NOT_IMPLEMENTED = 501
10     };
11 ..
12 };
```

overriding enumerated type values

```
1  class Http {
2    public:
3      enum Operation { GET, HEAD, PUT };
4      enum Status { ...  };
5  ..
6  };
7
8  std::ostream&
9  operator<<(std::ostream& os, Http::Status status) {
10   switch (status) {
11     case Http::OK: os << "OK"; break;
12     case Http::CREATED: os << "CREATED"; break;
13     case Http::ACCEPTED: os << "ACCEPTED"; break;
14     ..
15   }
16   return os;
17 }
```

typedef

`typedef Declaration;`

Defines short name for (complex) type expression.

```
1  typedef unsigned int uint;
2  uint    x; // equivalent with unsigned int x;
3
4  typedef Sql::Command::iterator  IT;
5
6  int square(int x) { return x * x; }
7  // also for function types:
8  typedef int UnaryFunction(int);
9  // UnaryFunction is type int -> int
10 // f is pointer to function (see later), it is
11 // initialized to (point to) the 'square' function
12 UnaryFunction*  f(square);
13 f(2); // same as square(2)
```

Outline

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 through this variable.

```
1  const int x(4);
2  x = 5; // error
3
4  int   y(6);
5  const int& z(y);
6  z = 7; // error
7  y = 5; // ok, explain
8
9  const int u(7);
10 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 than call-by-value

```
1  class Rational {
2    public:
3      Rational(int num, int denom): num_(num), denom_(denom) {
4        if (denom==0)
5          abort();
6      }
7      Rational multiply(const Rational& r) {
8        return Rational(num_ * r.num_, denom_ * r.denom_);
9      }
10   private:
11     int num_;
12     int denom_; // must not be 0!
13 };
```

constant members: problem

```
1  class Rational {
2    public:
3      Rational(int num, int denom): num_(num), denom_(denom) {
4        // ..
5      }
6      Rational multiply(const Rational& r) {
7        return Rational(num_ * r.num_, denom_ * r.denom_);
8      }
9    private:
10     // silly,
11     // just illustrates that data members can be const too
12     const int num_;
13     const int denom_; // must not be 0!
14 };
15
16 Rational  r1(1, 2);
17 const Rational  r2(2, 3);
18 r2.multiply(r1); // compile error
```

constant members: solution

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

```
1  class Rational {
2    public:
3      Rational(int num, int denom): num_(num), denom_(denom) {
4        // ..
5      }
6      Rational multiply(const Rational& r) const {
7        return Rational(num_ * r.num_, denom_ * r.denom_);
8      }
9    private:
10     const int num_;
11     const int denom_; // must not be 0!
12 };
13
14 Rational  r1(1, 2);
15 const Rational  r2(2, 3);
16 r2.multiply(r1); // ok
```

overloading and const

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

```
1  int
2  f(const int& i) { return i; }
3  int
4  f(int& i) { return ++i; }
5
6  int
7  main() {
8    const int c(5);
9    int d(5);
10   // calls f(const int&); prints 5
11   std::cout << f(c) << std::endl;
12   // calls f(int&); prints 6
13   std::cout << f(d) << std::endl;
14 }
```

conversion non-const to const

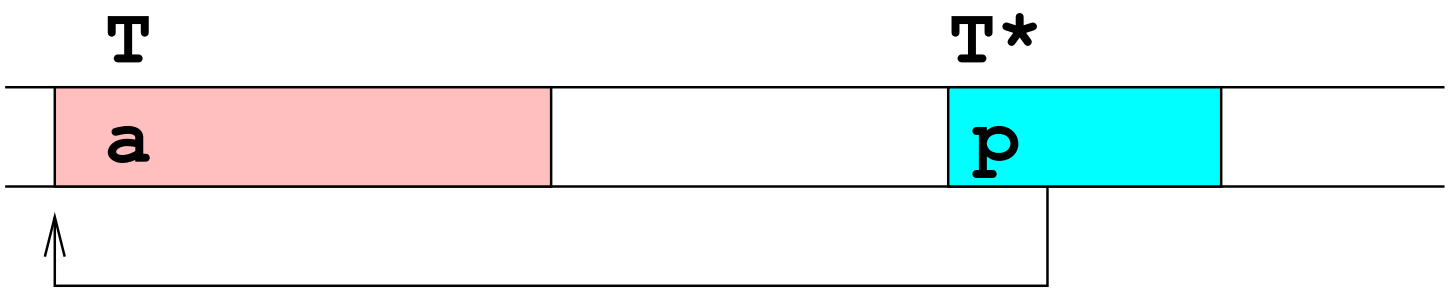
```
1  int
2  f(const int& i) { return i; }
3
4  int
5  main() {
6    int d(5);
7    // calls f(const int&);
8    // after 'converting' int& to const int&
9    std::cout << f(d) << std::endl;
10 }
```

pointers

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

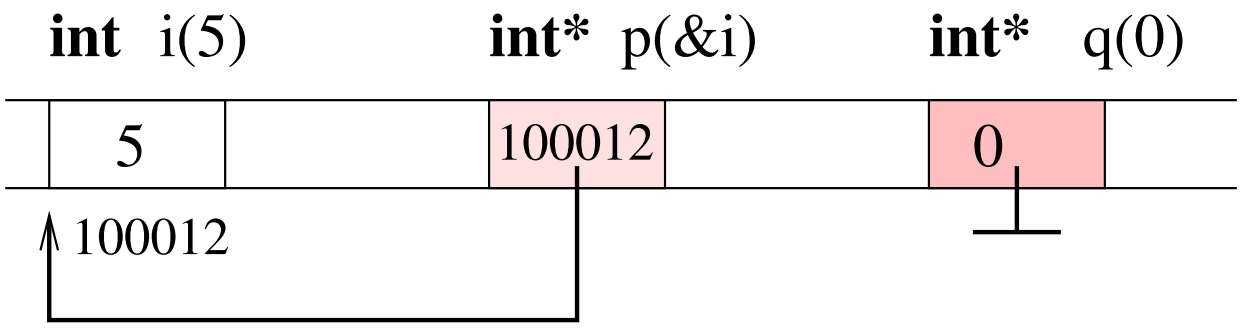
NameOfType$*$ is the type of "*addresses referring to objects of type* NameOfType".

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

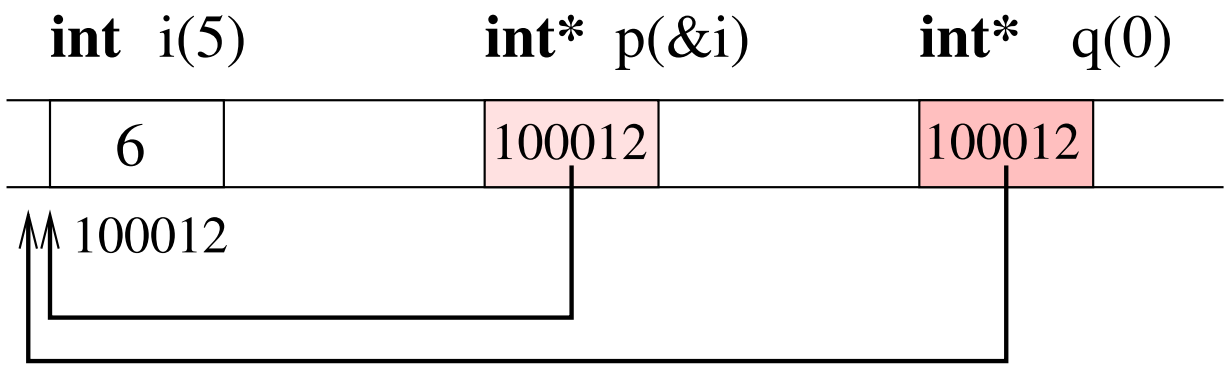


pointers example

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

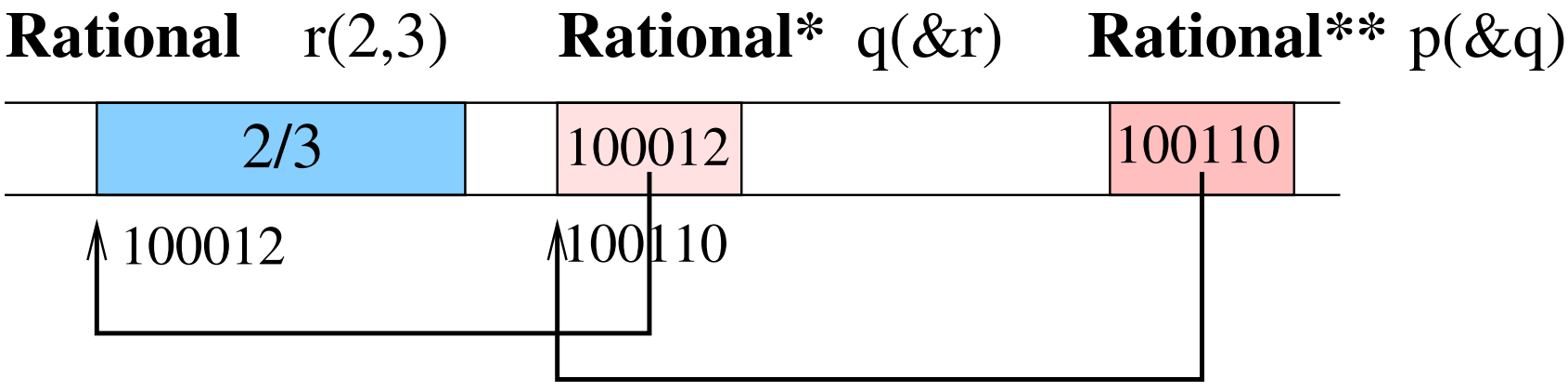



```
1  q = p;
2  *p = 6;
```



handles (pointers to pointers)

```
1  Rational     r(2, 3);
2  Rational*    q(&r);
3  Rational**   p(&q);
4
5  std::cout << **p + *q << "," << r << "," << *q << "," << **p;
6  std::cout << q->add(*q); // short for (*q).add(*q)
```




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

pointers as alternative for references

```
1  #include          <iostream>
2
3  void
4  swap_p(int* px, int* py) {
5    int tmp(*px);
6    // copy contents of what py points to
7    // to area that px points to
8    *px = *py;
9    *py = tmp;
10 }
11
12 void
13 swap_r(int& x, int& y) {
14   int tmp(x);
15   // copy contents of what y refers to
16   // to area that x refers to
17   x = y;
18   y = tmp;
19 }
```

pointers as alternative for references

```
1  int
2  main() {
3    int a(5);
4    int b(6);
5
6    swap_p(&a, &b); // pass pointers to a, b
7    std::cout << a << ", " << b << std::endl; // prints 6, 5
8
9    swap_r(a, b);
10   std::cout << a << ", " << b << std::endl; // prints 5, 6
11 }
```

What are the (dis)advantages of using reference, resp. pointer arguments?

pointers and const

- forbidding modification of an object “through a pointer”

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

- forbidding modification of the pointer itself

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

- forbidding both

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

pointers vs references

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

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

```
1  *p = 5; r = 5; // same effect
2
3  int j;
4  p = &j; // ERROR, it is also impossible to make r refer to j
```

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

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

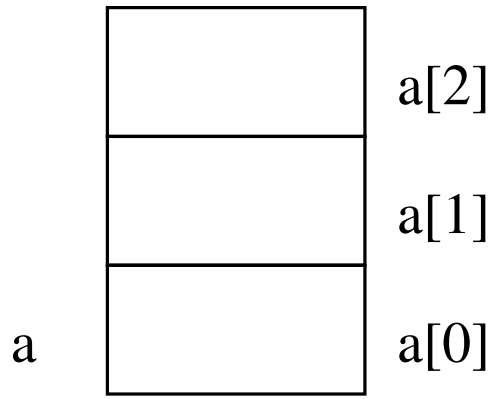
the this pointer

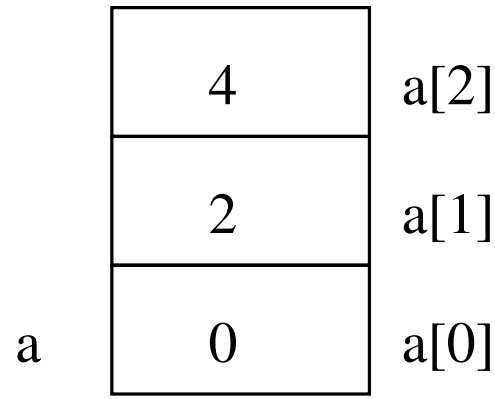
```
class T {
  ReturnType f(ParameterList) {
    T* const this(PtrToTargetObject);
    // ...
    }
  ReturnType f(ParameterList) const {
    const T* const this(PtrToTargetObject);
    // ...
    }
}
```

the this pointer: example

```
1  // should return reference to target object
2  // in order to support x = y = z;
3  Rational&
4  Rational::operator=(const Rational& r) {
5    num_ = r.num();
6    denom_ = r.denom();
7    simplify();
8    return *this; // return reference to target object
9  }
```

arrays






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

example: bubble sort an array: swap

```
1  #include <iostream>
2  #include <string>
3
4  void
5  swap(std::string& x, std::string& y) {
6    std::string tmp(x);
7    x = y;
8    y = tmp;
9  }
```

example: bubble sort an array: sort

```
1  const int  MAX_WORDS = 10;
2  std::string  words[MAX_WORDS];
3
4  int
5  main() { // read 10 strings from stdin and bubble-sort them
6    for (unsigned int i=0; (i<MAX_WORDS); ++i)
7      std::cin >> words[i];
8    for (unsigned int size=MAX_WORDS-1;(size>0);--size)
9      // find largest element in 0..size range and
10     // store it at index size
11     for (unsigned int i=0;(i<size);++i)
12       if (words[i+1]<words[i])
13         swap(words[i+1], words[i]);
14   for (unsigned int i=0; (i<MAX_WORDS); ++i)
15     std::cout << words[i] << " ";
16 }
```

array initialization

```
1  #include        <iostream>
2
3  // compiler can figure out how large the array should be
4  float   vat_rates[] = { 0, 6, 20.5 };
5
6  int
7  main() {
8    // how to find the number of elements in vat_rates?
9    unsigned int size(sizeof(vat_rates)/sizeof(float));
10   const char message[] = "VAT rates"; // special case
11   std::cout << message;
12   for (unsigned int i=0;(i<size);++i)
13     std::cout << " " << vat_rates[i];
14   std::cout << std::endl;
15 }
```

with default constructor

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

```
1  class Rational {
2    public:
3      Rational(int num=0, int denom=1):
4        num_(num), denom_(denom) {}
5      // ...
6    private:
7      int num_;
8      int denom_;
9  };
10
11 // calls Rational::Rational() on each element
12 Rational rationals[3];
13 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!)

```
1  int
2  sum(int a[], unsigned int size) {
3    int total(0);
4    for (unsigned int i=0; i<size ;++i)
5      total += a[i];
6    return total;
7  }
8
9  int
10 main() {
11   int numbers[] = { 1 , 2, 3, 4, 5 };
12   std::cout
13     << sum(numbers, sizeof(numbers)/sizeof(int))
14     << std::endl;
15 }
```

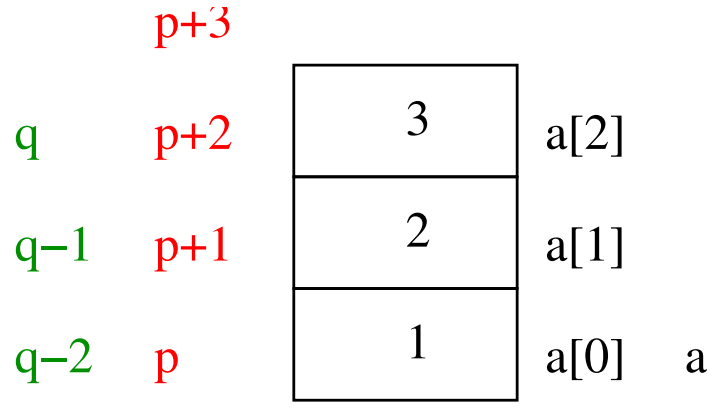
arrays vs pointers

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

```
1  #include <iostream>
2
3  void
4  f(int x[]) { // can take array or pointer parameter
5    x[0] = 1;
6  }
7
8  int
9  main() {
10   int a[] = { 0, 2, 3 };
11   int* p(a);
12   int* q(&a[0]); // exactly the same
13   f(p); // passing a pointer or an array is the same
14   std::cout << *p << ", " << a[0] << std::endl; // prints 1,1
15   // prints 1, 1\n 2, 2\n 3, 3
16   for (unsigned int i=0;(i<sizeof(a)/sizeof(int));++i)
17     std::cout << p[i] << ", " <<  a[i] << std::endl;
18 }
```

pointer arithmetic

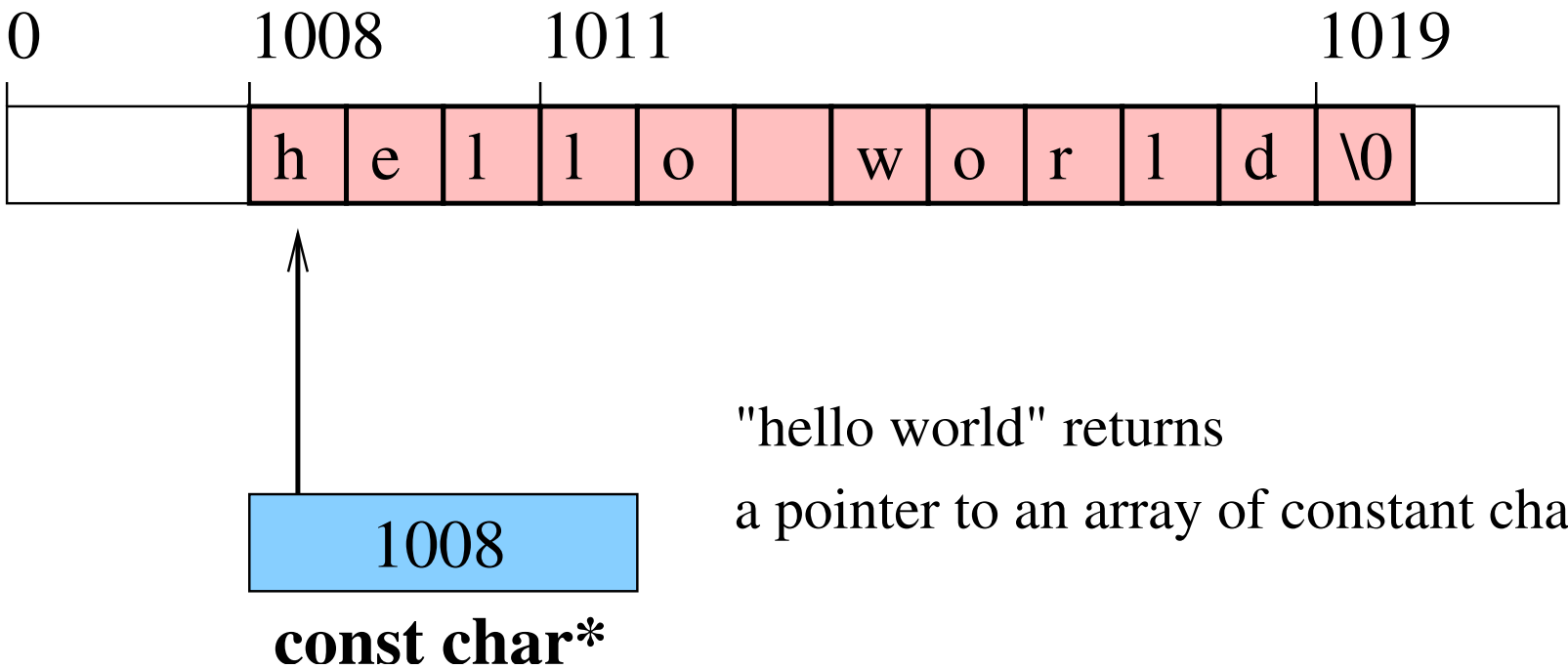
Pointers can be assigned, integers can be added (subtracted) to (from) pointers.




