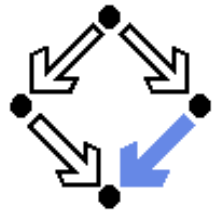
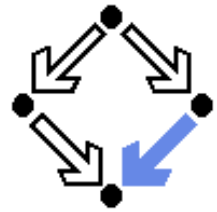
Classes and Objects

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

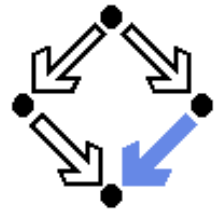


Classes as Record Types



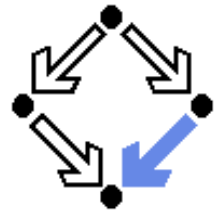
```
class Date {
  public:                       // access specifier
    int day;
    char *month;
};

Date date;                      // an object
date.day = 24;
date.month = "December";

Date *dptr = new Date;          // a pointer to an object
dptr->day = 1;
dptr->month = "January";
delete dptr;
```

The keywords `struct` and `class` mean (almost) the same; however, class values are called "objects" (rather than "structures").



1. **Classes as Namespaces**
2. Classes as Object Types
3. Objects with Functions
4. Objects and Arrays
5. Objects and Information Hiding
6. The Standard Class `string`

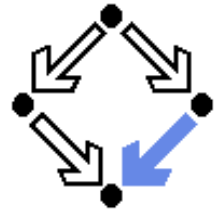
Classes as Namespaces



```
// Date.h
class Date
{
  ...
  // declarations of static members

  // data members
  static const int thisDay = 1;
  static char* thisMonth;

  // member functions
  static Date* create() {
    Date* d = new Date;
    d->day = thisDay;
    d->month = thisMonth;
    return d;
  }

  static void print(Date *date);
};
```

```
// Date.cpp
#include <iostream>
#include "Date.h"

// definitions of static data members
const int Date::thisDay;
char* Date::thisMonth = "January";

// definitions of static member functions
void Date::print(Date *date) {
  std::cout << date->day << "/"
    << date<-month;
}

// Main.cpp
#include "Date.h"
int main(int argc, char* argv[]) {
  Date::thisMonth = "February";
  Date* d = Date::create();
  Date::print(d);
  return 0;
}
```

Classes as Namespaces

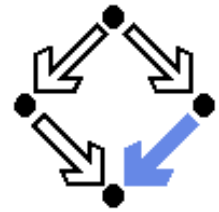


- Classes can serve as namespaces.
  - Static data members and member functions are "bound" to a class.
    They are also called class variables and class functions.
  - There exists only one instance of the static members (independently of the number of objects of the class to which the members belong).
- Names of static members must be qualified by the class name.

  Class`::`*member*
  - Other static members in the same class may use the short name.
- A static data member is only declared in the class.
  - Must have a corresponding definition/initialization somewhere else.
    - Constant members of integral types may be initialized in class.
- A static member function may be also defined in the class.
  - Then the definition may be inlined at the point of every application.
  - If not, there must exist a corresponding definition somewhere else.
- Static member definitions outside of class must not use `static`.
  - For global variables/functions, `static` means "internal linkage".

Use static members rather than variables/functions on namespace level.

File Organization



Typically there are two files related to a class *Class*.

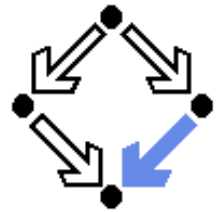
- File *Class*.h contains the class definition.
  - Contains declarations of all members of a class.
  - Must be included by every other file that wants to access members.

    ```
    #include "Class.h"
    ```

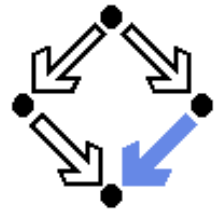
  - If the class declaration changes, all files that include *Class*.h must be recompiled.
- File *Class*.cpp contains the corresponding member definitions.
  - Must include *Class*.h
  - If some member definition changes, only this file must be recompiled (and the program must be relinked).

Any change to a member function that is defined in a class declaration may cause a lot of recompilations.

File Organization

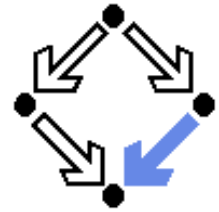


```
// C.h: declaration of C and its members
#ifndef C_H_
#define C_H_
class C {
  static T x;              // declaration, no definition
  static T f(...);         // declaration, no definition
  static T g(...) { ... } // declaration with definition
}
#endif /* C_H_ */

// C.cpp: definitions of members of C
#include "C.h"
T C::x;
T C::f(...) { ... }

// Main.cpp: use of C
#include "C.h"
int main() { ... C::x ... C::f(...) ... C::g( ...) ... }
```



1. Classes as Namespaces
2. **Classes as Object Types**
3. Objects with Functions
4. Objects and Arrays
5. Objects and Information Hiding
6. The Standard Class `string`

Nonstatic Data Members



```
class Date
{
  public:
    int day;
    char *month;
};
```

- Non-static data members "belong to" an object of the class.

  They are also called object variables.

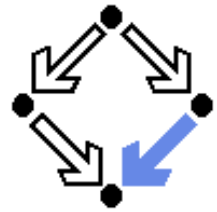
  - For every object of the class, the exists a separate instance of the object variable.
- Names of nonstatic data members must be qualified by an object.

  object.member

  objectptr->member

Data members with access specifier `public` can be freely used like the variables of a structure.

Constructors



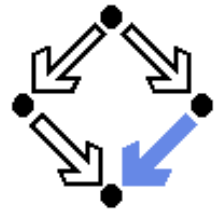
```
class Date // Date.h
{
  ...
  // inline declaration
  Date(int d, char *m)
  {
    day = d;
    month = m;
  }
};
```

```
class Date // Date.h
{
   ...
   Date(int d, char *m);
};

Date::Date(int d, char *m) // Date.cpp
{
  day = d;
  month = m;
}
```

```
Date date(24, "December");              // calls Date(int, char*)
Date *dptr = new Date(26, "October"); // calls Date(int, char*)
```

A constructor is a method that initializes an object's data members.

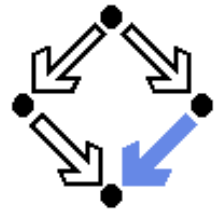
Constructors



- A constructor is a special method that is declared in a class.
  - The constructor has the same name as the class.
  - It has no return type (also not `void`).
- A constructor is bound to an object.
  - Called after the space for the object has been allocated.
  - Executed in the context of the object.
    - Can access all data members without qualification.
- There may be multiple constructors with different argument types.
  - Same constraints as for function overloading.
- If defined inside the class, the constructor is inlined.
  - Same effect as if using the keyword `inline`.
- A constructor may execute arbitrary code.
  - Not only initializations of data members.

Objects should be initialized by calling constructors.

The `this` Pointer

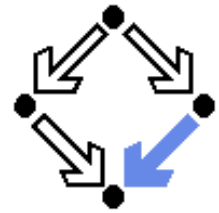


```
class Date // Date.h
{
   ...
   Date(int day, char* month);
};

Date::Date(int day, char *month) // Date.cpp
{
  this->day = day;
  this->month = month;
}
```

- The keyword `this` is a pointer to the current object.
  - Can be used e.g. inside the body of a constructor.
  - Can be used e.g. to resolve name ambiguities.

We will see later the general rules for the use of `this`.

The Default Constructor



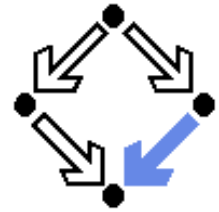
```
class Date // Date.h
{
   ...
   Date() {
     day = 1;
     month = "January";
   }
};
```

```
class Date // Date.h
{
   ...
   Date();
};

Date :: Date() { // Date.cpp
  day = 1;
  month = "January";
}
```

```
Date date;                    // calls Date()
Date date();                  // WRONG: declares function date()
Date *dptr1 = new Date;       // calls Date()
Date *dptr2 = new Date();     // calls Date()
Date darray[10];              // calls Date() for each object
Date *darr = new Date[10];    // calls Date() for each object
```

All objects are initialized with the default constructor of their class.

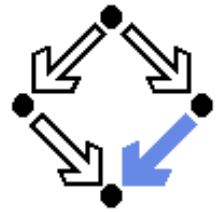
The Default Constructor



- A default constructor can be called without arguments.
  - Is called for object declarations without initializers.
  - Is called for initialization of array elements.
  - Is called for initialization of non-static data members before a user-defined constructor is called.
  - Is called for initializing static data members when program is started.
- If a class has no user defined constructor, an implicit default constructor is automatically generated.
  - Calls the default constructors of all non-static data members.
- If a class has user defined constructors, only these may be called.
  - If some constructor is defined, also a default constructor has to (respectively should) be explicitly defined.
  - Without a default constructor, it is not possible to declare a variable of this type without initialization (and thus no arrays with this base type can be created).

All objects are automatically initialized by constructors.

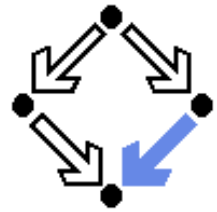
Initialization Lists



```
class Date // Date.h
{
   ...
   Date(int d, char* m);
};

// explicit initialization of non-static data members
Date::Date(int d, char *m): day(d), month(m)
{
  // executed after data members have been initialized
   ...
}
```

The preferred way of initializing non-static data members (avoids calling their default constructors).

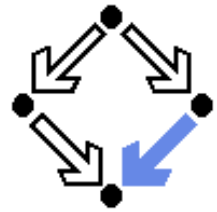
The Copy Constructor



```
class Date // Date.h
{
   ...
   Date(const Date date&);
};

// the copy constructor
Date::Date(const Date &date): day(date.day), month(date.month) { }

static void print(Date date);

Date date0;          // calls default constructor
Date date1(date0);   // calls copy constructor
Date date2 = date0; // calls copy constructor
print(date0);        // calls copy constructor
return date0;        // calls copy constructor
```

An object duplicate is created with the copy constructor of the class.

The Copy Constructor

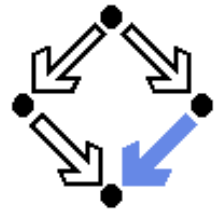


- The copy constructor of a class can be called with a reference to an object of this class as argument.
  - Is called in variable initializations

    Here the token "=" does here not denote assignment.
  - Is called when passing objects as function arguments.
  - Is called when returning objects as function results.
- If a class has no copy constructor, an implicit copy constructor is automatically generated.
  - Calls the copy constructors of all non-static data members.

Object duplication can be controlled by the programmer.