```
1  int a[] = { 1, 2, 3 };
2
3  int
4  main() {
5    int* p(a);
6    int* q(p+2);
7    int* s(q-2);
8    for (unsigned int i=0;(i<3);++i)
9      std::cout << *p++ << "\n"; // what happens?
10   std::cout << q - p << std::endl; // prints -1
11 }
```

C-strings

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



C-strings example

```
1  #include <iostream>
2
3  void
4  print(std::ostream& os, const char*p) {
5    while (*p)
6      os << *p++;
7    os << std::endl;
8  }
9
10 int
11 main() {
12   const char* s("hello world");
13   print(std::cout, s);
14 }
```

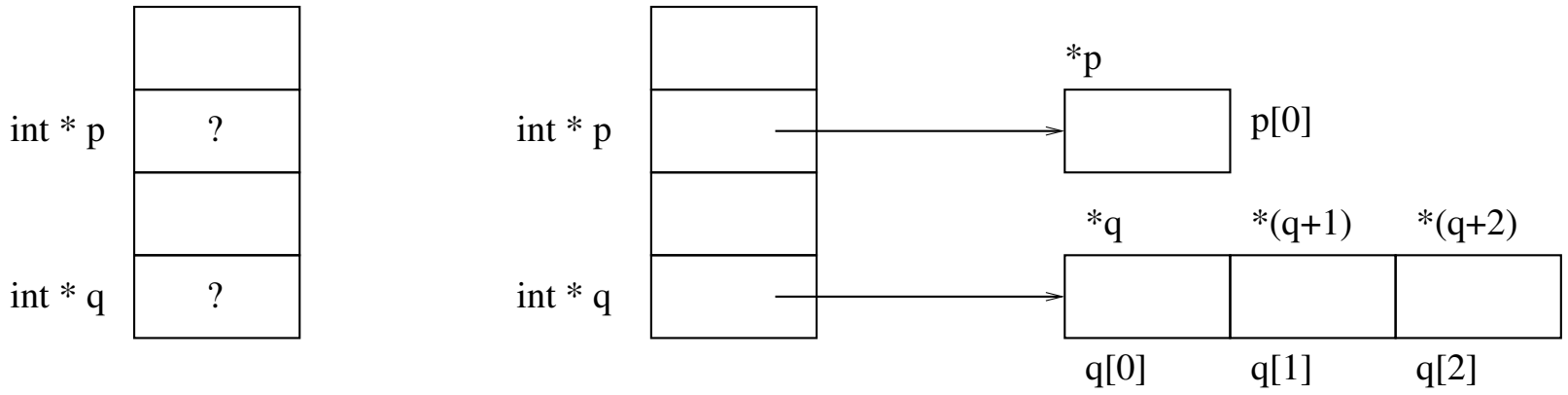
comparing C-strings

```
1  #include <iostream>
2  // returns
3  //      0       if s1 and s2 are (lexicographically) equal
4  //      >0      if s1 is lexicographically larger than s2
5  //      <0      if s1 is lexicographically smaller than s2
6  int
7  strcmp(const char*s1, const char* s2) {
8    while ((*s1) && (*s2) && (*s1==*s2)) {
9      ++s1; ++s2;
10   }
11   return *s1 - *s2;
12 }
13
14 int
15 main() {
16   const char*  s1("abc");
17   const char*  s2("abcde");
18   cout << strcmp(s1, s2) << endl; // prints -100
19 }
```

command line processing

```
1  #include  <iostream>
2  #include  <stdlib.h> // for atoi()
3
4  // this program computes the sum of its command line arguments
5  // usage: sum int..
6
7  int
8  main(unsigned int argc, char* argv[]) {
9    // - argv is an array of pointers to (arrays of) char,
10   //   one for each argument
11   // - argv[0] is the name of the program, i.e.
12   //   the first word in the command line
13   // - argc is the number of arguments
14   int sum(0);
15
16   // atoi(const char*) converts a string to an int
17   for (unsigned int i=1; (i<argc); ++i)
18     sum += atoi(argv[i]);
19
20   std::cout << sum << std::endl;
21 }
```

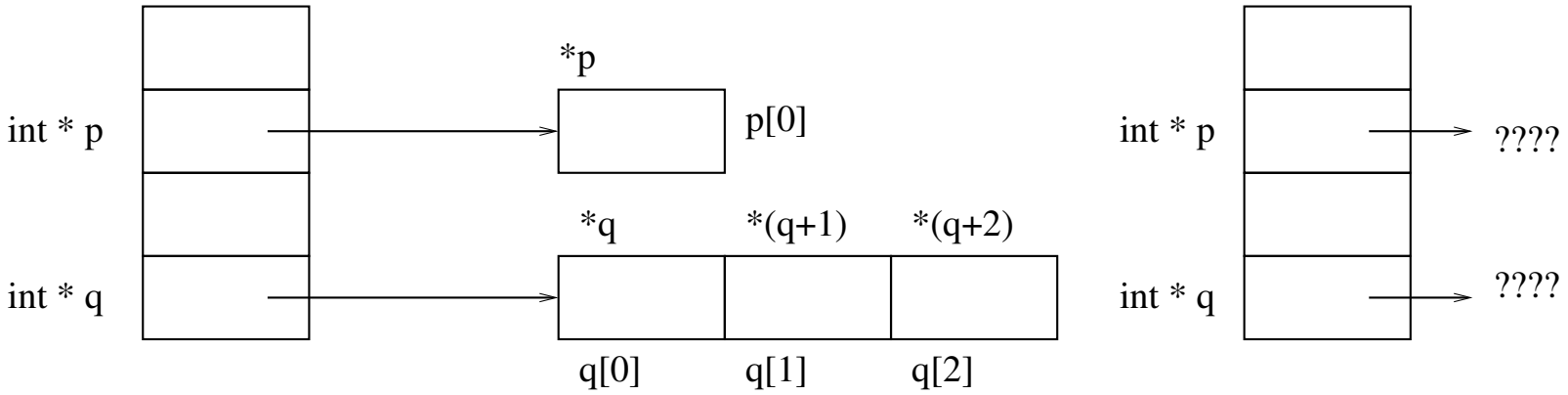
explicit memory allocation




```
1  int*  p; // not initialized
2  int*  q; // not initialized
3
4  p = new int; // allocate memory for 1 new integer
5  q = new int[3]; // allocate memory for 3 new integers
```

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

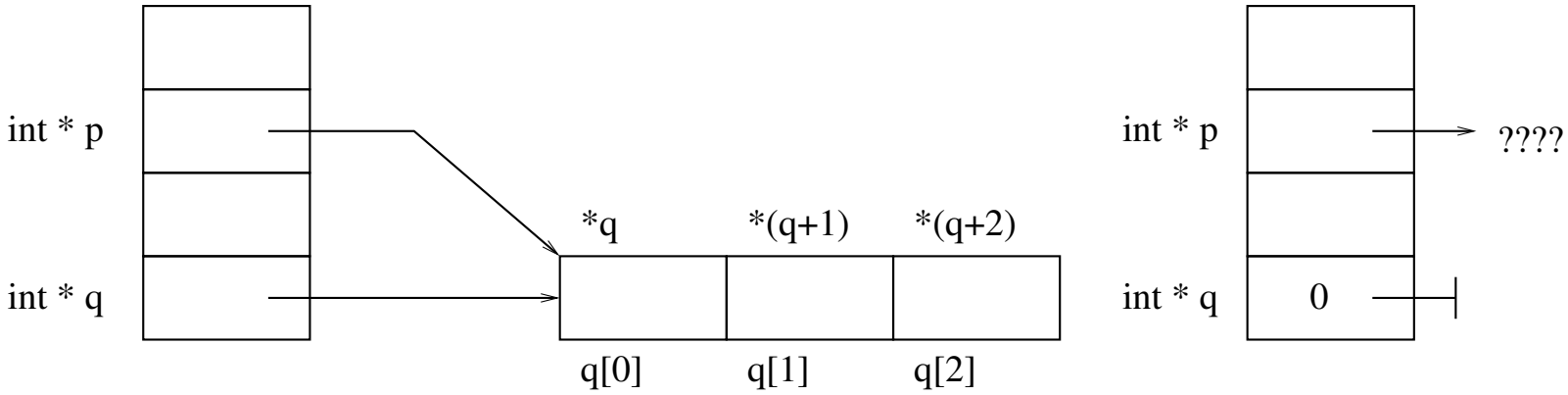
explicit memory deallocation




```
1  int*  p; // not initialized
2  int*  q; // not initialized
3  p = new int; // allocate memory for 1 new integer
4  q = new int[3]; // allocate memory for 3 new integers
5  delete p; // deallocate memory that was allocated using new
6  // deallocate memory that was allocated using new []
7  delete [] q;
```

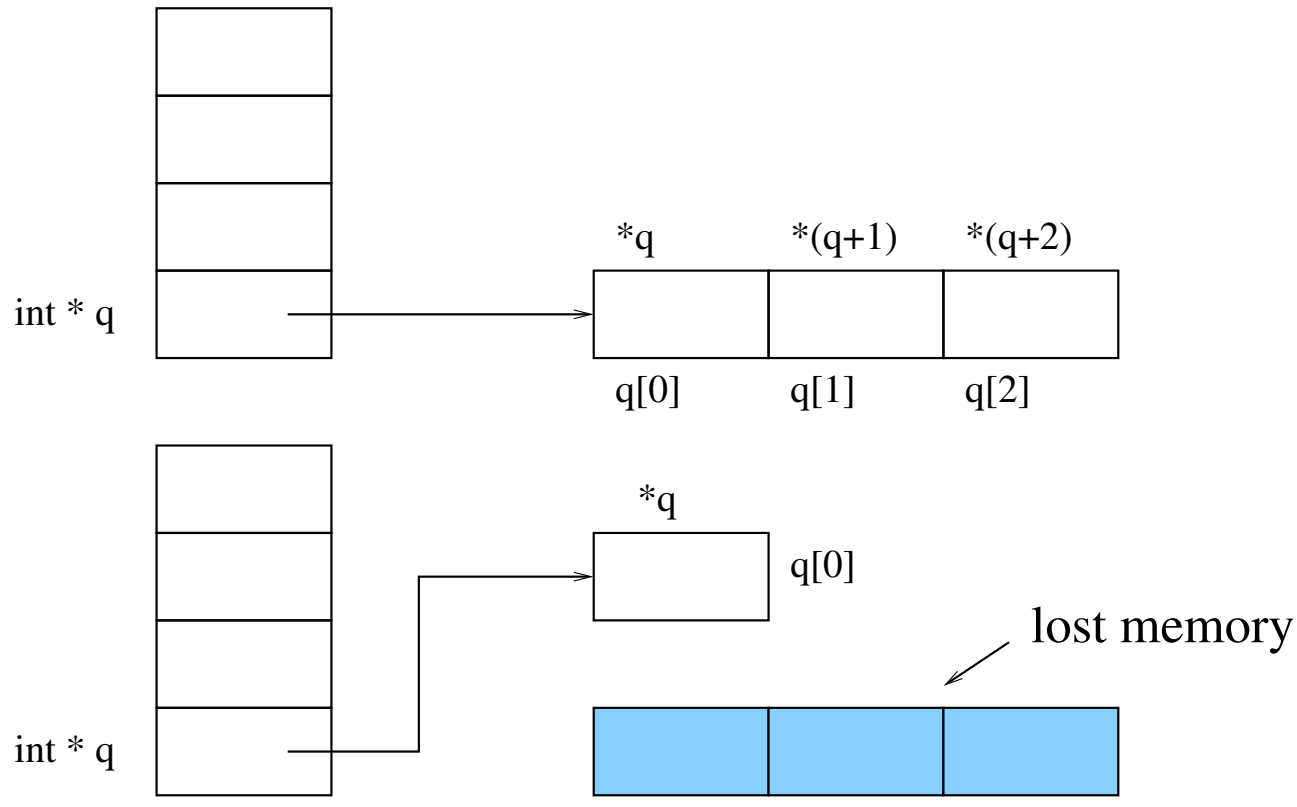
After delete pointers are left "**dangling**"

the dangling pointer syndrome




```
1  int*     q(new int[3]);
2  int*     p(q);
3  delete [] q;
4  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	just before owner	just after owner
free	using new/delete	determined by programmer	determined by programmer

example: a Ztring class

```
1  #ifndef ZTRING_H
2  #define ZTRING_H
3  #include <iostream>
4
5  class Ztring {
6    public:
7      Ztring(const char* cstring=0);
8      Ztring(const Ztring&); // copy constructor
9      ~Ztring(); // destructor
10
11     Ztring& operator=(const Ztring&);
12
13     const char* data() const; // why return const char*?
14     unsigned int size() const;
15     void print(std::ostream&) const;
16     void concat(const Ztring&); // concatenates to *this
17   private:
18     char* data_; // a C string
19     char* init(const char* string); // return a copy of string
20 };
```

Ztring overloaded operators

```
1  // Auxiliary functions: overloaded operators
2  // E.g.
3  // Ztring x("abc");
4  // Ztring y("def");
5  // cout << x + y << endl;
6
7  Ztring
8  operator+(const Ztring& s1, const Ztring& s2);
9
10 std::ostream&
11 operator<<(std::ostream& os, const Ztring& s);
12
13 #endif
```

Ztring constructors and destructor

```
1  #include "ztring.h"
2  #include <string.h> // for strlen
3
4  // constructors and destructor
5
6  Ztring::Ztring(const char* cstring): data_(init(cstring)) {
7  }
8
9  Ztring::Ztring(const Ztring& s): data_(init(s.data())) {
10 }
11
12 Ztring::~Ztring() {
13   // very important: avoid memory leak
14   delete [] data_;
15 }
```

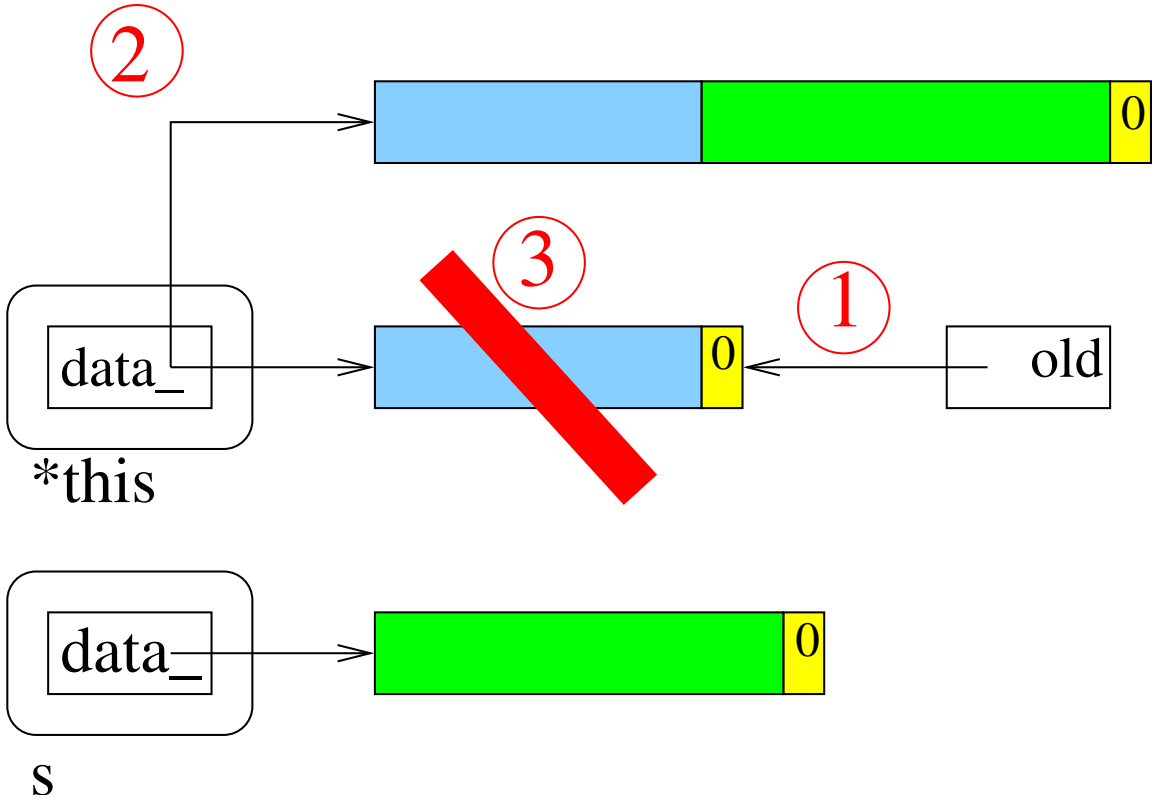
Ztring assignment operator

```
1  // assignment operator
2  Ztring&
3  Ztring::operator=(const Ztring& s) {
4    if (this==&s) // why?
5      return *this;
6    delete [] data_; // avoid memory leak
7    data_ = init(s.data());
8    return *this;
9  }
```