Implicit Conversions

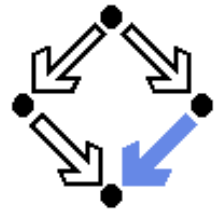


Constructors may be also called in unexpected situations.

```
class Complex
{
   // constructors may be also provided with default arguments
   Complex(double re = 0.0, double im = 0.0);
   Complex add(Complex c);
};

Complex a;             // calls Complex(0, 0)
Complex b = 1;         // calls Complex(1, 0)
Complex c = b.add(2); // calls Complex(2, 0) to create argument object
```

Constructors are also implicitly called to perform type conversions.

Preventing Implicit Constructor Calls

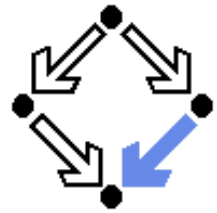


Implicit constructor calls may be unwanted.

```
class Complex
{
   // constructor declaration with keyword "explicit"
   // constructors may be also provided with default arguments
   explicit Complex(double re = 0.0, double im = 0.0);
   Complex add(Complex c);
};

Complex a;             // calls Complex(0, 0)
Complex b = 1;         // ERROR
Complex c(1);          // calls Complex(1, 0)
Complex c = b.add(2); // ERROR
Complex c = add(static_cast<Complex>(2)); // calls Complex(2, 0)
```

With keyword `explicit`, unexpected constructor calls can be avoided.

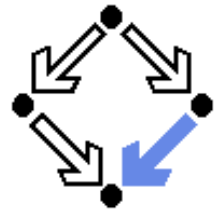
The Copy Assignment Operator



```
class Date // Date.h
{
   ...
   Date& operator=(const Date &date);
};

// the copy assignment operator
Date& operator=(const Date& date)
{
  day = date.day;
  month = date.month;
  return *this;
}

Date date0(4, "July"); // calls Date(int, char*)
Date date1(date0);     // calls copy constructor
date1 = date0;         // calls copy assignment operator
```

Object assignment is performed with the copy assignment operator.

The Copy Assignment Operator

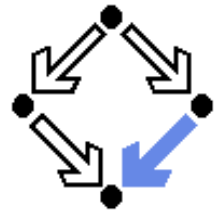


- The copy assignment operator of a class can be called with a reference to an object of this class as argument.
  - Is called in object assignments.

    Not in object initializations!
- If a class has no copy assignment operator, an implicit copy assignment operator is automatically generated.
  - Calls the copy assignment operators of all non-static data members.

Destructive object assignment can be controlled by the programmer.

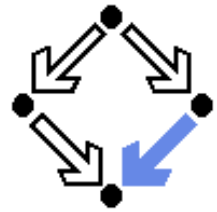
The Destructor



```
class Date {
   ...
   Date(int d, char *m);
   ~Date();
};

Date::Date(int d, char *m): day(d), month(new char[100]) {
  strncpy(month, m, 100);
}

// the destructor
Date::~Date() { delete[] month; }

{
  Date date(24, "December");        // calls Date(int, char*)
}                                   // calls destructor
Date *dptr = new Date(14, "July"); // calls Date(int, char*)
delete dptr;                        // calls destructor
```

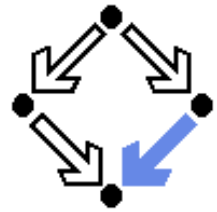
Objects are destroyed by the destructor of the class.

The Destructor



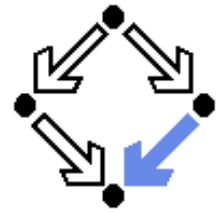
- The destructor of a class can be called without arguments.
  - Is called on local variable when declaration scope is left.
  - Is called on dynamically allocated objects when `delete` is called.
  - Is called on statically allocated objects when program is terminated.
  - Is called on every array element, if an array is destroyed.
- If a class has no destructor, an implicit destructor is generated.
  - Calls the destructors of all non-static data members.

Object destruction can be controlled by the programmer.



1. Classes as Namespaces
2. Classes as Object Types
3. **Objects with Functions**
4. Objects and Arrays
5. Objects and Information Hiding
6. The Standard Class `string`

Non-Static Member Functions

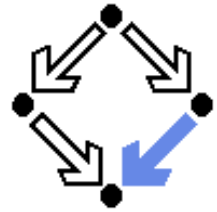


```
class Date // Date.h
{
  ...
  void print();
  void set(int d, char* m);
  int getDay() const;
  char* getMonth() const;
};

// main program
Date date(1, "January");
date.print();
date.set(2, "January");
Date *dptr = &date;
dptr->print();
dptr->set((dptr->getDay())+1, dptr->getMonth());
```

```
// Date.cpp
#include <iostream>
#include "Date.h"

void Date::print() {
  std::cout << day << " " << month;
}

void Date::set(int d, char* m) {
  day = d; month = m;
}

int Date::getDay() const { return day; }

char* Date::getMonth() const { return month; }
```

Objects can have member functions “attached”.

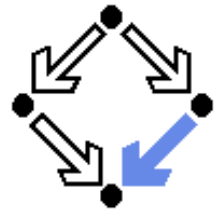
Non-Static Member Functions



- A non-static member function is "bound to" a particular object.
  Such a function is also called an object function.
  - Every object of a class has an instance of this function attached.
- The function operates "within" the object to which it is bound.
  - It can access all non-static data members of the object without object qualification (and also all static data members of the class without class qualification).
  - The data members are "global variables" for the function.
- A member function declared as `const` does not change the object.
  - May be called on objects declared as `const`.
- Constructors/destructors/copy assignment operators are special cases of non-static member functions.

With non-static member functions objects can be used without exposing their internal representation.