Ztring inspectors

```
1  // constant public member functions
2  const char*
3  Ztring::data() const {
4    return data_;
5  }
6
7  unsigned int
8  Ztring::size() const {
9    if (data_==0)
10     return 0;
11   return strlen(data_);
12 }
13
14 void
15 Ztring::print(std::ostream& os) const {
16   // can be made shorter; how?
17   for (unsigned int i=0; (i<size()); ++i)
18     os << data_[i];
19 }
```

Ztring concatenation



```
1  void
2  Ztring::concat(const Ztring& s) {
3    // save old data of this ztring
4    unsigned int  old_size(size());
5    char*  old(data_);
6    // allocate buffer large enough to hold both + trailing '\0'
7    data_ = new char[old_size+s.size()+1];
8
9    unsigned int j(0);
10   for (unsigned int i=0; (i<old_size); ++i)
11     data_[j++] = old[i]; // copy original string
12   for (unsigned int i=0; (i<s.size()); ++i)
13     data_[j++] = s.data_[i]; // after that the argument string
14   data_[j] = '\0'; // don't forget trailing '\0' character
15
16   delete [] old; // avoid memory leak
17 }
```

Ztring private member functions

```
1  // private member functions
2  //   see also: strdup()
3  char*
4  Ztring::init(const char* s) {
5    if (s==0)
6      return 0;
7    else {
8      unsigned int len(strlen(s)+1); // why +1?
9      char* p(new char[len]);
10     for (unsigned int i=0; (i<len); ++i)
11       p[i] = s[i];
12     return p;
13   }
14 }
```

Ztring auxiliary functions

```
1  // auxiliary functions (overloaded operators)
2
3  Ztring
4  operator+(const Ztring& s1, const Ztring& s2) {
5    Ztring  s(s1);
6    s.concat(s2);
7    return s;
8  }
9
10 std::ostream&
11 operator<<(std::ostream& os, const Ztring& s) {
12   s.print(os);
13   return os;
14 }
```

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).

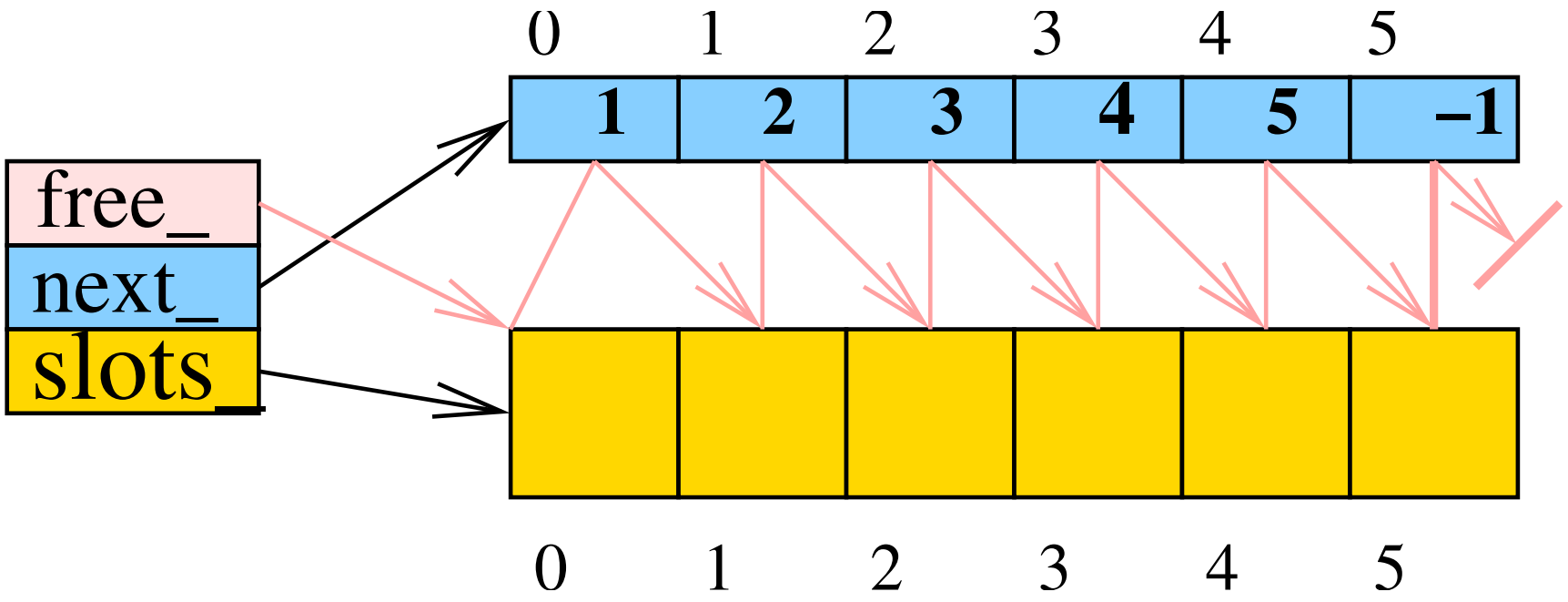
overloading new, delete

```
1  class Rational {
2    // ..
3    public:
4      void* operator new(size_t) { return pool_.alloc(); }
5      void operator delete(void* p, size_t size) {
6        assert(size==sizeof(Rational)); // sanity check
7        if (p) // do not attempt to delete a null pointer
8          pool_.dealloc(p);
9      }
10     // ...
11   private:
12     static Pool pool_;
13     // ..
14 };
15
16 // will allocate from Rational::pool_
17 Rational*  p(new Rational(1, 3));
```

Pool for allocating free Rational objects

```
1  #ifndef POOL_H
2  #define POOL_H
3  #include "rational.h"
4  // a pool of reusable areas, each of size sizeof(Rational)
5  class Pool {
6    public:
7      Pool(unsigned int number_of_areas);
8      ~Pool();
9      bool is_full() const { return free_<0; }
10     Rational*  alloc(); // return pointer to free area
11     void  dealloc(void* p); // free an area
12   private:
13     Pool(const Pool&); // forbidden
14     Pool& operator=(const Pool&); // forbidden
15     Rational*  slots_;
16     // if i is the index of a free slot, then next_[i] is the
17     // index of another free slot or -1
18     int* next_;
19     // index of first free slot, <0 if the pool is full
20     int free_;
21 };
22 #endif
```

pool after construction



pool constructor, destructor

```
1  #include <stdlib.h> // for abort()
2
3  // note: Rational::operator new[] is not overloaded
4  Pool::Pool(unsigned int size): slots_(new Rational[size]),
5      next_(new int[size]), free_(0) {
6    // initially, the free list is 0, 1, 2, ..
7    for (unsigned int i=0;(i<(size-1));++i)
8      next_[i] = i+1;
9    next_[size-1] = -1; // end of free list
10 }
11
12 // note: Rational::operator delete[] is not overloaded
13 Pool::~Pool() {
14   delete [] next_;
15   delete [] slots_;
16 }
```

(de)allocation from a Pool

```
1  Rational*
2  Pool::alloc() {
3    if (is_full())
4      abort();
5    Rational* r(&slots_[free_]); // address of first free area
6    free_ = next_[free_]; // update start of free list
7    return r;
8  }
9
10 void
11 Pool::dealloc(void* p) { // deallocate a Rational area
12   // compute index of pointer p in slots_ array
13
14   // static_cast converts ``related'' types,
15   // e.g. here from void* to Rational*
16   int index(static_cast<Rational*>(p)-slots_);
17   // add index of deallocated area to front of free list
18   next_[index] = free_;
19   free_ = index;
20 }
```

smart pointers

```
1  class Url; // defined elsewhere
2
3  class HtmlPage { // a page is uniquely identified by its URL
4    friend class Proxy;
5    public:
6      std::string title() const;
7    private:
8      // retrieve the page corresponding to a Url
9      static HtmlPage* fetch(const Url&);
10     HtmlPage(const HtmlPage&); // forbidden
11     HtmlPage& operator=(const HtmlPage&); // forbidden
12 };
```

smart pointers

```
1  class Proxy {
2    public:
3      Proxy(const std::string& url): key_(url) {}
4      // smart pointer
5      HtmlPage* operator->() const {
6        return HtmlPage::fetch(url());
7      }
8      const Url&  url() const { return key_; }
9    private:
10     Url  key_;
11 };
12
13 Proxy proxy("http://tinf2.vub.ac.be/index.html");
14 proxy->title(); // calls HtmlPage::fetch("http://tinf2..");
```

Outline

why templates

Suppose we have written a function

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

and now we need a function

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

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

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

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

```
std::string  sa[]; sort(sa); // should work
int  ia[]; sort(ia); // should work
```

template functions

```
1  #ifndef BUBBLE_SORT_H
2  #define BUBBLE_SORT_H
3  #include <iostream>
4  #include <string>
5
6  template<typename T>
7  void
8  swap(T& x, T& y) {
9    T tmp(x);
10   x = y;
11   y = tmp;
12 }
```

template functions

```
13  template <typename T>
14  void
15  bubble_sort(T a[], unsigned int total_size) {
16    for (unsigned int size=total_size-1;(size>0);--size)
17      // find largest element in 0..size range
18      // and store it at size
19      for (unsigned int i=0;(i<size);++i)
20        if (a[i+1]<a[i])
21          swap(a[i+1], a[i]);
22  }
23  #endif
```

What are the (hidden) requirements on T?

instantiate template function for std::string

```
23  #include "bubble_sort.h"
24
25  int
26  main(unsigned int argc, char* argv[]) {
27    unsigned int  size = argc-1;
28    std::cerr << "size=" << size << std::endl;
29
30    std::string  *args = new std::string[size];
31    for (unsigned int i=0;(i<size);++i)
32      args[i] = std::string(argv[i+1]); // why i+1?
33
34    bubble_sort<std::string>(args, size);
35    bubble_sort(args, size); // will also work
36
37    for (unsigned int i=0;(i<size);++i)
38      std::cerr << "args[" << i << "] = "
39        << args[i] << std::endl;
40  }
```

instantiate template function for int

```
23  #include "bubble_sort.h"
24  #include  <stdlib.h> // for atoi()
25
26  int
27  main(unsigned int argc, char* argv[]) {
28    unsigned int  size = argc-1;
29    std::cerr << "size=" << size << std::endl;
30
31    // now args[] is an int array
32    int *args = new int[size];
33    for (unsigned int i=0;(i<size);++i)
34      args[i] = atoi(argv[i+1]); // why i+1?
35
36    // use same bubble_sort, now instantiated for int
37    bubble_sort<int>(args, size);
38    bubble_sort(args, size); // will also work
39
40    for (unsigned int i=0;(i<size);++i)
41      std::cerr << "args[" << i << "] = "
42        << args[i] << std::endl;
43  }
```

overloading template functions

```
1  template <typename T>
2  T // general case
3  maximum(T x1, T x2) { return (x1 > x2 ? x1 : x2 ); }
4
5  template <typename U>
6  U* // compare what is pointed to, not pointers
7  maximum(U* p1, U* p2) { return (*p1 > *p2 ? p1 : p2 ); }
8
9  const char* // special case for C strings
10 maximum(const char* s1, const char* s2) {
11   return (strcmp(s1, s2)>0 ? s1 : s2 );
12 }
13
14 double d1(1.23);
15 double d2(4.5);
16
17 maximum(d1, d2);
18 maximum(&d1, &d2);
19 maximum("abc", "abcd");
```

overloading and specialization

Essentially same rule as before (i.e. use “best match”):

- 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.

template classes

```
1  #ifndef ARRAY_H
2  #define ARRAY_H
3
4  #include <assert.h>
5
6  // safe array: subscripts are checked
7  template <typename T>
8  class Array {
9    public:
10     Array(unsigned int size);
11     Array(const Array&); // gang of three
12     ~Array();
13     unsigned int size() const;
14     // overloaded operator[], will check legality of index
15     T& operator[](unsigned int i); // why two versions?
16     const T& operator[] (unsigned int i) const;
17   private:
18     T* data_; // will point to C++ array of T
19     unsigned int size_; // size of data_ array
20     Array& operator=(const Array&); // we forbid assignment!
21 };
```

Array implementation: constructors

```
22  // constructors
23  template <typename T>
24  Array<T>::Array(unsigned int size):
25    data_(new T[size]), size_(size) {
26  }
27
28  template <typename T>
29  Array<T>::Array(const Array& a):
30      data_(new T[a.size()]), size_(a.size()) {
31    for (unsigned int i=0;(i<size_);++i)
32      data_[i] = a[i];
33  }
34
35  // destructor
36  template<typename T>
37  Array<T>::~Array() {
38    delete [] data_;
39  }
```

Array implementation: inspectors

```
40  // inspector functions
41
42  template <typename T>
43  unsigned int
44  Array<T>::size() const {
45    return size_;
46  }
47
48  template <typename T>
49  const T& // supports e.g. x = a[k]
50  Array<T>::operator[](unsigned int i) const {
51    assert(i<size_); // abort if bad index
52    return data_[i];
53  }
```

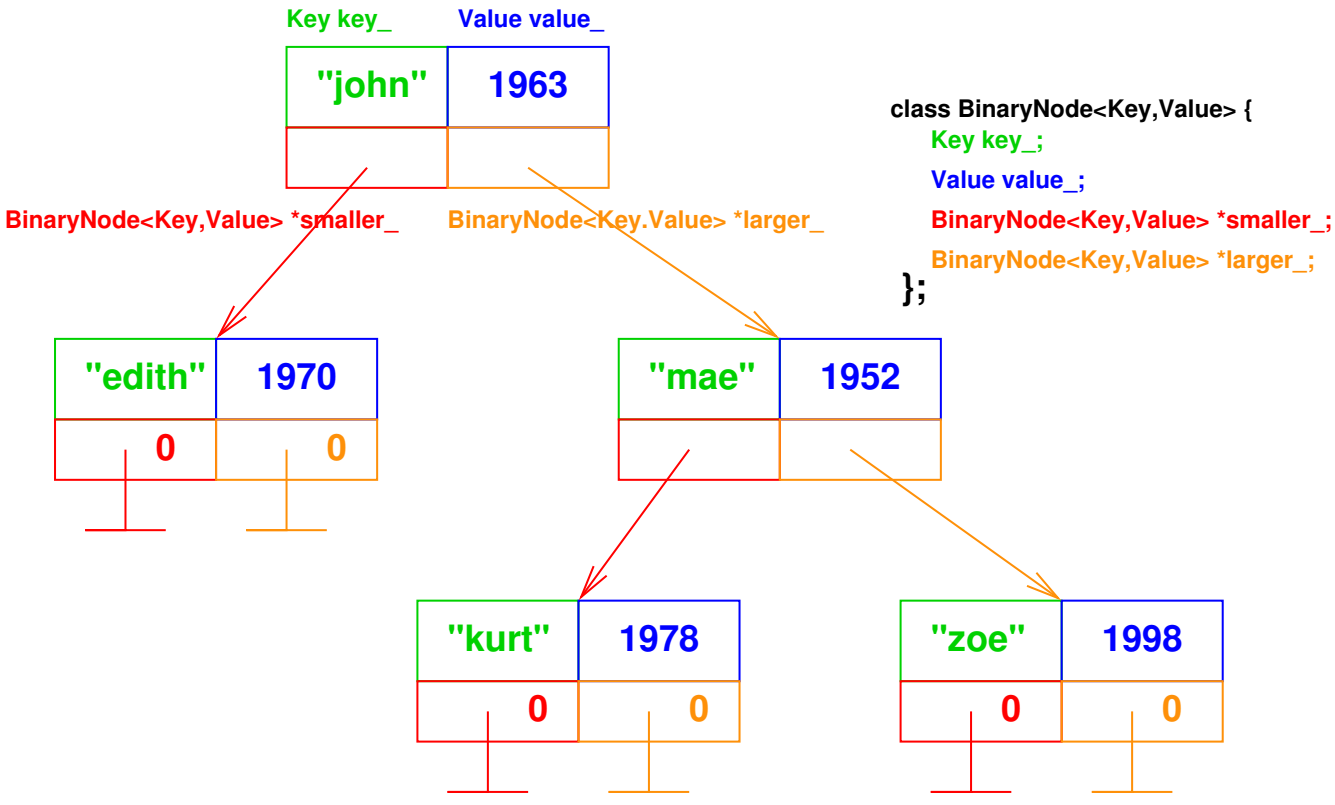
Array: indexing for assignment

```
54  // non-const operator[]: return non-const reference
55  // can be used as in a[k] = ..
56  template <typename T>
57  T&
58  Array<T>::operator[](unsigned int i) {
59    assert(i<size_);
60    return data_[i];
61  }
62  #endif
```

Array usage example

```
1  #include <string>
2  #include "array.h"
3
4  Array<std::string>
5  f(Array<std::string> a) { // to test copy-ctor
6    return a;
7  }
8
9  int
10 main(int argc, char *argv[]) {
11   unsigned int  size = argc-1;
12
13   Array<std::string>  a(size);
14
15   for (unsigned int i=0; i<size; ++i)
16     a[i] = std::string(argv[i+1]);
17
18   for (unsigned int i=0; i<size; ++i)
19     std::cout << f(a)[i] << std::endl;
20 }
```

binary search trees




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

the BinaryTree template class

```
1  #ifndef BINTREE_H
2  #define BINTREE_H
3
4  template <typename Key, typename Value>
5  class BinTree {
6    private: // Node represents a node in the binary
7      // search tree. There is one Node class for
8      // every instantiation of BinTree.
9      struct Node { // everything is public,
10       // but the whole class is private
11       Node(Key k, Value v, Node* sml=0, Node *lrg=0):
12         key_(k), val_(v), smaller_(sml), larger_(lrg) {}
13       Key  key_;
14       Value  val_;
15       Node*  smaller_;
16       Node*  larger_;
17     };
18
19   Node*  root_; // the root of the tree
```

```
22    public:
23      BinTree(): root_(0) {}
24      ~BinTree() { zap(root_); root_ = 0; }
25      // try to find node with key, if found, return
26      // true and copy its value to val parameter
27      bool  find(const Key& key, Value& val) const {
28        // find a node containing key
29        Node*  node = find_node(root_, key);
30        if (node) { // we found the node, get the value
31          val = node->val_;
32          return true;
33        }
34        else return false; // no such node, bad luck,
35      }
```

```
43      // insert inserts (key,val) in tree,
44      void  insert(const Key& key, const Value& val) {
45        insert_node(root_, key)->val_ = val;
46      }
47
48      // t[key] = val inserts, so operator[] returns
49      // a non-const reference
50      Value&  operator[](const Key& key) {
51        return insert_node(root_, key)->val_;
52      }
```

```
56    private:
57      BinTree(const BinTree&); // forbid copy constructor
58      BinTree&  operator=(const BinTree&); // forbid assignment
59
60      // zap(node) deallocates memory used by
61      // subtree starting at node
62      void   zap(Node* node) {
63        if (node==0)
64          return;
65        zap(node->smaller_);
66        zap(node->larger_);
67        delete node;
68      }
```

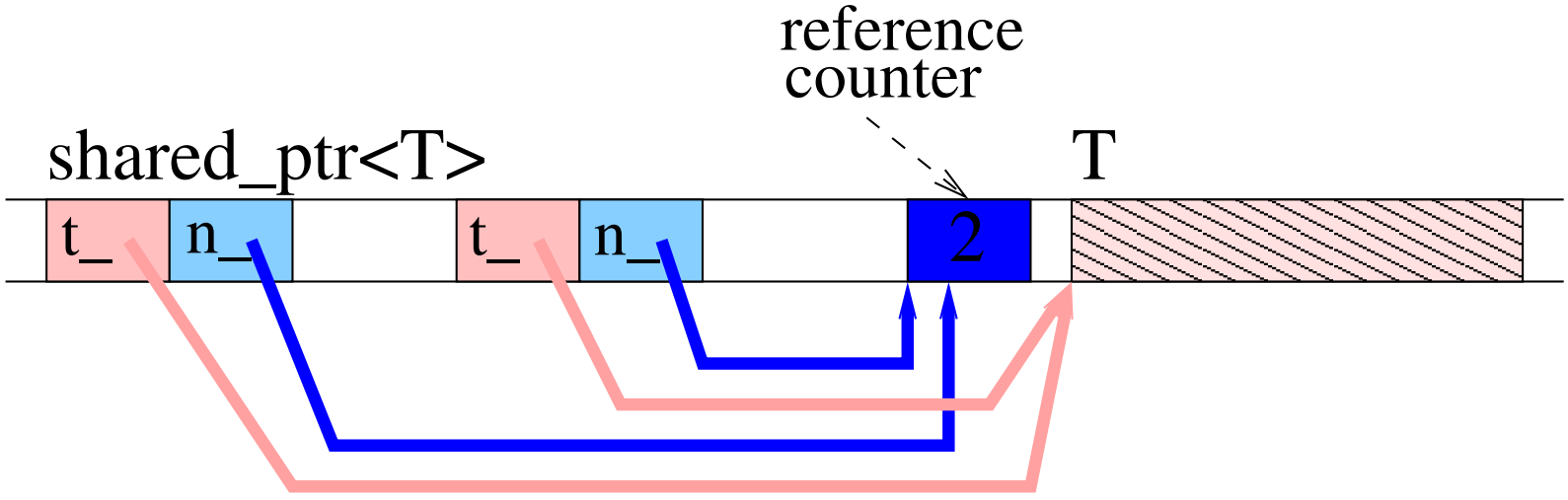
```
69      // find_node returns a pointer to a node containing key
70      // or 0, if such a node cannot be found
71      Node*  find_node(Node* node, const Key& key) const {
72        if (node==0)
73          return 0; // key could not be found
74        if (node->key_ == key)
75          return node; // got it
76        else
77          if (key < node->key_)
78            return find_node(node->smaller_, key);
79          else
80            return find_node(node->larger_, key);
81      }
```

```
69      // - insert_node returns the node where key should be
70      // - if key cannot be found, insert_node returns a
71      //   (pointer to a) fresh node where it should be
72      // - insert_node will also properly update the tree
73      //   if it needs to create a new node; this is why
74      //   the first parameter is a reference to a pointer
75      Node*  insert_node(Node*& node, const Key& key) {
76        if (node == 0)
77          // Don't forget: where there was a 0-pointer,
78          // there will now be pointer to a fresh node.
79          // Note default ctor for Value.
80          return (node = new Node(key, Value()));
81        if (key == node->key_)
82          return node;
83        if (key < node->key_)
84          return insert_node(node->smaller_, key);
85        else
86          return insert_node(node->larger_, key);
87      }
88  };
89  #endif
```

example program using BinaryTree

```
1  #include <string>
2  #include <iostream>
3  #include "bintree.h"
4
5  int
6  main(int argc, char* argv[]) {
7    BinTree<std::string, unsigned int> birth_year;
8
9    birth_year.insert("lisa", 1970);
10   birth_year.insert("john", 1936);
11   birth_year.insert("mae", 1952);
12   birth_year.insert("kurt", 1978);
13   // alternative way to insert
14   birth_year["zoe"] = 1998;
15   birth_year["john"] = 1963;
16
17   unsigned int  year(0);
18
19   if (birth_year.find("john", year))
20     std::cout << "birth year of john = " << year << std::endl;
21 }
```

reference-counted pointers



reference-counted pointers

```
1  #ifndef SHARED_PTR
2  #define SHARED_PTR
3  // a reference counting encapsulation of a pointer.
4  template<class T>
5  class shared_ptr {
6  public:
7    shared_ptr(T* t=0): t_(t), n_(new unsigned int(1)) {}
8    shared_ptr(const shared_ptr& r): t_(r.t_), n_(r.n_) {
9      ++*n_; // copy ctor increments shared counter *n_
10   }
11   ~shared_ptr();
12   operator bool() const { return (t_!=0); }
13   T& operator*() const { return *t_; }
14   T* operator->() const { return t_; }
15   shared_ptr& operator=(const shared_ptr& r);
16   // ...
17 private:
18   T* t_; // the encapsulated pointer
19   unsigned int* n_; // reference count for *t_
20 };
```

reference-counted pointers

```
21  template <typename T>
22  inline shared_ptr<T>&
23  shared_ptr<T>::operator=(const shared_ptr& r) {
24    if (this == &r) // check for self-assignment
25      return *this;
26    if (--*n_ == 0) {
27      // target is last pointer to *t_: delete it
28      // and the reference count
29      delete t_; delete n_;
30      }
31    // actual copy of parameter to target
32    t_ = r.t_;  n_=r.n_; ++*n_;
33    return *this;
34  }
```

reference-counted pointers

```
21  template <typename T>
22  inline shared_ptr<T>&
23  shared_ptr<T>::~shared_ptr() {
24    // destructor decrements shared counter *n_
25    if (--*n_)
26      // there are remaining pointers to *t_
27      // do not delete, return immediately
28      return;
29    delete t_;
30    delete n_;
31  }
32  #endif
```

ref-counted pointers example

```
1  #include <std::string>
2  #include "shared_ptr.h"
3
4  class Huge {
5    public: // ctor is private; force use of a factory method
6      static shared_ptr<Huge>  create(const char *s) {
7        return shared_ptr<Huge>(new Huge(s));
8      }
9      // ...
10   private:
11     std::string* data_;
12     Huge(const char* s): data_(new std::string_(s)) {}
13     ~Huge() {delete data_}
14     Huge& operator=(const Huge&); // forbidden
15 };
16
17 int
18 main(int, char **) {
19   shared_ptr<Huge> r = Huge::create("c++");
20   // next line copies only reference and increments ref. count
21   shared_ptr<Huge> p(r);
22 }
```

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

partial specialization of templates

```
1  template<int A, int B> // template pars can be int
2  struct power {
3    enum { result = power<A,B-1>::result * A };
4  };
5
6  template<int A> // partial specialization
7  struct power<A,0> {
8    enum { result = 1 };
9  };
10
11 std::cout << power<2,3>::result; // prints 8
12 //   power<2,3>::result
13 // = power<2,2>::result * 2 by line 3
14 // = power<2,1>::result * 2 * 2 by line 3
15 // = power<2,0>::result *2 * 2 * 2 by line 3
16 // = 1 *2 * 2 * 2 = 8 by line 8
```

metaprograms

- Compute result at compile time.
- Compute with types (and (enum) constants) as values.

```
template<typename parameter_type_1,
         typename parameter_type_2>
struct Function {
  typedef ... result_type;
  enum { result = .. };
};

.. Function<T1, T2>::result ..
.. Function<T1, T2>::result_type ..
```

- Use partial specialization to
  - test condition
  - stop recursion.
- computationally complete!

meta if .. then .. else

```
1  #ifndef IFTHENELSE_H
2  #define IFTHENELSE_H
3
4  // default
5  template<bool Condition, typename T_true, typename T_false>
6  struct IfThenElse;
7
8  // if Condition = true
9  template<typename T_true, typename T_false>
10 struct IfThenElse<true, T_true, T_false> {
11   typedef T_true result_type;
12 };
13
14 // if Condition = true
15 template<typename T_true, typename T_false>
16 struct IfThenElse<false, T_true, T_false> {
17   typedef T_false result_type;
18 };
19
20 // IfThenElse<(1<3),int,double>::result_type => int
21 #endif
```

meta square root function

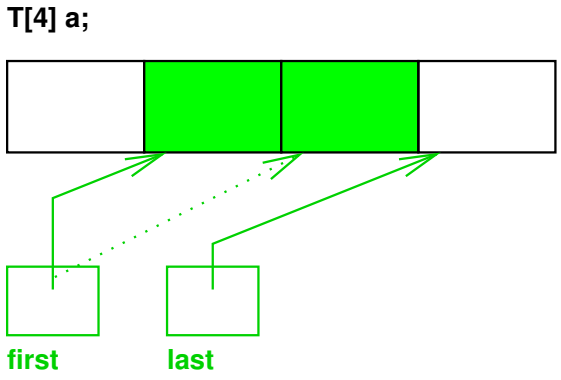
```
1  #ifndef SQRT_H
2  #define SQRT_H
3  // sqrt<N, Lo, Hi>: Lo <= sqrt(N) <= Hi
4  template<int N, int Lo=1, int Hi = N>
5  struct Sqrt {
6    enum { mid = (Lo + Hi + 1)/2 };
7
8    enum { result =
9      IfThenElse< (N < mid * mid),
10                   Sqrt<N, Lo, mid-1>,
11                   Sqrt<N, mid, Hi>
12               >::result_type::result };
13 };
14
15 // sqrt<N,L,L>: sqrt(N) = L
16 template<int N, int L>
17 struct Sqrt<N,L,L> {
18   enum { result = L };
19 };
20 #endif
```

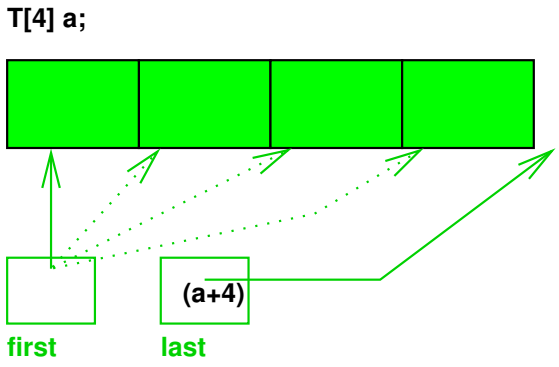
square root example computation (by compiler)

```
1  Sqrt<12>::result
2  Sqrt<12,1,12>::result
3  IfThenElse<(12<6*6),
4    Sqrt<12,1,5>,
5    Sqrt<12,6,12> >::result_type::result
6  Sqrt<12,1,5>::result_type::result
7  IfThenElse<(12<3*3),
8    Sqrt<12,1,2>,
9    Sqrt<12,3,5> >::result_type::result
10 Sqrt<12,3,5>::result_type::result
11 IfThenElse<(12<4*4),
12   Sqrt<12,3,3>,
13   Sqrt<12,4,5> >::result_type::result
14 Sqrt<12,3,3>::result_type::result
15 3
```

Outline

motivation: sequential search



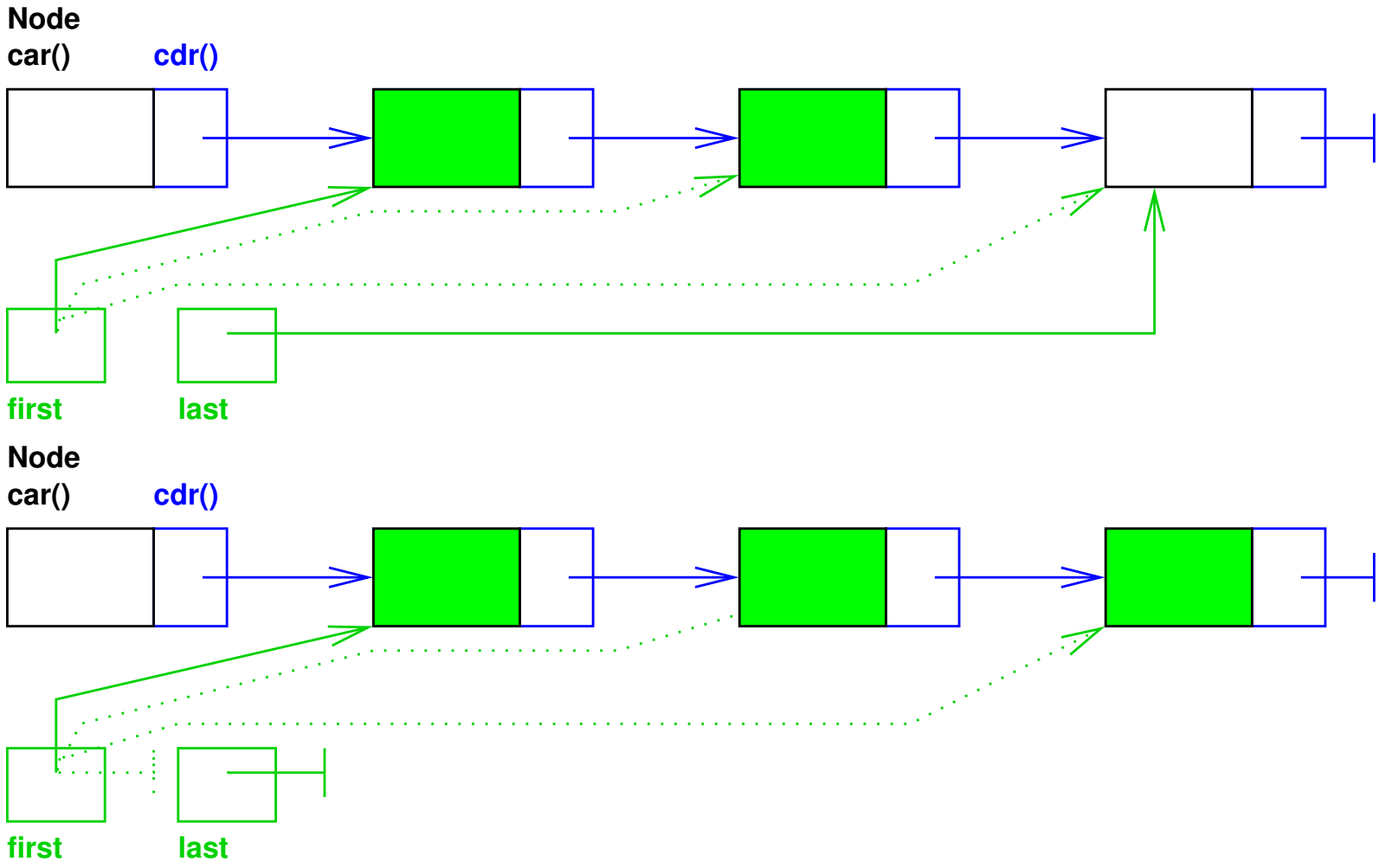



```
1  template <typename T>
2  const T*
3  find(const T* first, const T* last, const T& value) {
4    // sequential search for value in [*first..*last[;
5    // return last if not found
6    while (first!=last && *first!=value)
7      ++first;
8    return first;
9  }
10
11 extern int a[SIZE];
12
13 // find '20' in array a
14 find(a, a+SIZE, 20); // return pointer to 20 or a+SIZE
```

sequential search in linked list

```
1  // Node<T>* is a (too) simple linked list of T
2  template <typename T>
3  class Node {
4    public:
5      Node(const T& t, Node* cdr = 0): value_(t), cdr_(cdr) {}
6      // cons as in Scheme, i.e. prepend
7      Node*  cons(const T& t) { return new Node(t, this); }
8      const T&  car() const { return value_; }
9      Node*  cdr() const { return cdr_; }
10   private:
11     T  value_;
12     Node*  cdr_;
13 };
```

sequential search in linked list



sequential search in linked list

```
14  template <typename T>
15  Node<T>* // search for value in linked list [first .. last[
16  find(Node<T>* first, Node<T>* last, const T& value) {
17    while (first!=last && first->car()!=value)
18      first = first->cdr();
19    return first;
20  }
21
22  extern Node<int> *list;
23  find(list, static_cast<Node<int>*>(0), 20);
```

the essence of sequential search in a container

Advance a **Cursor** until either

- the value is found, or
- all elements of the container have been checked

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

requirements on cursor

```
1  class Cursor { // + constructors
2    public:
3      // dereference cursor to obtain an object
4      // from the container
5      T operator*();
6      // return cursor that refers to the next
7      // object in the container
8      Cursor operator++();
9      bool operator!=(Cursor); // compare
10     Cursor& operator=(const Cursor&); // assignable
11 };
12
13 // sequential search
14 Cursor
15 find(Cursor start, Cursor end, T value) {
16   while (start!=end && (*start != value) )
17     ++start;
18   return start;
19 }
```

a generic find function

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

```
1  template <typename InputIterator, typename T>
2  InputIterator
3  find(InputIterator first, InputIterator last, const T& value)
4    while (first != last && *first != value)
5      ++first;
6    return first;
7  }
```