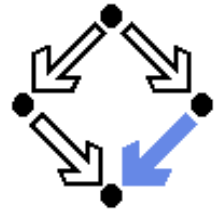
Calling Non-Static Member Functions



```
class Date // Date.h
{
  ...
  void nextDay();
};

void Date::nextDay() {
  int d = getDay();
  char* m = getMonth();
  set(d+1, m); // stupid, of course
}

Date date(28, "February");
date.nextDay();
```

The non-static member functions of a class may call each other without object qualification; then they refer to the same object.

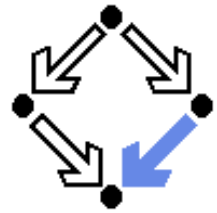
Calling Non-Static Member Functions



```
class Date // Date.h
{
  ...
  void copyDay(Date date);
};

void Date::copyDay(Date date) {
  int d = date.getDay();
  char* m = getMonth();
  set(d, m);
}

Date date1(28, "February");
Date date2(15, "March");
date1.copyDay(date2);
```

To call the member function of another object, explicit qualification by the object is necessary.

The `this` Pointer



- In a non-static context, `this` is a pointer to the "current" object.
  - Initialization expressions of non-static data members.
  - Bodies of constructors/destructors/non-static member functions.
- Any plain (unqualified) reference to a non-static member is implicitly extended to a qualified reference by adding `this`.

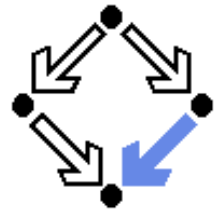
  var → `this->`*var*

  fun(...) → `this->`*fun*(...)
  - From static contexts, such unqualified references are not allowed.
- For a non-static data member, `this` is just the address of the structure in which the member is looked up.
- For a non-static member function, `this` is just an additional parameter whose value is provided by the caller.

  fun(...) { ... } → *fun*(`this`, ...) { ... }

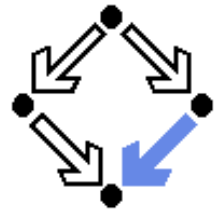
  object.*fun*(...) → *fun*(&*object*, ...)

The difference between static and non-static members is just the availability of the `this` pointer.



1. Classes as Namespaces
2. Classes as Object Types
3. Objects with Functions
4. **Objects and Arrays**
5. Objects and Information Hiding
6. The Standard Class `string`

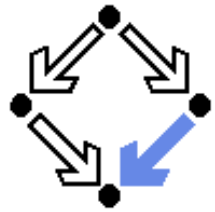
Objects as Array Elements



```
// default constructions, then creation and assignments of new objects
Class a[N]; // objects initialized by default constructor
for (int i=0; i < N; i++) a[i] = Class(...);

// new objects initialized by arbitrary constructor
Class* b[N]; // uninitialied pointers
for (int i=0; i < N; i++) b[i] = new Class(...);

// default constructions, then creation and assignments of new objects
Class *c = new Class[N];  // objects initialized by default constructor
for (int i=0; i < N; i++) c[i] = Class(...);

// new objects initialized by arbitrary constructor
Class* *d = new Class*[N]; // uninitialized pointers
for (int i=0; i < N; i++) d[i] = new Class(...);
```

Be sure to have array objects appropriately initialized.

Example: Phone Book

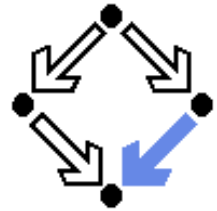


Write a program that reads a sequence of at most N phone book entries consisting of a name and a phone number. The program then reads sequences of names and prints the corresponding phone numbers.

```
const int N = 100;
void mainPhoneBook()
{
  Entry* book[N];
  int n = readPhoneBook(book, N);
  usePhoneBook(book, n);
  deletePhoneBook(book, n);
}
```

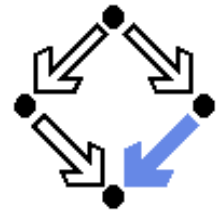
Central data structure is an array of (pointers to) objects.

A Phone Book Entry



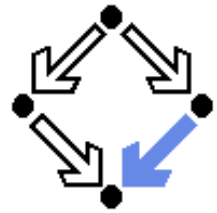
```
class Entry {
  public:
    const char *name;
    const char *number;

    Entry() { }
    Entry(char* na, char *nu): name(copy(na)), number(copy(nu)) { }
    ~Entry() { delete[] name; delete[] number; }

    const char* getName() const { return name; }
    const char* getNumber() const { return number; }
};

static const char* copy(char *str) {
  int n = strlen(str);
  char* result = new char[n+1];
  strncpy(result, str, n+1);
  return result;
}
```

Constructing a Phone Book



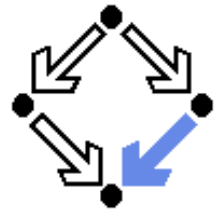
```
int readPhoneBook(Entry **book, int N)
{
  for (int i=0; i<N; i++)
  {
    Entry *entry = readEntry();
    if (entry == NULL) return i;
    book[i] = entry;
  }
  return N;
}

Entry* readEntry()
{
  cout << "Another entry (y/n)? ";
  char ch[2]; cin.getline(ch, 2); if (ch[0] != 'y') return NULL;
  cout << "Name: "; char name[100]; cin.getline(name, 100);
  cout << "Number: "; char number[100]; cin.get(number, 100);
  return new Entry(name, number);
}
```