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

```
1  template <typename T>
2  class Node {
3    public: // ..
4      // Cursor that satisfies find requirements
5      class Cursor {
6        public:
7          Cursor(Node* node=0): node_(node) {}
8          const T&  operator*() const { return node_->car(); }
9          Cursor& operator++() {
10           node_ = node_->cdr();
11           return *this;
12         }
13         bool  operator!=(const Cursor& c) const {
14           return node_ != c.node_;
15         }
16       private:
17         Node*  node_;
18     };
19 };
```

linked list iterator example

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

If we define

```
1  template <typename T>
2  class Node {
3    public:
4      // ...
5      class Cursor {
6        // ...
7      };
8      // return Cursor pointing to the start of the list
9      Cursor begin() { return Cursor(this); }
10     // return Cursor pointing past the end of the list
11     Cursor end() { return Cursor(); }
12 };
```

`find(list->begin(), list->end(), 20)` will work

containers and algorithms

The STL provides many container class templates:

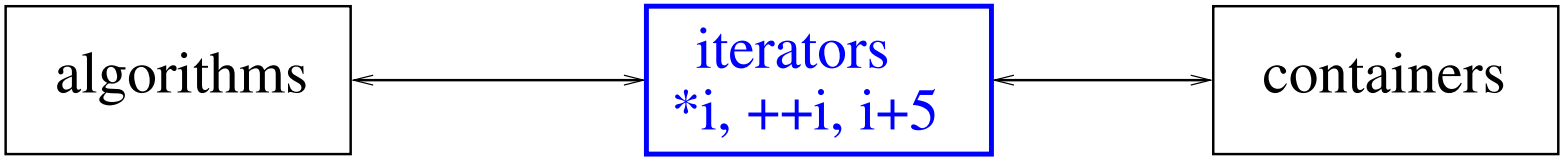
- **sequences**: vector, list, dequeue
- **associative**: set, multiset, map, multimap, hash table
- **adapters**: stack, queue, priority queue

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?

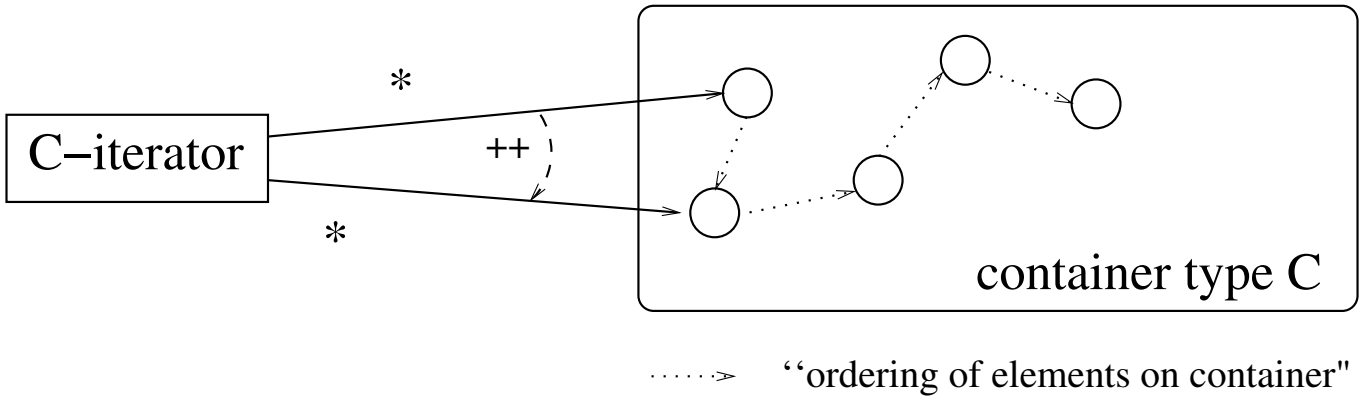
- To implement each algorithm for each container would require hundreds of implementations.
- **Why not write algorithms that work for many containers?**
- This is possible by putting a new abstraction between algorithms and containers (data structures): **iterators**



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 <typename Element>
class  C {
  CIterator  begin(); // "pointer" to first element in C
  CIterator  end(); // "pointer" beyond last element in C
};

template <typename Element>
class CIterator {
  public:
    Element  operator*();
    CIterator  operator++();
};
```

a simple algorithm using containers

```
1  // works for any iterator on any container
2  template <typename Iterator, typename T>
3  // return iterator pointing to cell containing value or last
4  Iterator
5  find(Iterator first, Iterator last, const T& value) {
6    while (first != last && *first != value)
7      ++first;
8    return first;
9  }
10
11 C<std::string> box;
12 std::string s("abc");
13 CIterator it;
14
15 if ( (it=find(box.begin(), box.end(), s)) == box.end())
16   cout << s << " not found" << endl;
17 else
18   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 +** $it[4], it+14$

algorithms and iterator kinds

- Some algorithms require a certain kind of iterator: what kind is required by **`find()`**?
- pointers are random access iterators!

```
1  extern std::string a[];
2  extern unsigned int a_size;
3
4  find(a, a + a_size, "abc");
```

example algorithm: sum

```
1  template <typename InputIterator, typename T>
2  T
3  sum(InputIterator first, InputIterator last) {
4    T result(*first++); // assume first!=last !!
5    while (first!=last)
6      result += *first++;
7    return result;
8  }
```

does not work:

```
extern Node<int>* l;
typedef Node<int>::Cursor list_iterator;

// error; compiler cannot deduce T=int
sum(l->begin(), l->end());
// ok
sum<list_iterator, int>(l->begin(), l->end());
```

```
1  template <typename InputIterator, typename T>
2  T
3  sum(InputIterator first, InputIterator last, T& result) {
4    if (first == last)
5      return result;
6    do
7      result += *first++;
8    while (first != last);
9    return result;
10 }
11
12 int r;
13 extern Node<int>*  l;
14
15 sum(l->begin(), l->end(), r); // ok
```

Now it works (why?)

associating types with iterators

```
1  template <typename T>
2  class Node {
3    public:
4      // ...
5      class Cursor {
6        public:
7          // Node::Cursor::value_type is type of *Cursor
8          typedef T value_type;
9          Cursor(Node* node=0): node_(node) {}
10         const T&  operator*();
11         Cursor&  operator++();
12         bool  operator!=(const Cursor& c) const;
13       private:
14         Node*  node_;
15     };
16     Cursor begin() const { return Cursor(this); }
17     Cursor end() const { return Cursor(); }
18   private:
19     // ...
20 };
```

sum with only 1 (deducible) template parameter

```
1  template <typename InputIterator>
2  typename InputIterator::value_type // what is ``typename''?
3  sum(InputIterator first, InputIterator last) {
4    assert(first!=last);
5    typename InputIterator::value_type result(*first++);
6    while (first!=last)
7      result += *first++;
8    return result;
9  }
10
11 extern Node<int>* list;
12 // ok
13 // InputIterator = Node<int>::Cursor &&
14 // InputIterator::value_type
15 // = Node<int>::Cursor::value_type
16 // = int
17 sum(list->begin(), list->end());
18 // BUT:
19 extern int a[10];
20 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 metaprogramming.

iterator traits

```
1  template <typename Iter>
2  // default; ok if Iter is a class type
3  struct iterator_traits {
4    typedef typename Iter::iterator_category iterator_category;
5    typedef typename Iter::value_type value_type;
6    typedef typename Iter::difference_type difference_type;
7    typedef typename Iter::pointer pointer;
8    typedef typename Iter::reference reference;
9  };
```

defines compile-time “function”

$$\mathit{Iterator} \longrightarrow \textbf{iterator\_traits}\langle \mathit{Iterator} \rangle$$

returning e.g. iterator_traits <Iterator>::value_type, the type of $*$i where i is of type Iterator .

iterator categories

- Found by iterator_traits <Iterator>::iterator_category
- Indicate kind of iterator.

```
1  struct input_iterator_tag {};
2  struct output_iterator_tag {};
3
4  // inheritance: see later
5  struct forward_iterator_tag: public input_iterator_tag {};
6
7  struct bidirectional_iterator_tag:
8      public forward_iterator_tag {};
9
10 struct random_access_iterator_tag:
11     public bidirectional_iterator_tag {};
```

iterator category of Cursor

```
1  template <typename T>
2  class Node {
3    public:
4      // ...
5      class Cursor {
6        public:
7          // type of *Cursor
8          typedef T value_type;
9          // Cursor is a forward iterator
10         typedef forward_iterator_tag iterator_category;
11         Cursor(Node* node=0);
12         const T&  operator*() const;
13         Cursor&  operator++();
14         bool  operator!=(const Cursor& c) const;
15       private:
16         Node*  node_;
17     };
18   private:
19     // ...
20 };
```

other iterator traits components

iterator_traits < Iterator >::

value_type	type of ∗i, with Iterator i
category	iterator kind of Iterator
reference	usually Iterator :: value_type&
pointer	& Iterator :: reference
difference_type	distance (ptrdiff_t)

specializing iterator traits for pointer types

```
1  template <typename T>
2  // iterator_traits specialization for pointer types
3  struct iterator_traits<T*> {
4    typedef random_access_iterator_tag iterator_category;
5    typedef T  value_type;
6    typedef ptrdiff_t difference_type;
7    typedef T*  pointer;
8    typedef T&  reference;
9  };
```

sum with iterator traits

```
1  template <typename InputIt>
2  typename iterator_traits<InputIt>::value_type
3  sum(InputIt first, InputIt last) {
4    assert(first!=last);
5    typename
6      iterator_traits<InputIt>::value_type result(*first++);
7    while (first != last)
8      result += *first++;
9    return result;
10 }
11
12 int  a[10];
13 sum(a, a+10);  // ok; why?
```

dispatching on iterator category 1

```
1  template <typename InputIt>
2  inline void // version for input and forward iterators
3  advance(InputIt& i,
4      typename iterator_traits<InputIt>::difference_type n,
5      input_iterator_tag) {
6    for (; n>0 ; --n)
7      ++i;
8  }
9
10 template <typename BidirectIt>
11 inline void // version for bidirectional iterator
12 advance(
13     BidirectIt& i,
14     typename iterator_traits<BidirectIt>::difference_type n,
15     bidirectional_iterator_tag) {
16   if (n>=0)
17     for (; n>0 ; --n) ++i;
18   else
19     for (; n<0 ; ++n) --i;
20 }
```

dispatching on iterator category 2

```
21  template <typename RandomIt>
22  inline void // version for random access iterators
23  advance( RandomIt& i,
24      typename iterator_traits<RandomIt>::difference_type n,
25      random_access_iterator_tag) {
26    i += n;
27  }
28
29  // the general version of advance dispatches to a more
30  // specialized one, depending on the iterator kind
31  template <typename Iterator>
32  inline void
33  advance(Iterator& i,
34      typename iterator_traits<Iterator>::difference_type n) {
35    advance(
36        i,
37        n,
38        typename iterator_traits<Iterator>::iterator_category()
39        );
40  }
```

advance call resolution

```
// I is the iterator type
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); // a+=10
```

will eventually resolve (explain!) to `a += 10`.

example containers

All in namespace std

- pair (of values)
- map (key->value)
- set (key->bool)
- (vector, list, dequeue, multimap, multiset)
- (adaptors: stack, queue, priority queue)
- (hash_map, hash_set)

pair

```
1  template <typename T1, typename T2>
2  struct pair {
3    typedef T1 first_type;
4    typedef T2 second_type;
5    T1 first;
6    T2 second;
7    pair() : first(T1()), second(T2()) {}
8    pair(const T1& a, const T2& b) : first(a), second(b) {}
9  };
```

pair

```
10  template <typename T1, typename T2>
11  inline bool
12  operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {
13    return x.first == y.first && x.second == y.second;
14  }
15
16  template <typename T1, typename T2>
17  inline bool
18  operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {
19    return x.first < y.first
20      || (!(y.first < x.first) && x.second < y.second);
21  }
22  // easier to type, e.g. make_pair(20, "abc")
23  template <typename T1, typename T2>
24  inline pair<T1, T2> make_pair(const T1& x, const T2& y) {
25    return pair<T1, T2>(x, y);
26  }
```

the “less” function template class

```
1  template <typename T>
2  struct less {
3    bool operator()(const T& x, const T& y) const {
4      return x <y;
5    }
6  };
```

useful, e.g. as template parameter for sort:

```
1  template <typename RandomAccessIt,
2            typename StrictWeakOrdering>
3  void
4  sort(
5    RandomAccessIt first,
6    RandomAccessIt last,
7    StrictWeakOrdering comp =
8       less<iterator_traits<RandomAccessIt>::value_type>()
9    );
```

example container: map

```
1  // a map implements a [Key->T] finite function
2  template <typename Key,
3            typename T,
4            typename Compare=less<Key>,
5            typename Alloc = alloc>
6  class map { // usually reptype is red-black tree
7    public:
8      typedef rep_type::iterator iterator;
9      typedef .. const_iterator;
10     // type of *iterator is <const Key,T> pair
11     typedef pair<const Key, T> value_type;
12     map(); // constructor
13     // put [*first,..,*last[ in a map
14     template <class InputIt> map(InputIt first, InputIt last);
15     iterator begin();
16     const_iterator begin() const;
17     iterator end();
18     const_iterator end() const;
```

example container: map

```
19      size_type size() const;
20      iterator find(const Key& x); // end() if not found
21      const_iterator find(const Key& x) const;
22      size_type count(const Key& x);
23      // result.first = where inserted,
24      // result.second = true iff ok
25      pair<iterator,bool> insert(const value_type& x);
26      T& operator[](const Key& k); // can be assigned to
27      template <class InputIt>
28        void insert(InputIt first, InputIt last);
29      void erase(iterator position);
30      // find and erase, return 1 iff ok
31      size_type erase(const Key& x);
32      // erase [*first .. *last[
33      void erase(iterator first, iterator last);
34      void clear(); // erase [begin(),..,end()[
35  };
```

map usage example

```
1  typedef std::map<std::string,int> Examen;
2
3  int main(int, char**) {
4    Examen  scores;
5    // insert
6    scores["john"] = 18; // std::string::string(const char*)
7    scores.insert(std::make_pair("fred", 5));
8    // retrieve
9    for (Examen::const_iterator i=scores.begin();
10       i!=scores.end(); ++i)
11     std::cout << (*i).first << ": " << (*i).second << std::endl
12   // update
13   scores["fred"] = 11; // 2de zittijd
14   Examen::iterator i = scores.find("john");
15   if (i != scores.end()) {
16     (*i).second = 13; // another way to update
17     std::cout << (*i).first << ": "
18       << (*i).second << std::endl;
19   }
20   return 0;
21 }
```

set (specialization of map)

```
1  // elements are kept in sorted order
2  template <typename Key, typename Compare = less<Key>,
3            typename Alloc = alloc>
4  class set {
5    public:
6      typedef Key value_type;
7      // there are only "constant" iterators. why?
8      typedef rep_type::const_iterator iterator;
9      set(); // constructor
10     set(const set<Key, Compare, Alloc>& x);
11     // create set from [*first,..,*last[
12     template <class InputIt> set(InputIt first, InputIt last);
13     iterator begin();
14     iterator end();
```

set

```
15      size_type size(); // cardinality
16      iterator find(const Key& x); // end() if not found
17      size_type count(const Key& x); // 1 or 0
18      // result.first = where inserted,
19      // result.second = true iff ok
20      pair<iterator,bool> insert(const value_type& x);
21      void erase(iterator position); // erase at position
22      size_type erase(const Key& x); // 1 if ok, 0 if not
23      // erase [*first,..,*last[
24      void erase(iterator first, iterator last);
25      void clear(); // erase(begin(),end());
26  };
```

for_each, find, find_if algorithms

```
1  template <typename InputIt, typename Function>
2  Function for_each(InputIt first, InputIt last, Function f) {
3    for ( ; first != last; ++first)
4      f(*first);
5    return f;
6  }
7
8  template <typename InputIt, typename T>
9  InputIt find(InputIt first, InputIt last, const T& value) {
10   while (first != last && *first != value)
11     ++first;
12   return first;
13 }
14
15 template <typename InputIt, typename Predicate>
16 InputIt find_if(InputIt first, InputIt last, Predicate pred) {
17   while (first != last && !pred(*first))
18     ++first;
19   return first;
20 }
```

the copy algorithm

```
1  // code is a simplification of the real thing
2  template <typename InputIt,
3            typename OutputIt>
4  inline OutputIt
5  copy(InputIt first, InputIt last, OutputIt result) {
6    for ( ; first != last; ++result, ++first)
7      *result = *first;
8    return result;
9  }
```

- how to copy to e.g. a set container?
- what if the receiver is a vector that is “too small”?

insert_iterator 1

```
1  // an iterator that translates *it = v to a container
2  // insert operation e.g.
3  //    copy(c1.begin(), c1.end(), inserter(c2,c2.begin()))
4  // will work fine
5  template <typename Cont>
6  class insert_iterator {
7    protected:
8      Cont* container;
9      typename Cont::iterator iter;
10   public:
11     insert_iterator(Cont& x, typename Cont::iterator i):
12         container(&x), iter(i) {}
13     insert_iterator<Cont>& operator=(
14         const typename Cont::value_type& value) {
15       iter = container->insert(iter, value);
16       ++iter;
17       return *this;
18     }
```

insert_iterator 1

```
19      // the next member functions do nothing
20      insert_iterator<Cont>& operator*() { return *this; }
21      insert_iterator<Cont>& operator++() { return *this; }
22      insert_iterator<Cont>& operator++(int) { return *this; }
23  };
24
25  // this function makes it easy to use an insert iterator
26  template <typename Cont, typename Iterator>
27  inline insert_iterator<Cont>
28  inserter(Cont& x, Iterator i) {
29    typedef typename Cont::iterator iter;
30    return insert_iterator<Cont>(x, iter(i));
31  }
```

other interator adaptors

Other iterator adaptors are available: e.g.

- back_insert_iterator (push_back),
- front_insert_iterator (push_front),
- stream iterators

istream_iterator

```
1  // input iterator that reads values from a stream
2  template <typename T, typename Distance = ptrdiff_t>
3  class istream_iterator {
4    friend bool operator==(
5        const istream_iterator<T, Distance>& x,
6        const istream_iterator<T, Distance>& y
7        );
8    protected:
9      // e.g. value = *iter++; will read value from stream
10     istream* stream; // from which data is read
11     T value; // last read from stream
12     bool can_read; // true iff not yet at end
13     void read() {
14       can_read = (stream && *stream) ? true : false;
15       if (can_read)
16         *stream >> value;
17       can_read = (*stream) ? true : false;
18     }
```

istream_iterator

```
19    public:
20      typedef T  value_type; // for iterator_traits
21      typedef const T*  pointer; // for iterator_traits
22      typedef const T&  reference; // for iterator_traits
23      istream_iterator() : stream(0), can_read(false) {}
24      istream_iterator(istream& s) : stream(&s) { read(); }
25      reference operator*() const { return value; }
26      pointer operator->() const { return &(operator*()); }
27      istream_iterator<T, Distance>& operator++() {
28        read();
29        return *this;
30      }
31      istream_iterator<T, Distance> operator++(int)  {
32        istream_iterator<T, Distance> tmp = *this;
33        read();
34        return tmp;
35      }
36  };
```

ostream_iterator

```
1  // an output iterator that writes to a stream
2  // e.g. *it = value will write value on the stream of it
3  template <typename T>
4  class ostream_iterator {
5    protected:
6      ostream* stream;
7      const char* string; // what is this used for?
8    public:
9      typedef void  value_type; // ...
10     ostream_iterator(ostream& s) : stream(&s), string(0) {}
11     ostream_iterator(ostream& s, const char* c) : stream(&s),
   {}
12     ostream_iterator<T>& operator=(const T& value) {
13       *stream << value;
14       if (string)
15         *stream << string;
16       return *this;
17     }
18     ostream_iterator<T>& operator*() { return *this; }
19     ostream_iterator<T>& operator++() { return *this; }
20     ostream_iterator<T>& operator++(int) { return *this; }
21 };
```

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
```

tel.h

```
1  #ifndef TEL_H
2  #define TEL_H
3  // $Id: structuur2.tex,v 1.26 2008/09/02 11:51:15 dvermeir Exp
4  #include <string>
5  #include <iostream>
6
7  // associates a string info_ with a string name_
8  class TelRecord {
9    public:
10     TelRecord(const std::string& name="",
11       const std::string& info=""): name_(name), info_(info) {}
12     TelRecord(const TelRecord& r):
13       name_(r.name_), info_(r.info_) {}
14     // two records are the same if their names match
15     bool operator==(const TelRecord& r) const {
16       return name_ == r.name_;
17     }
```

```
18      // needed for std::set<TelRecord>
19      bool operator<(const TelRecord& r) const {
20        return name_ < r.name_;
21      }
22      friend std::ostream& operator<<(std::ostream& os,
23        const TelRecord& r);
24      friend std::istream& operator>>(std::istream& os,
25        TelRecord& r);
26    private:
27      // name_ is a unique key: no 2 records can
28      // have the same name_, name_ may not
29      // contain white space
30      std::string name_;
31      std::string info_;
32  };
33  #endif
```

tel.C

```
1  // $Id: structuur2.tex,v 1.26 2008/09/02 11:51:15 dvermeir Exp
2  // usage:
3  //  tel name       -- query: retrieve info for name
4  //                    from database
5  //  tel name info -- insert: store/replace info for
6  //                    name in database
7  //  tel name ""    -- delete: remove name from database
8  //  tel            -- list whole database
9  //
10 // error return codes:
11 //
12 //    1       -- query: not found
13 //    2       -- insert: cannot write database file
14 //    3       -- delete: not found
15 //    4       -- too many arguments
16 //
17 // only 1 info string can be associated with a name
18 // in the database
```

```
19 #include <iterator>
20 #include <fstream>
21 #include <set>
22 #include "tel.h"
23
24 std::ostream&
25 operator<<(std::ostream& os, const TelRecord& r) {
26   os << r.name_ << '\t' << r.info_ << std::endl;
27   return os;
28 }
29
30 std::istream&
31 operator>>(std::istream& is, TelRecord& r) {
32   is >> r.name_ ; // line starts with name
33   is.ignore(); // ignore \t following name
34   std::getline(is, r.info_); // rest of line is info
35   return is;
36 }
```

```
37 // note that to use set<T>, T needs a default and a
38 //   copy ctor as well as an operator<
39 typedef std::set<TelRecord> record_set;
40 typedef std::istream_iterator<TelRecord> record_input_it;
41
42 const char* const database_name = "tel.data";
43
44 int
45 main (int argc, char *argv[]) {
46   record_set dir;
47   { // Load the database into a record_set
48     std::ifstream data_file(database_name);
49     if (data_file)
50       std::copy(
51         record_input_it(data_file),
52         record_input_it(),
53         std::inserter(dir, dir.begin())
54         );
55   }
56   // data_files is automatically closed by the ifstream destru
```

```
57    switch (argc) {
58      // no arguments: just show all records on std::cout
59      case 1:
60        copy( dir.begin(), dir.end(),
61              std::ostream_iterator<TelRecord>(std::cout)
62            );
63        return 0;
```

```
64      // 1 argument: retrieve info associated with argv[1]
65      case 2:
66        { // use the find() algo with dummy TelRecord with name_
67          std::string key(argv[1]);
68          record_set::iterator r = dir.find(TelRecord(key, ""));
69
70          if (r!=dir.end()) {
71            // success: iterator points to found record
72            std::cout << *r; // output it
73            return 0;
74          }
75          else {
76            std::cerr << "\"" <<  key
77              << "\" not found" << std::endl;
78            return 1;
79          }
80        }
81        break;
```

```
82    // 2 arguments, insert, replace or delete
83    case 3: {
84      std::string key(argv[1]);
85      std::string info(argv[2]);
86      TelRecord r(key, info);
87
88      if (info.size()==0) { // delete
89        if (dir.erase(r)!=1) {
90          // set::erase() returns number of deleted elements
91          std::cerr << "\"" << key
92            << "\" not found; cannot erase" << std::endl;
93          return 3;
94        }
95      }
96      else { // insert or replace
97        std::pair<record_set::iterator, bool>  pair =
98            dir.insert(r);
99        if (!pair.second) { // insert failed: duplicate key
100         dir.erase(pair.first); // erase,
101         dir.insert(r); // then insert again
102       }
103     }
```

```
104       // save to file after update
105       std::ofstream data_file(database_name);
106       if (!data_file) {
107         std::cerr << "Cannot open \"" << database_name
108           << "\" for writing" << std::endl;
109         return 2;
110       }
111       std::copy(dir.begin(), dir.end(),
112            std::ostream_iterator<TelRecord>(data_file));
113       return 0;
114       }
115       break;
116     default:
117       std::cerr << "Usage: " << argv[0]
118         << " [name [info|\"\"]] " << std::endl;
119       return 4;
120       break;
121     }
122     return 0;
123   }
```

Outline

problem description

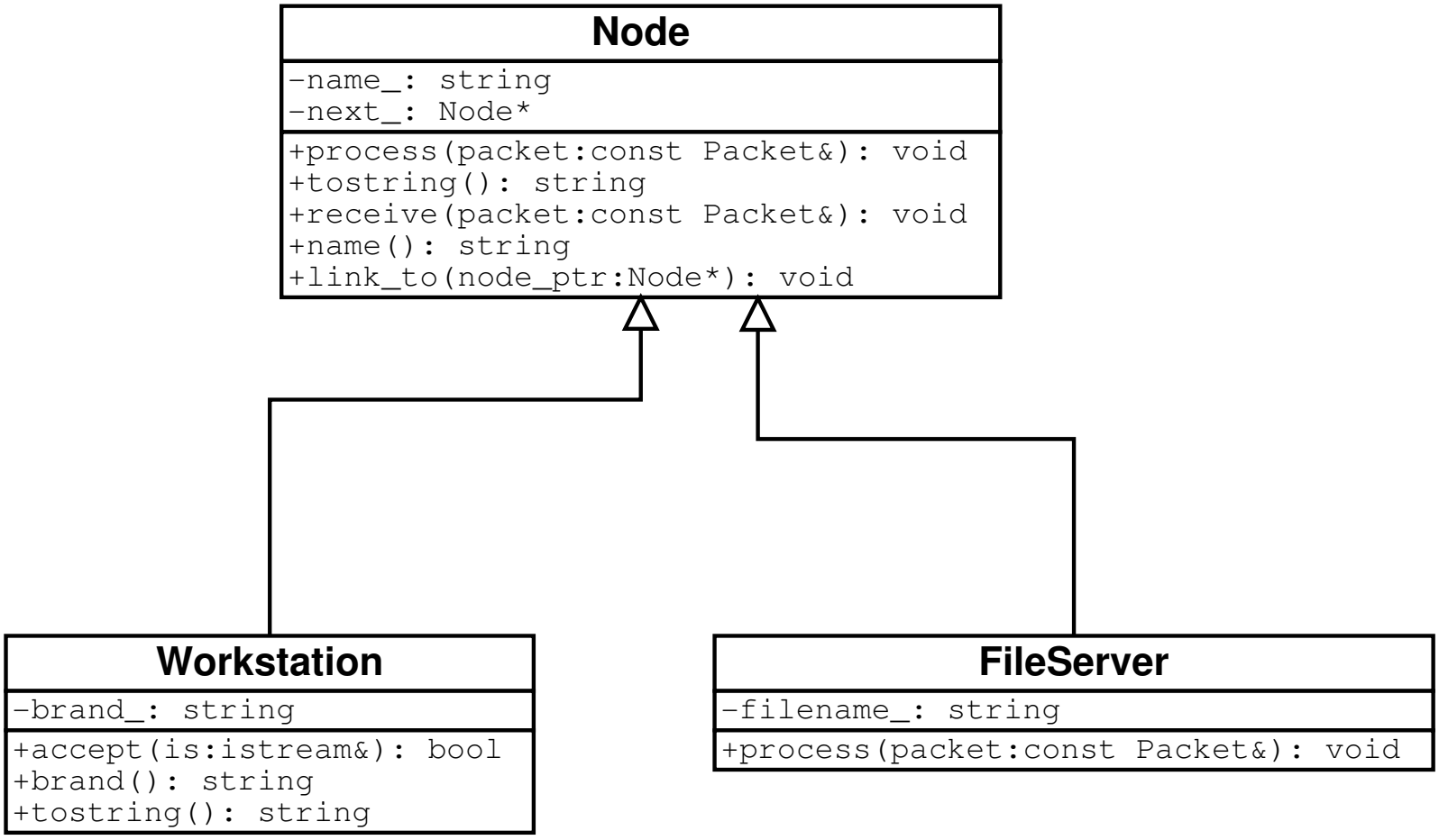
A circular LAN consists of **nodes**. Nodes process **packets** that are addressed to them; and may pass other packets on to the next node. Besides “simple nodes” there are several more sophisticated types of nodes: e.g. **workstations** may generate new packets, **fileservers** save their packets in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

derived classes

- 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**.

```
1  class Node;
2  // Workstation is derived from Node
3  class Workstation: public class Node;
4  // Fileserver is derived from Node
5  class Fileserver: public class Node;
```

uml class diagram



packet.h

```
1  #ifndef PACKET_H
2  #define PACKET_H
3  #include <string>
4  #include <iostream>
5  class Packet { // ``bare bones'' implementation
6    public:
7      Packet(const std::string& destination,
8          const std::string& load): destination_(destination),
9                                    contents_(load) {}
10     std::string  destination() const { return destination_; }
11     std::string  contents() const { return contents_; }
12   private:
13     std::string  destination_; // name of node
14     std::string  contents_;
15 };
16
17 inline std::ostream&
18 operator<<(std::ostream& os, const Packet& p) {
19   return os << "[" << p.destination()
20     << ", \"" << p.contents() << "\"]";
21 }
22 #endif
```

node.h

```
1  #ifndef NODE_H
2  #define NODE_H
3  #include <string>
4  #include "packet.h"
5  class Node {
6    public:
7      Node(const std::string& name, Node* next=0):
8        name_(name), next_(next) {}
9      // what we do with a packet we receive
10     void receive(const Packet&);
11     // what we do with a packet destined for us
12     void process(const Packet&);
13     void link_to(Node* node_ptr) { next_ = node_ptr; }
14     std::string name() const { return name_; }
15     std::string tostring() const; // nice printable name
16   private:
17     std::string name_; // unique name of Node
18     Node* next_; // in LAN
19 };
20 #endif
```

node.C

```
1  #include "node.h"
2  void
3  Node::receive(const Packet& packet) {
4    std::cout << tostring() << " receives " << packet << std::en
5    if (packet.destination()==name())
6      process(packet);
7    else
8      if (next_) // pass it on
9        next_->receive(packet);
10 }
11
12 void
13 Node::process(const Packet& packet) {
14   std::cout << tostring() <<  " processes "
15     << packet << std::endl;
16 }
17
18 std::string
19 Node::tostring() const {
20   return "node " + name_; // explain?
21 }
```

workstation.h

```
1  #ifndef WORKSTATION_H
2  #define WORKSTATION_H
3  #include <iostream>
4  #include "node.h"
5  // Workstation is publicly derived from Node
6  class Workstation: public Node {
7    public:
8      Workstation(const std::string name,
9          const std::string brand, Node* next=0):
10         Node(name,next), brand_(brand) {}
11     // extra: accept packet from cin and send it to next
12     bool  accept(std::istream& is);
13     // override Node::tostring()
14     std::string  tostring() const;
15     // extra function
16     std::string  brand() const { return brand_; }
17   private:
18     std::string  brand_; // extra data member
19 };
20 #endif
```

workstation.C

```
1  #include "workstation.h"
2
3  bool
4  Workstation::accept(std::istream& is) {
5    // packet format: destination contents \n
6    std::string destination;
7    std::string contents;
8    std::cout << tostring()
9      << " accepts a packet.." << std::endl;
10   is >> destination;
11   std::getline(is, contents);
12   if (destination.size()==0) {
13     std::cerr << "error: no destination" << std::endl;
14     return false;
15   }
16   Packet packet(destination, contents);
17   receive(packet); // ``send'' the packet to myself
18   return true;
19 }
```

workstation.C

```
1  std::string
2  Workstation::tostring() const {
3    return Node::tostring() + " (" + brand() + ")";
4  }
```

fileserver.h

```
1  #ifndef FILESERVER_H
2  #define FILESERVER_H
3  #include <iostream>
4  #include "node.h"
5  // FileServer is publicly derived from Node
6  class FileServer: public Node {
7    public:
8      FileServer(const std::string name, Node* next=0):
9          Node(name,next) {}
10     // override Node::process()
11     void process(const Packet& packet) {
12       std::cout << "FileServer " << tostring()
13         << ": storing packet" << packet << std::endl;
14     }
15 };
16 #endif
```

program lan.C

```
1  #include  "node.h"
2  #include  "workstation.h"
3
4  int
5  main(int, char**) {
6    Workstation  wilma("wilma", "DELL_BIG");
7    Node  node1("fred", &wilma); // node1 --> wilma
8    Node  node2("jane", &node1); // node2 --> node1
9    wilma.link_to(&node2); // wilma --> node2
10   // now we have a circular net:
11   //   wilma --> node2 --> node1 --> wilma
12   while (wilma.accept(std::cin))
13     ;
14 }
```

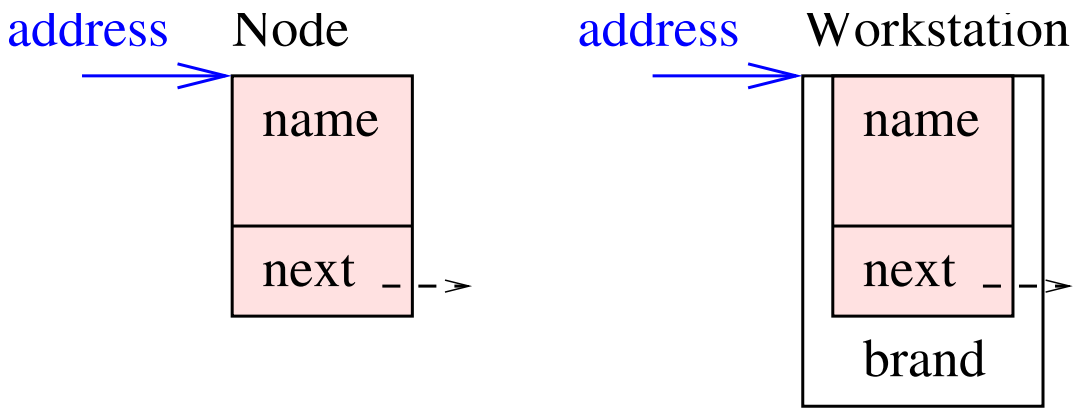
running lan

```
15 3039 dv2:~/courses/structuur2/slides$ lan
16 node wilma (DELL_BIG) accepts a packet..
17 fred hi there, fred
18 node wilma receives packet [fred, " hi there, fred"]
19 node jane receives packet [fred, " hi there, fred"]
20 node fred receives packet [fred, " hi there, fred"]
21 node fred processes [fred, " hi there, fred"]
22 node wilma (DELL_BIG) accepts a packet..
23 3040 dv2:~/courses/structuur2/slides$
```

Explain these lines:

node wilma (DELL_BIG) accepts a packet..
node wilma 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

- Consider a pointer nodeptr (to a Workstation) with type Node∗:

```
Workstation ws("wilma", "DELL_BIG");
Node* nodeptr(&ws); // ok
```

- We would like nodeptr−>tostring() to call Workstation::tostring ().
- This can be achieved by declaring Node::tostring to be **virtual**:

```
        virtual std::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.

node2.h (virtual..)