Using a Phone Book



```
void usePhoneBook(Entry** book, int n) {
  while (true) {
    cout << "Another lookup (y/n)? ";
    char ch[2]; cin.getline(ch, 2); if (ch[0] != 'y') return;
    cout << "Name: "; char name[100]; cin.get(name, 100);
    const char *number = getNumber(book, n, name);
    if (number == NULL) { cout << "Name not found\n"; continue; }
    cout << "Number: " << number << "\n";
  }
}

const char* getNumber(Entry **book, int n, char *name) {
  for (int i=0; i<n; i++) {
    Entry *entry = book[i];
    if (strcmp(name, entry->getName()) == 0)
      return entry->getNumber();
  }
  return NULL;
}
```

Deleting a Phone Book



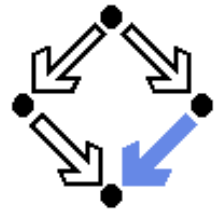
```
void deletePhoneBook(Entry **book, int n)
{
  for (int i=0; i<n; i++)
    delete book[i];
}
```

Solution has various disadvantages:

- Fixed maximum N of number phone book entries.
- Number of actual entries n has to be passed around.
- No abstraction from representation of phone book as `Entry**`.
- Application directly operates on phone book entries.
- Explicit deallocation of allocated phone book entries.

Not yet a really "object-oriented" solution.

Objects Containing Arrays

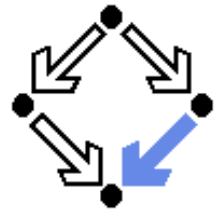


A "naked" array is usually not sufficient to represent program data.

- Typically a data structure consists of various bits and pieces.
- All these should be packaged together into a single object.
- Object provide via methods "high-level" access to its data.

Typically arrays are just part of an object representation.

Prime Number Computation

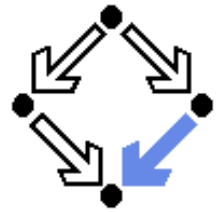


Print the first n prime numbers.

```
void printPrimes(int n)
{
  if (n < 2) return;
  cout << 2 << "\n";
  PrimeTable p; // table to hold all odd primes computed so far
  for (int c = 3; c < n; c += 2)
  {
    if (!p.isPrime(c)) continue;
    cout << c << "\n";
    p.add(c);
  }
}
```

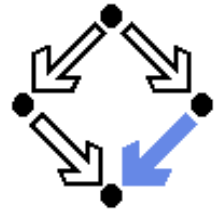
Core functionality is packed into a “prime table”.

A Prime Table



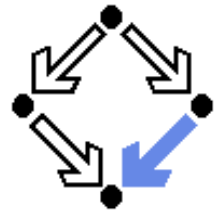
```
class PrimeTable {
  public:
    int N;  // the size of the table
    int n;  // the number of elements actually contained in it
    int *p; // the table itself

    PrimeTable();          // create the table
    ~PrimeTable();         // delete the table

    bool isPrime(int c); // check whether candidate c is prime
    void add(int c);     // add a new prime c to the table

    void resize();         // make table bigger
};

PrimeTable::PrimeTable(): N(100), n(0), p(new int[N]) { }
PrimeTable::~PrimeTable() { delete[] p; }
```

A Prime Table



```
bool PrimeTable::isPrime(int c) {
  for (int i=0; i<n; i++)
    if (c % p[i] == 0) return false;
  return true;
}

void PrimeTable::add(int c) {
  if (n == N) resize();
  p[n] = c;
  n = n+1;
}

void PrimeTable::resize() {
  int N0 = 2*N+1;
  int *p0 = new int[N0];
  for (int i=0; i<n; i++) p0[i] = p[i];
  delete[] p;
  N = N0; p = p0;
}
```

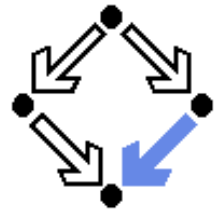
The Phone Book Revisited



Now let us make the phone book really "object-oriented".

```
void mainPhoneBook()
{
  PhoneBook book;       // empty book is created by default constructor
  readPhoneBook(book); // book is filled
  usePhoneBook(book);  // book is used
}                       // book is destroyed by destructor

void readPhoneBook(PhoneBook &book) {
  while (true)
  {
    cout << "Another entry (y/n)? ";
    char ch[2]; cin.getline(ch, 2); if (ch[0] != 'y') return;
    cout << "Name: "; char name[100]; cin.getline(name, 100);
    cout << "Number: "; char number[100]; cin.getline(number, 100);
    book.add(name, number); // an entry is added to the book
  }
}
```

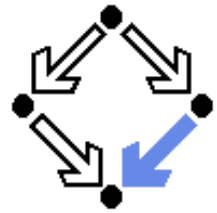
The Phone Book Revisited



```
void usePhoneBook(PhoneBook &book)
{
  while (true)
  {
    cout << "Another lookup (y/n)? ";
    char ch[2]; cin.getline(ch, 2); if (ch[0] != 'y') return;
    cout << "Name: "; char name[100]; cin.getline(name, 100);
    const char *number = book.search(name); // name looked up in book
    if (number == NULL) { cout << "Name not found\n"; continue; }
    cout << "Number: " << number << "\n";
  }
}
```

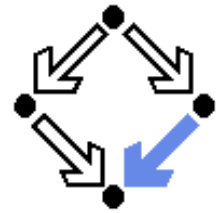
Core functionality is now completely hidden in the phone book.

The New Phone Book



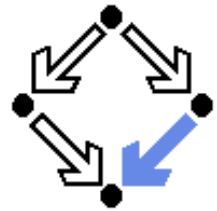
```
class PhoneBook {
  public:
    int N;         // the size of the book
    int n;         // the number of entries allocated in it
    Entry **b;     // the book itself, a table of entry *pointers*

    PhoneBook();   // construct the book
    ~PhoneBook();  // delete the book

    void add(char *name, char *number); // add an entry
    const char* search(char *name);     // search for a number
    void resize();                      // make the book bigger
};

PhoneBook::PhoneBook(): N(100), n(0), b(new Entry*[N]) { }
PhoneBook::~PhoneBook() {
  for (int i=0; i<n; i++) delete b[i]; // delete entry
  delete[] b;                          // delete book itself
}
```

The New Phone Book



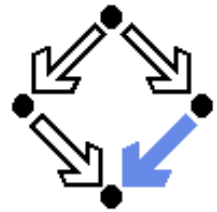
```
void PhoneBook::add(char *name, char *number) {
  if (n == N) resize();
  b[n] = new Entry(name, number);
  n = n+1;
}

const char* PhoneBook::search(char *name) {
  for (int i=0; i<n; i++) {
    Entry *entry = b[i];
    if (entry->hasName(name)) return entry->getNumber();
  }
  return NULL;
}

void PhoneBook::resize() {
  int N0 = 2*N+1; Entry **b0 = new Entry*[N0];
  for (int i=0; i<n; i++) b0[i] = b[i]; // only *pointers* are copied
  delete[] b; N = N0; b = b0;
}
```

The New Phone Book Entry



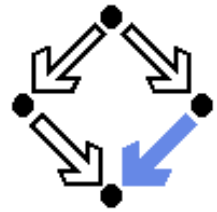
```
class Entry {
  public:
    const char *name;
    const char *number;

    Entry() { }
    Entry(char* na, char *nu): name(copy(na)), number(copy(nu)) { }
    ~Entry() { delete[] name; delete[] number; }

    const char* getName() const { return name; }
    const char* getNumber() const { return number; }
    bool hasName(char *name) const { return strcmp(name, this->name) == 0; }

    static const char* copy(char *str) {
      int n = strlen(str);
      char* result = new char[n+1];
      strncpy(result, str, n+1);
      return result;
   }
};
```

Objects versus Pointer to Objects



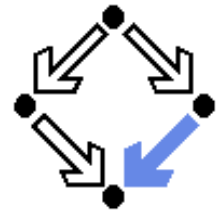
- Object values may become unhandy.
  - It is costly to copy full objects.
  - Objects should be mainly passed to functions by reference.
    - Use of reference parameters in method declarations is recommended.
    - Otherwise the copy constructor is invoked on each function call with an object as argument to create a temporary copy of the object.
- Object pointers are frequently preferred.
  - It is cheap to copy pointers to objects.
  - Objects referenced by pointers should be created on the heap by `new`.
    It is unwise to use pointers to stack-allocated data.
  - However, such objects must be then explicitly destroyed by `delete`.
    - Otherwise "memory leaks" will arise in the program.
    - Destructors of objects must explicitly free the space of all objects referenced by pointers (provided that there exist nowhere else references to these objects, otherwise "dangling pointers" will arise).

If the representation of an object contains dynamically created objects, these objects should be better "hidden" from the outside world.



1. Classes as Namespaces
2. Classes as Object Types
3. Objects with Functions
4. Objects and Arrays
5. **Objects and Information Hiding**
6. The Standard Class `string`

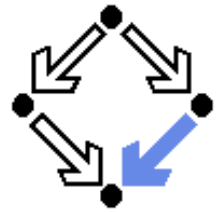
Access Specifiers



- Special labels restrict who can access a member.
  - `public` Anyone can access public members.
  - `protected` Only the class, derived classes, and friends can access protected members.
  - `private` Only the class and friends can access private members.
    - Derived classes and friends will be introduced later.
- Without an access specifier, default access levels are used.
  - `class` Default is `private`.
  - `struct` Default is `public`.
    - While this is actually the only difference between `class` and `struct`, the later is typically used for plain structures only.
- Distinguish between a class's interface and its implementation.
  - Interface: "contract" between user and implementor; members that belong to the interface are declared `public`.
  - Implementation: "internals" of a particular realization; members that belong to the implementation are declared `private` or `protected`.

Explicit access specifiers should be always used.

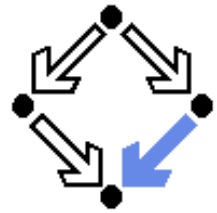
Typical Class Layout



```
class Class {
  private:                          // object representation
    type var ;
    ...
  public:                           // interface constructors/functions
    Class(...) { }
    type fun(...) { ...}
    ...
  private:                          // implementation functions
    type fun(...) { ...}
    ...
 };
```

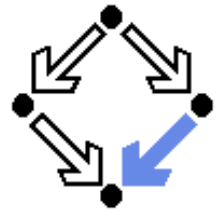
Data members should generally not be declared `public`.