```
1  #ifndef  NODE_H
2  #define  NODE_H
3
4  #include <string>
5  #include "packet.h"
6
7  class Node { // version with virtual functions
8  public:
9    Node(const std::string& name, Node* next=0):
10     name_(name), next_(next) {}
11   virtual ~Node() {} // virtual destructor, see later
12   void receive(const Packet&);
13   virtual void  process(const Packet&); // for FileServer
14   void link_to(Node* node_ptr) { next_ = node_ptr; }
15   std::string name() const { return name_; }
16   virtual std::string  tostring() const;
17 private:
18   std::string name_; // unique name of Node
19   Node* next_; // in LAN
20 };
21 #endif
```

Source with virtual functions

No change in `node.C`, `workstation.[hC]`, `lan.C`, `packet.h`, `fileserver.h`.

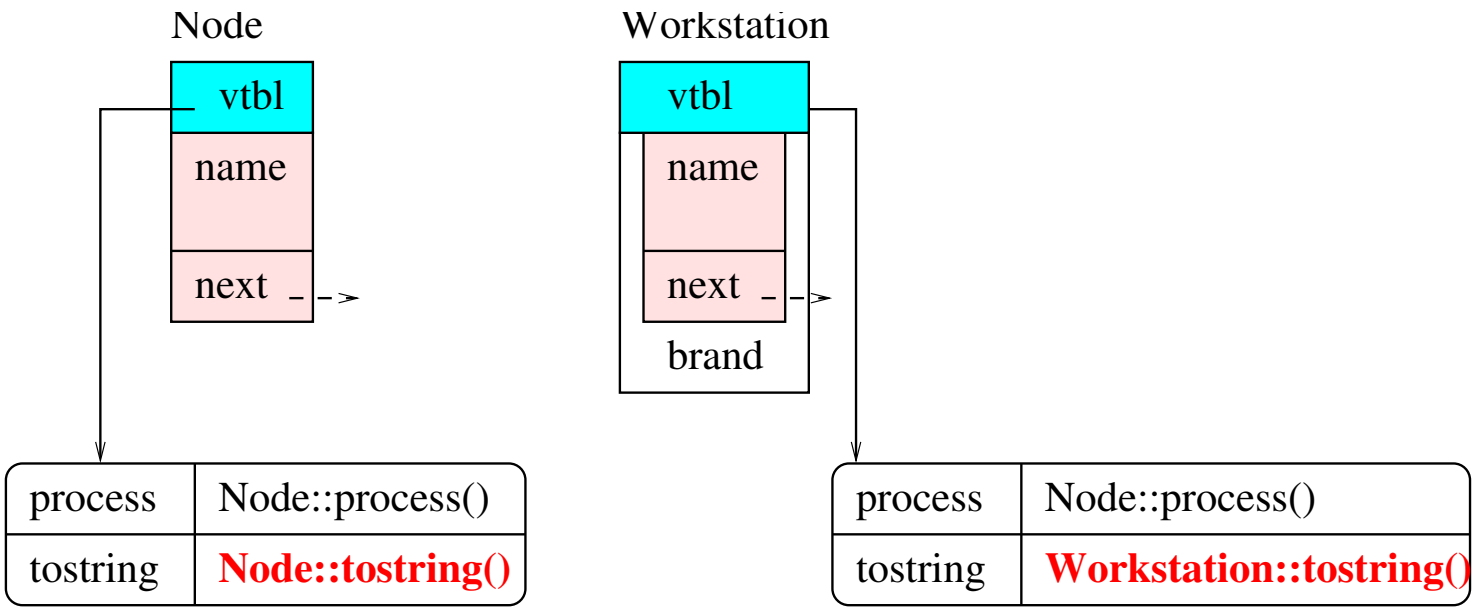
running lan2

```
1  3089 dv2:~/courses/structuur2/slides$ lan2
2  node wilma (DELL_BIG) accepts a packet..
3  fred hi there,fred
4  node wilma (DELL_BIG) receives packet [fred, " hi there,fred"]
5  node jane receives packet [fred, " hi there,fred"]
6  node fred receives packet [fred, " hi there,fred"]
7  node fred processes [fred, " hi there,fred"]
8  node wilma (DELL_BIG) accepts a packet..
```

Explain this line:

```
4  node wilma (DELL_BIG) receives packet [fred, " hi there,fred"]
```

object layout w. virtual functions: vtbls




```
1  Node*    nodeptr;
2  nodeptr->tostring()
```

is translated to

```
nodeptr->vtbl[1]()
```

virtual functions without refs or pointers

```
1  Workstation ws("wilma", "DELL");
2  Node node(ws);
3
4  node.tostring();
```

Which function is executed? why?

pure virtual functions

```
1  class Animal {
2  public:
3    virtual std::string sound() const = 0;
4  };
5
6  class Dog: public Animal {
7    std::string sound() const { return "woef"; }
8  };
9  class Cat: public Animal {
10   std::string sound() const { return "miauw"; }
11 };
12
13 std::set<Animal*>    pets;
14 for (std::set<Animal*>::const_iterator a=pets.begin();
15       a!=pets.end();
16       ++a)
17   std::cout << (*a)->sound() << std::endl;
18
19 Animal  fred; // error, why?
```

abstract classes

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

```
1  #ifndef STACK_H
2  #define STACK_H
3  // an abstract class, aka interface in java
4  template<typename T>
5  class Stack {
6    public:
7      virtual T  pop() = 0;
8      virtual void  push(const T&) = 0;
9      virtual unsigned int  size() const = 0;
10     // clear() only uses pure virtual functions
11     void  clear() {
12       while (size()>0)
13         pop();
14     }
15     virtual ~Stack() {} // see further
16 };
17 #endif
```

implementing abstract classes

```
1  #ifndef LISTACK_H
2  #define LISTACK_H
3  #include "stack.h"
4  #include <list>
5  // We implement a stack using a list; the top of the stack
6  // is represented by the front of the list.
7  template <typename T>
8  class ListStack: public Stack<T> {
9    private:
10     std::list<T> list_;
11   public:
12     ListStack(): list_() {}
13
14     T  pop() {
15       T t = list_.front(); list_.pop_front(); return t;
16     }
17     void  push(const T& t) { list_.push_front(t); }
18     unsigned int  size() const { return list_.size(); }
19 };
20 #endif
```

using ListStack

```
1  #include "liststack.h"
2
3  int
4  main(int, char**) {
5    Stack<int>* s = new ListStack<int>();
6    for (int i=0; i<10; ++i)
7      s->push(i);
8    while (s->size()>0)
9      std::cout << s->pop() << std::endl;
10 }
```

virtual destructors

```
1  class Person {
2    public:
3      Person(const std::string& name): name_(name) {}
4      std::string  name() const { return name_; }
5    private:
6      std::string  name_;
7  };
8
9  class Image { // ... something *BIG*
10 };
11
12 class FamilyMember: public Person {
13   public:
14     FamilyMember(const std::string& name, Image* im=0):
15       Person(name), image_(im) {}
16     ~FamilyMember() { if (image_) delete image_; }
17     Image*  image() const { return image_; }
18   private:
19     Image*  image_;
20 };
```

virtual destructors

```
1  #include "person.h"
2
3  Person* p(new FamilyMember("fred",im));
4  delete p; // which destructor will be called? why?
```

To solve:

```
1  class Person {
2    public:
3      Person(const std::string& name): name_(name) {}
4      virtual ~Person() {} // how will this solve the problem?
5      std::string  name() const { return name_; }
6    private:
7      std::string  name_;
8  };
```

private derivation & multiple inheritance

```
1  #ifndef LISTACK_H
2  #define LISTACK_H
3  #include "stack.h"
4  #include <list>
5  // inheriting privately from list<T> ensures that users of
6  // ListStack cannot access the underlying list<T>
7  // operations; this makes ListStack a properly encapsulated
8  // Abstract Data Type
9  template <typename T>
10 class ListStack: public Stack<T>, private std::list<T> {
11 public:
12   ListStack(): std::list<T>() {}
13
14   T  pop() { T t = front(); pop_front(); return t; }
15   void  push(const T& t) { push_front(t); }
16   unsigned int  size() const {
17     return std::list<T>::size(); // what's this?
18   }
19 };
20 #endif
```

private derivation & multiple inheritance

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

multiple and virtual inheritance

```
1  #include          <string>
2
3  struct Person {
4    std::string  name;
5  };
6
7  struct Staff: public virtual Person {
8    int  salary;
9  };
10
11 struct Student: public virtual Person {
12   int  rolnr;
13 };
14
15 struct Tutor: public Student, public Staff {
16 }; // how many name's does a Tutor have?
```

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

iostreams: protected members

```
1  // manages buffer and communication with device
2  class streambuf;
3
4  class ios { // format state, status info, streambuf etc.
5    protected: // only derived classes can create an ios
6      ios(streambuf*);
7  };
8
9  class istream: virtual public ios {
10   public:
11     istream(streambuf* buf): ios(buf) { /* .. */ }
12     int read(char* buffer, unsigned int size);
13 };
14
15 class ostream: virtual public ios {
16   public:
17     ostream(streambuf* buf): ios(buf) { /* .. */ }
18     int write(char* buffer, unsigned int size);
19 };
```

iostreams: protected members

```
20  class iostream: public istream, public ostream {
21    public: // why explicit call to ios(buf)?
22      iostream(streambuf* buf):
23        istream(buf), ostream(buf), ios(buf) {/*..*/}
24  };
25
26  // a stream connected to a file
27  class fstream: public iostream {
28  };
```

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 if necessary **and** virtual destructor.
- 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

```
1  struct Base { // 4 bytes
2    Base(int i=0): i(i) {}
3    int i;
4  };
5
6  struct Derived: public Base { // 8 bytes
7    Derived(int i=0,int j=0): Base(i), j(j) {}
8    int j;
9  };
10
11 void
12 f(Base a[]) {
13   std::cout <<  a[0].i << ", " << a[1].i << std::endl;
14 }
15
16 int
17 main() {
18   Derived da[] = { Derived(1,2), Derived(3,4) };
19   f(da); // prints 1,2 instead of the expected 1,3
20 }
```

Outline

exceptions: motivation

```
1  Rational::Rational(int num=0, int denom=0): num_(num),
2      denom_(denom) {
3    assert(denom!=0); // >>>>>> Abort if denom == 0.
4  }
5
6  std::istream&
7  operator>>(std::istream& is,Rational& r) {
8    // reads things like 2/3, 4
9    // Lots of stuff omitted: see book p. 78.
10   is >> r.denom_;
11   assert(r.denom_!=0); // >>> Abort if denominator == 0.
12   return is;
13 }
```

Find a better way to handle such errors, e.g. in a constructor.

an exception class

```
1  class RationalZeroDenom {
2    public:
3      RationalZeroDenom(int n): num_(n) {}
4      friend std::ostream& operator<<(std::ostream& os,
5                                      const RationalZeroDenom& e) {
6        return os << num_ << "/0 is not a legal Rational";
7      }
8    private:
9      int num_;
10 };
```

throwing exceptions

```
1  std::istream&
2  operator>>(std::istream& is, Rational& r)
3      throw (RationalZeroDenom, std::exception) {
4    is >> r.denom_; // Stuff omitted: see book p. 78.
5    if (r.denom_==0)
6      throw RationalZeroDenom(r.num_);
7    return is;
8  }
```

Make copy of thrown object and exit function, its caller, etc. up to a call in a try block with a catch clause matching the type of the exception.

catching exceptions: catch(arg) is like function call.

```
1  while (std::cin) {
2    try { // any exceptions thrown in (functions called from
3      // within) this block will be caught.
4      Rational r;
5      std::cout << "input? " << std::endl;
6      std::cin >> r;
7      ..
8    }
9    catch (RationalZeroDenom& e) { // Complain and continue.
10     std::cerr << e << ", try again" << std::endl;
11   }
12   catch (std::exception& e) { // Complain and throw it again.
13     std::cerr << e.what() << std::endl;
14     throw; // Re-throw e.
15   }
16   catch (...) { // Complain and throw it again
17     std::cerr << "A weird exception was thrown" << std::endl;
18     throw;
19   }
20 }
```

more on exceptions

- When unwinding the stack, local objects are destructed $\Rightarrow$ release resources in destructors.
- If an exception propagates to the top, **`std::terminate()`** is called.
- When throwing an exception from a constructor **`C::C(..)`**, the destructor `C::~C()` is **not** called (but the destructors of the data members are).
- Exceptions thrown from a destructor: see book, p. 212 - 213.
- Exception specifications and unexpected exceptions: see book p. 213 - 214.

Outline

a good programs is:

- **correct**, i.e. it implements its **specification**,
- **robust**, i.e. it behaves **gracefully** when confronted with unexpected events,
- easy to **maintain** (maintenance costs 5$\times$ development). This implies:
  - it is easy to **understand**.
  - it is easy to **modify** and **extend**.
  - it consists of **parts** that can be **reused** elsewhere.

These criteria are not completely independent. E.g. decomposing into reusable parts may make the program easier to understand.

an example program

```
1  #include <iostream>
2  #include <stdlib.h> // for strtod(char*,char**)
3  // quotient: write quotient of arg1 and arg2 on stdout.
4  int
5  main(int argc,char* argv[]) {
6    // strtod(char* p,0) converts initial part of
7    //   C-string starting at p to double
8    std::cout << strtod(argv[1], 0)/strtod(argv[2], 0) << "\n";
9  }
```

Is this a good program?

a robust version

But more difficult to read & maintain.

```
1  #include <string>
2  #include <iostream>
3  #include <stdlib.h> // for strtod(char*,char**)
4  // quotient: write quotient of arg1 and arg2 on stdout.
5  static const std::string USAGE("quotient number number");
6  static const std::string FORMAT_ERR("not a number");
7  static const std::string DIVIDE_BY_ZERO("divide by 0");
8
9  int
10 main(int argc, char* argv[]) {
11   char *end; // see the man page for strtod
12
13   if (argc!=3) {
14     std::cerr << "usage: " << USAGE << std::endl;
15     return 1;
16   }
```

```
1   double  a1(strtod(argv[1], &end));
2   if (end==argv[1]) {
3     std::cerr << "\"" << argv[1] << "\": "
4       << FORMAT_ERR << std::endl;
5     return 1;
6   }
7
8   double  a2(strtod(argv[2], &end));
9   if (end==argv[2]) {
10    std::cerr << "\"" << argv[2] << "\": "
11      << FORMAT_ERR << std::endl;
12    return 1;
13  }
14  if (a2==0) {
15    std::cerr << DIVIDE_BY_ZERO << std::endl;
16    return 1;
17  }
18
19  std::cout << a1/a2 << std::endl;
20  return 0;
21 }
```

good decomposition: better maintenance

```
1  #include <iostream>
2  #include <string>
3  #include <stdexcept> // for standard exception classes
4  #include <stdlib.h> // for strtod(char*, char**)
5  // quotient: write quotient of arg1 and arg2 on stdout.
6  static const std::string
7    USAGE("usage: quotient number number");
8  static const std::string
9    DIVIDE_BY_ZERO("cannot divide by 0");
```

a reusable part

```
1  // A reusable part: this function returns the double
2  // represented by s; it throws a range_error exception
3  // if s does not represent a double.
4
5  double
6  cstr2double(const char* s) throw(std::range_error) {
7    static const std::string
8      FORMAT_ERR("cstr2double: not a number");
9    char*  end;
10   double  d(strtod(s, &end));
11
12   if (s==end)
13     throw std::range_error(std::string(s)+": "+FORMAT_ERR);
14   return d;
15 }
```

new main program

```
1  int
2  main(int argc, char* argv[]) {
3    try {
4      if (argc!=3)
5        throw std::runtime_error(USAGE);
6      double  a1(cstr2double(argv[1]));
7      double  a2(cstr2double(argv[2]));
8      if (a2==0)
9        throw std::runtime_error(DIVIDE_BY_ZERO);
10     std::cout << a1/a2 << std::endl;
11     return 0;
12   }
13   catch (std::exception& e) {
14     // reference preserves e's "real" type
15     std::cerr << e.what() << std::endl;
16     return 1; // error return
17   }
18 }
```

Easier to understand **and** more reusable parts.

a bad decomposition