Example



```
class Entry {
  private:
    const char *name;
    const char *number;

  public:
    Entry() { }
    Entry(char* na, char *nu): name(copy(na)), number(copy(nu)) { }
    ~Entry() { delete[] name; delete[] number; }

    const char* getName() const { return name; }
    const char* getNumber() const { return number; }
    bool hasName(char *name) const { return strcmp(name, this->name) == 0; }

  private:
    static const char* copy(char *str) {
      int n = strlen(str);
      char* result = new char[n+1];
      strncpy(result, str, n+1);
      return result;
    }
};
```

Friends



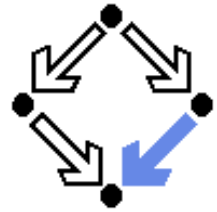
- A class may declare an external entity as `friend`.

```
class Class {
  ...
  friend type fun(...);
  friend type C::fun(...);
  friend class D;
  ...
};
```

- Friend gets full access to all members of *Class*.
  - Function *fun* and member function *C*::*fun* receive friend status.
  - All member functions of *D* receive friend status.
- Friendship is not transitive.
  - A friend of a friend of *Class* is not automatically a friend of *Class*.
- Friendship is not inherited.
  - The concept of "inheritance" will be introduced later.

Controlled break of access rules; use with care!

Nested Classes

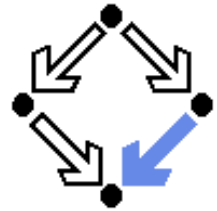


A class definition may contain the definition of another class.

```
class Outer {
  ...
  class Inner {
    ...
  };
  ...
};
```

- Inner class may be externally referred as `Outer::Inner`.
  - Similar to access of static class members.
- However, an inner class also obeys access specifiers.
  - Private inner class can be only used by outer class and its friends.
- The outer and the inner class are not automatically friends.
  - Each class can only refer to the non-public members of the other class, if it is explicitly declared as `friend`.

Nested Classes

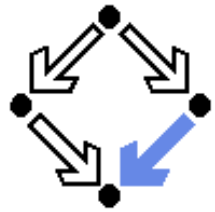


A class definition may contain just the declaration of another class.

```
// Outer.h
class Outer {
  ...
  class Inner;
  ...
};

// Inner.h
class Outer::Inner {
  ...
  type fun(...);
};

// Inner.cpp
type Outer::Inner::fun(...) { ... }
```

If only type `Inner*` is used, definition of `Inner` needs not be included.

Example: Dynamic Lists

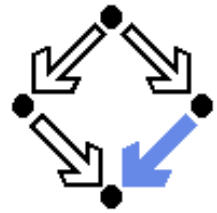


```
// IntList.h
class IntList
{
  class IntNode;

  private:
    IntNode *head;
    int number;

  public:
    IntList();
    ~IntList();
    int length() const;
    IntList& insert(int e);
    int get(int i) const;
};
```

```
#include <iostream>
#include "IntList.h"

using namespace std;

int main()
{
  IntList l;
  l.insert(2).insert(3).insert(5);
  cout << l.length();  // 3
  cout << l.get(2);    // 5
  return 0;
}
```

IntNode is only declared (not defined) in IntList.

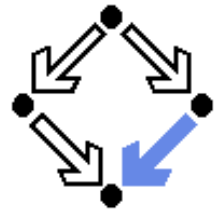
Example: Dynamic Lists



```
// IntList.cpp
#include "IntList.h"
#define NULL (0)

class IntList::IntNode {
  friend class IntList;
  private:
    int value; IntNode* next;
    IntNode(int v, IntNode *n):
    value(v), next(n) { }
}

IntList::IntList():
head(NULL), number(0) { }

IntList::~IntList() {
  IntNode *node = head;
  while (node != null) {
    IntNode *node0 = node->next;
    delete node;
    node = node0;
  }
}
```

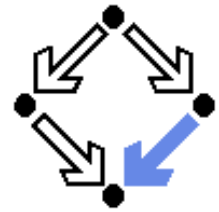
```
int IntList::length() const {
  return number;
}

IntList& IntList::insert(int e) {
  IntNode* node = new IntNode(e, head);
  head = node;
  number = number+1;
  return *this;
}

int IntList::get(int i) const {
  IntNode *node = head;
  for (int j=0; j<number-i-1; j++)
    node = node->next;
  return node->value;
}
```

Frequently used technique called "Pointer to Implementation".



1. Classes as Namespaces
2. Classes as Object Types
3. Objects with Functions
4. Objects and Arrays
5. Objects and Information Hiding
6. **The Standard Class `string`**

The Standard Class `string`



C++ has a much more convenient representation of strings than C.

- C/C++: `char[`*N*`] s`
  - Strings as arrays of characters terminated by the null character.
  - Very rigid representation because each string has a fixed size.
  - Constructing new strings (reading, concatenating, ...) is tedious.
  - There is the persistent danger that the allocated buffer is overwritten.

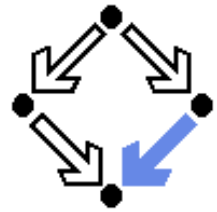
```
// reads up to 4 characters, may cause buffer overflow
char s[3]; cin.getline(s, 4);
```

- C++: `string s (#include <string>)`
  - Strings as objects with associated operations.
  - No fixed length; easy construction and manipulation.
  - Automatic memory management hidden in class.

```
// reads safely text line of arbitrary length
string s; getline(cin, s);
```

Class `string` is strongly recommended for string processing in C++.

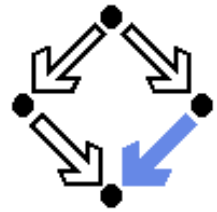
String Input and Output



```
// reads line and places it in s (excluding the end of line marker)
istream& getline(istream& in, string &s);

// reads characters until delim occurs and places it in s (excluding delim)
istream& getline(istream &in, string &s, char delim);

// reads one word (excluding white space) and returns string
istream &operator>>(istream &in, const string& s);

// writes string
ostream &operator<<(ostream &out, const string& s);
```

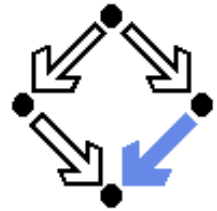
The result of the input operations (converted to type `bool`) is `false`, if and only if no (more) input was available.