```
1  #include <iostream>
2  #include <string>
3  #include <stdexcept> // for standard exception classes
4  #include <stdlib.h> // for strtod(char*, char**)
5
6  static const std::string
7    USAGE("quotient number number");
8  static const std::string
9    FORMAT_ERR("not a number");
10 static const std::string
11   DIVIDE_BY_ZERO("cannot divide by 0");
```

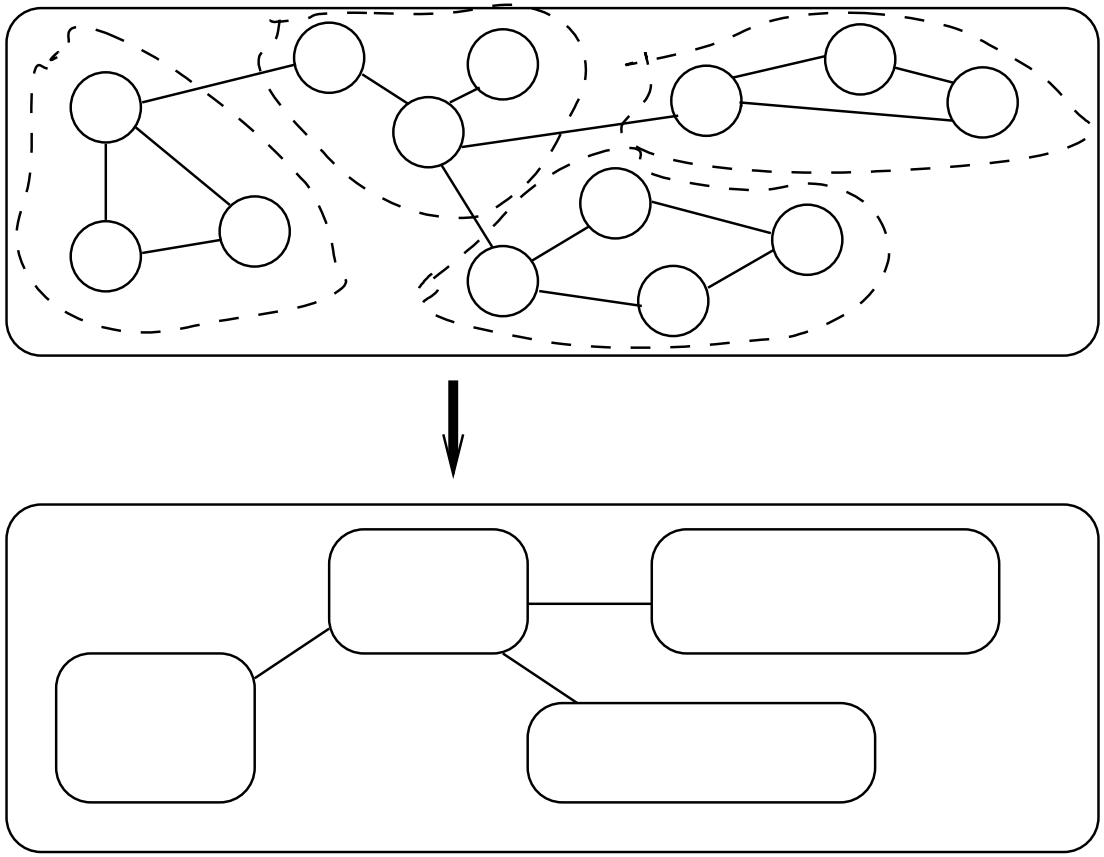
a part that is less reuseable

```
1  // Get two doubles from two C strings in an array.
2  bool
3  get_arguments(char* args[], double& arg1, double& arg2) {
4    char *end; // see the man page for strtod
5
6    arg1 = strtod(args[0], &end);
7    if (end==args[0]) {
8      std::cerr << "\"" << args[0] << "\": "
9        << FORMAT_ERR << std::endl;
10     return false;
11   }
12
13   arg2 = strtod(args[1], &end);
14   if (end==args[1]) {
15     std::cerr << "\"" << args[1] << "\": "
16       << FORMAT_ERR << std::endl;
17     return false;
18   }
19   return true;
20 }
```

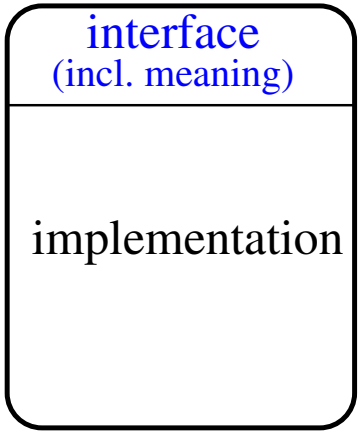
```
1  int
2  main(int argc, char* argv[]) {
3    if (argc!=3) {
4      std::cerr << "usage: " << USAGE << std::endl;
5      return 1; // program failed
6    }
7    double a1;
8    double a2;
9    if (!get_arguments(argv+1, a1, a2))
10     return 1; // program failed
11   if (a2==0) {
12     std::cerr << DIVIDE_BY_ZERO << std::endl;
13     return 1; // program failed
14   }
15   std::cout << a1/a2 << std::endl;
16   return 0;
17 }
```

design = decomposition

Decompose such that overall structure (**architecture**) becomes **simpler**, using **abstractions**.



an example abstraction



(a)

```
double cstr2double(const char* s) throw(range_error)
// convert C string to double or throw range_error
 static const FORMAT_ERROR("not a number");
 char* end;
 double d(strtod(s,&end));
 if (s==end)
   throw range_error(string(s)+": "+FORMAT_ERR;
 return d;
```

(b)

An abstraction has

- An **interface** which is as simple as possible and which hides
- a possibly complex **implementation**.

C++ abstraction mechanisms

- **Functions** abstract **behavior**.
- **Classes** abstract **data + behavior**.
- **Templates** abstract structurally similar skeleton data and/or behaviors.
- **Overloading** abstracts different behavior with same “meaning”.
- **Inheritance** abstracts common interface for related concepts.

components and modules

Several functions and/or classes may be needed to represent a single abstraction. A **component** is such a collection. A **module** is the physical representation of a component: typically a header file with the interface(s) and a collection of source files containing the implementation.

Example

class AccountDatabase,
class AccountDatabase::iterator.

Example

class Rational,
Rational operator+(const Rational&, const Rational&)

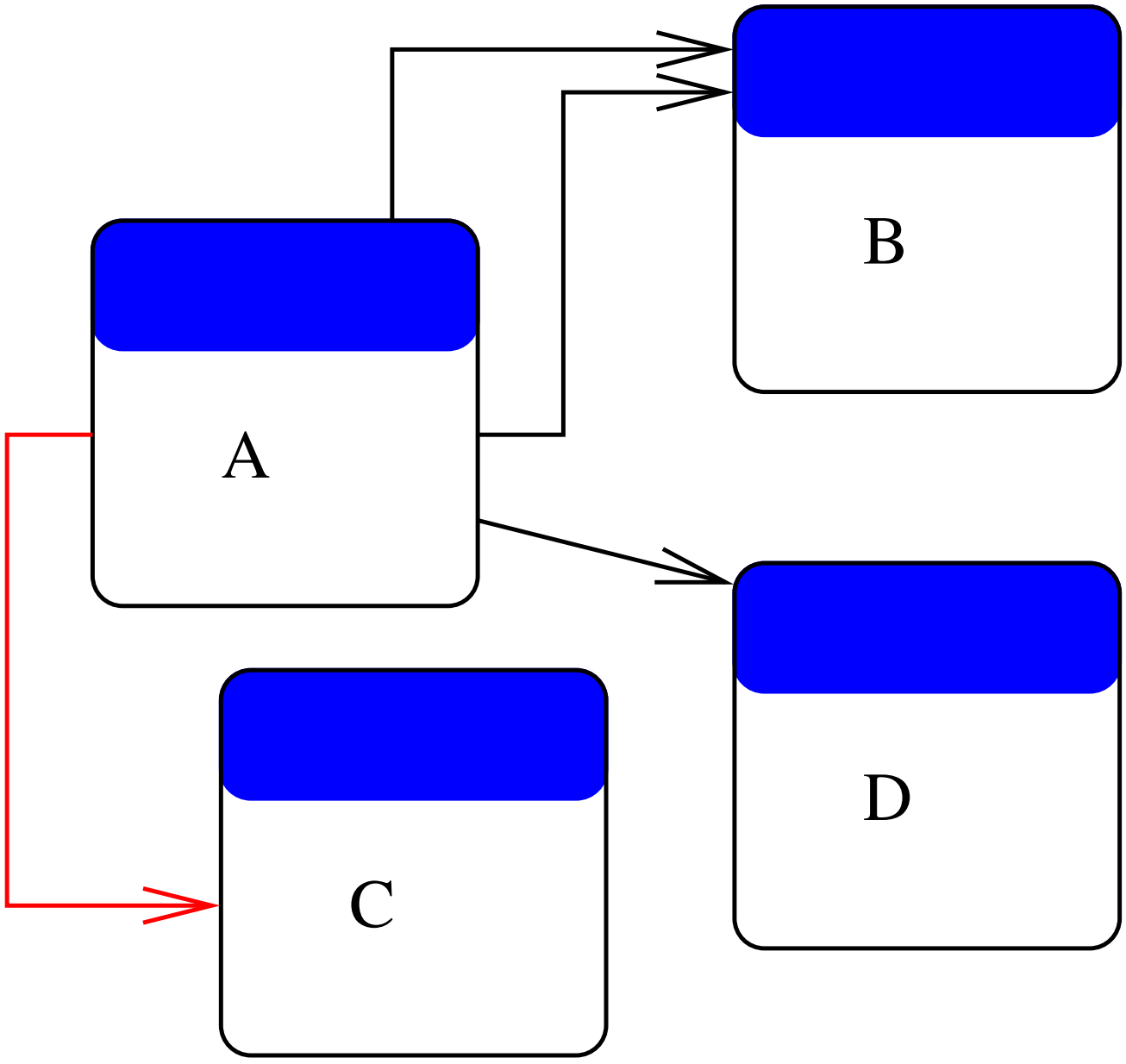
dependencies between abstractions

- **Interface** dependency: when the interface of an abstraction depends on another abstraction, e.g. a function depends on the (class) type of its parameters.
- **Implementation** dependency: when the implementation of an abstraction depends on another abstraction, e.g. a function's body may call other functions.

$\Rightarrow$ Ideal for ease of understanding and reuse:

- Minimize dependencies.
- **Only depend on interface** of other abstractions (**encapsulation**).
- The interface should have fewer dependencies than the implementation.

dependencies between abstractions



interface dependencies in C++

Mechanism	Interface dependencies
function	types of parameters and of thrown exceptions
class	types of public members (functions and data)
publicly derived class	as above, plus the interface, implementation and dependencies of the base class
base class	as above, plus the implementation of any pure virtual functions by its derived classes
function template	types of non-template parameters and thrown exceptions, abstract types of template parameters
class template	non-template types of public members, abstract types of template parameters

inheritance dangers: BookCollection

```
1  class BookCollection {
2    public: // stuff omitted
3      typedef std::set<Book>::const_iterator iterator;
4
5      iterator  begin() const { return books_.begin(); }
6      iterator  end() const { return books_.end(); }
7
8      virtual bool add(const Book& book) {
9        return books_.insert(book).second;
10     }
11
12     virtual void merge(const BookCollection& collection) {
13       for (iterator i=collection.begin();
14            i!=collection.end(); ++i)
15         add(*i);
16     }
17   private:
18     std::set<Book>  books_;
19 };
```

TrackedBookCollection

A BookCollection that keeps statistics on the number of additions.

```
1  class TrackedBookCollection: public BookCollection {
2    public: // stuff omitted
3      int statistics() const { return n_additions_; }
4
5      virtual bool add(const Book& book) {
6        bool ok(BookCollection::add(book));
7        if (ok) // keep count in n_additions_
8           ++n_additions_;
9        return ok;
10     }
11   private:
12     int  n_additions_; // number of books added to collection
13 };
```

more efficient BookCollection

```
1  class BookCollection {
2    public: // stuff omitted
3      virtual bool add(const Book& book) {
4        return books_.insert(book).second;
5      }
6      virtual void merge(const BookCollection& collection) {
7        // more efficient: set<T> bulk insert
8        books_.insert(collection.begin(), collection.end());
9      }
10   private:
11     std::set<Book> books_;
12 };
```

- Now TrackedBookCollection::statistics() has different meaning!
- TrackedBookCollection depends on implementation of BookCollection::merge

commonalities and variabilities

- An abstraction can be regarded as representing the set of its **instances**. E.g. a function represents all its calls, a class all its instance objects, a template all its instantiations.
- Each abstraction supports the specification of certain **commonalities** over its instances as well as **variabilities** that vary with the instantiation.

what to use when (1/2)

Commonality	Variability	C++ feature
function name and behavior, parameter types	parameter values	function
function name and semantics	everything else	overloaded function definitions
function name and behavior	everything else, e.g. parameter types	function template
precise behavior of operations available for an object and data structure of an object	actual data member values (“state”) representing an object	class

what to use when (2/2)

Commonality	Variability	C++ feature
name and semantics (including type) of the related operations available on an object and (possibly) some data structure	everything else	abstract class and inheritance
precise behavior of operations available for an object and “template” data structure of an object	actual types used in the data structure and the operations	class template

negative variability

C++ has mechanisms to support **negative variability**, i.e. certain instances of the abstraction differ w.r.t. some commonalities:

- overloading
- template specialization
- function overriding in derived classes

sources for abstractions

Abstractions may “come from”:

- **problem space**, i.e. the specifications of the application. E.g. `Customer`, `Account`.
- **solution space**, i.e. the implementation techniques used to implement the system. E.g. `Thread` for a multi-threaded system, container classes etc.

a good abstraction:

- is **non-trivial**
- is **abstract**
- has **high cohesion**
- has **low coupling**

non-trivial

```
double add6percent(double x) { return x * 1.06; }
```

is too trivial but

```
double
add_vat(double x) {
  static const double VAT_RATE(6.0);
  return x * (100 + VAT_RATE) / 100;
}
```

may be ok.

abstract

More abstract entities have a better chance of being reusable.

```
1  class Person {  // lots of stuff omitted
2    public:
3      Date  birth_date() const;
4  };
5
6  class Student: public Person {
7    // lots of stuff omitted
8  };
9
10 int age(const Student& p) {
11   return Date::now() - p.birth_date()
12 }
```

increasing abstraction

- More abstract:

```
1  int age(const Person& p) {
2    return Date::now() - p.birth_date();
3  }
```

- Even more abstract:

```
1  template <class ThingWithBirthDate>
2  int age(const ThingWithBirthDate& t) {
3    return Date::now() - t.birth_date();
4  }
```

more abstract is often more powerful

```
1  // product: write product of arg1, arg2, arg3
2  static const std::string  USAGE("usage: product num num num");
3
4  int
5  main(int argc, char* argv[]) {
6    try {
7      if (argc!=4) throw std::runtime_error(USAGE);
8      double  a1(cstr2double(argv[1]));
9      double  a2(cstr2double(argv[2]));
10     double  a3(cstr2double(argv[3]));
11     std::cout << a1*a2*a3 << std::endl;
12     return 0;
13   }
14   catch (std::exception& e) {
15     // reference preserves e's "real" type
16     std::cerr << e.what() << std::endl;
17     return 1; // error return
18   }
19 }
```

```
1  #include <algorithm> // for transform
2  #include <numeric> // for accumulate
3  #include <vector>
4
5  static const std::string USAGE("usage: product [number]..");
6
7  int
8  main(int argc, char* argv[]) {
9    try {
10     std::vector<double> args(argc-1);
11     std::transform(argv+1, argv+argc,
12                    args.begin(), cstr2double);
13     std::cout
14       << std::accumulate(args.begin(), args.end(),
15                          1.0, multiplies<double>())
16       << std::endl;
17     return 0;
18   }
19   catch (std::exception& e) {
20     std::cerr << e.what() << std::endl;
21     return 1; // error return
22   }
23 }
```

high cohesion

- a **function** should do only 1 thing (and do it well)

$\Rightarrow$ **functional cohesion**

- a **class** should encapsulate data that are closely related and all necessary operations on these data (as member or friend functions)

$\Rightarrow$ **data cohesion**

low cohesion example

```
1  class Person { // stuff omitted
2    public:
3      Person(const std::string& name, int yr, int mo, int dy);
4      std::string  name() const;
5      std::string  birth_date(const std::string& format) const;
6    private:
7      std::string  name_;
8      int  birth_year_;
9      int  birth_month_;
10     int  birth_day_;
11 };
12
13 Person lisa("Lisa", 1980, 12, 1);
14 std::cout << lisa.birth_date("%d %b, %Y");
15 // prints "1 december, 1980"
```

higher cohesion example

```
1  class Date { // stuff omitted
2    public:
3      Date(int year, int month, int day);
4      std::string  str(const std::string& format) const;
5      int  day_of_week() const;
6    private:
7      int  year_;
8      int  month_;
9      int  day_;
10 };
11
12 class Person { // stuff omitted
13   public:
14     Person(const std::string& name, const Date& d);
15     std::string  name() const;
16     const Date&  birth_date() const { return birth_date_; }
17   private:
18     std::string  name_;
19     Date  birth_date_;
20 };
```

low coupling, mimimize dependencies

(**bad**) **representational** coupling: e.g. (member) function directly accessing a data member of another class.

$\Rightarrow$ always declare data members **private**

$\Rightarrow$ use accessor functions, if possible also within member functions

(**bad**) **global** coupling: e.g. dependence on global variable

$\Rightarrow$ never use global variables

(**ok**) **parameter** coupling

$\Rightarrow$ function uses only its parameter objects

(**bad**) **control** coupling

(**bad**) **derived class** coupling

control coupling

Caller explicitly determines flow of control in function, e.g. by passing a “flag”.

```
1  class Database {
2    public: // lots of stuff omitted
3      bool store(bool open_first, const Tuple& tuple) {
4        if (open_first) {
5          // open database
6        }
7        // store tuple
8      }
9  };
```

control coupling, how to avoid

```
1  class Database {
2    public:
3      bool open(const std::string& name);
4
5      // returns true iff database has been opened
6      bool is_open() const;
7
8      bool store(const Tuple& tuple) {
9        if (!is_open())
10         return false;
11       // store tuple
12     }
13 };
```

derived class coupling vs composition

- Derivation causes mutual dependencies between base and derived classes.

```
1  class Person: public Date {
2    private:
3      std::string name_;
4  };
```

$\Rightarrow$ use **composition** unless there is a clear **is-a** relationship between derived and base.

```
1  class Person {
2    private:
3      std::string  name_;
4      Date  date_of_birth_;
5  };
```

- Private (or protected) derivation does not commit the public interface of a derived class.

design

- **iterative**: analyze $\rightarrow$ design $\rightarrow$ implement $\rightarrow$ evaluate $\rightarrow$ analyze $\rightarrow \ldots$
- design steps:
  1. find abstractions
     - 1.1 distribute desired functionality over **domain classes** that provide **services**, possibly in collaboration with (objects of) other classes.
     - 1.2 add solution space classes to support the work of the domain classes (e.g. containers).
  2. **refactor**: improve by introducing more general and reusable abstractions; e.g. fuse into template, introduce common base class, ...