Other String Operators



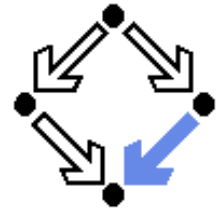
```
// returns concatentation of strings, character sequences, characters
string& operator+(string& s, string& t);
string& operator+(const char*s, const string &t);
string& operator+(const string& s, char* t);
string& operator+(const string& s, char c);

// compares string
// - operator== for equality
// - operator!= for inequality
// - operator<, operator <=, operator >, operator >= for lexical ordering
bool operator==(const string& s, const string& t);
bool operator==(const char* s, const string& t);
bool operator==(const string& s, const char* t);

// swaps contents of strings
void swap(string &a, string &b);
```

Constructors and Basic Access Functions



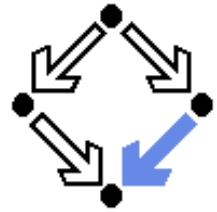
```
// the empty string
string();

// a copy of s
string(const string& s);

// a copy of the null-terminated character sequence s as a string
string(const char* s);

// denotes "no position" in several methods
static const int npos = -1;

// a substring of s starting at pos and having at most n characters
// (for n = npos everything up to the end of the string is copied)
string(const string& s, int pos, int  n = npos);

// a copy of the first n characters of s as a string
string(const char* s, int n);

// number of characters in string and a test for emptiness
int length() const;
int size() const;
bool empty() const;
```

Non-Destructive Member Functions



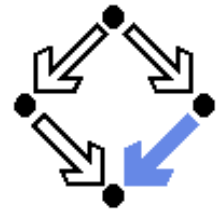
```
// reference to character at position i
char& operator[](int i);

// substring of this string starting at pos with at most n characters
string substr(int pos = 0, int n = npos);

// return string as a null-terminated sequence of characters
// (must not be modified and becomes invalid after modifying this string)
const char* c_str() const;

// copies from this string into the buffer up to n character starting at pos
int copy(char* buffer, int n, int pos = 0);

// compare (a substring of) this string with (a substring of) another string
// 0 if equal, <0 if this string is lexicographically smaller, >0 otherwise
int compare(const string& s) const;
int compare(const char* s) const;
int compare(int pos, int n, const string& s) const;
int compare(int pos, int n, const char* s) const;
int compare(int pos, int n, const string& s, int pos2, int n2) const;
int compare(int pos, int n, const char* s, int pos2, int n2) const;
```

Destructive Member Functions



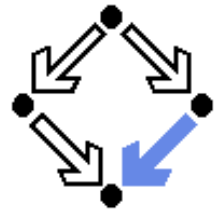
```
// erase characters in string
void clear();

// assign to this string another string, character sequence, or character
string& operator=(const string&s); // also: assign
string& operator=(const char* s);  // also: assign
string& operator=(char c);         // also: assign

// assign to this string the denoted substring to s
string &assign(const string& s, int pos, int n);
string &assign(const char* s, int pos);

// append to this string another string or character
string& operator+=(string &s); // also: append
string& operator+=(char *s);   // also: append
string& operator+=(char c);    // also: append

// append to this string the denoted substring of s
string &append(string &s, int pos, int n);
string &append(char *s, int pos, int n);
```

Member Functions for Inserting/Replacing



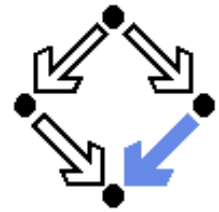
```
// insert into this string at pos another string or character sequence
string& insert(int pos, const string& s);
string& insert(int pos, const char* s);
string& insert(int pos, int n, char c);

// insert into this string at pos the denoted substring
string& insert(int pos, const string& s, int pos2, int n2);
string& insert(int pos, char* s, int n);

// erases denoted substring from this string and inserts other string instead
string &replace(int pos, int n, const string& s);
string &replace(int pos, int n, const char* s);

// insert denoted substring instead
string &replace(int pos, int n, const string& s, int pos2, int n2);
string &replace(int pos, int n, const char* s, n2);

// insert n2 copies of character c instead
string &replace(int pos, int n, int n2, char c);
```

Member Functions for String Searching



```
// search (starting at pos) for smallest position where s occurs in this string
// (npos, if s does not occur in this string)
int find(const string& s, int pos = 0) const;
int find(const char* s, int pos = 0) const;
int find(char c, int pos = 0) const;

// search for substring of s with at most n characters
int find(const char* s, int pos, int n) const;

// search (starting at pos) for largest position where s occurs in this string
// (npos, if s does not occur in this string)
int rfind(const string& s, int pos = npos) const;
int rfind(const char* s, int pos = npos) const;
int rfind(char c, int pos = 0) const;

// search for substring of s with at most n characters
int rfind(const char* s, int pos, int n) const;
```

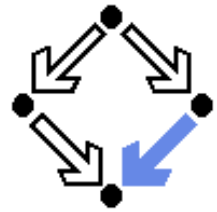
Member Functions for Character Searching



```
// search for smallest position where some chararacter of s occurs
// (does not occur) in this string
int find_first_of(const string&s, int pos = 0) const;
int find_first_of(char *s, int pos = 0) const;
int find_first_not_of(const string&s, int pos = 0) const;
int find_first_not_of(const string&s, int pos = 0) const;

// search for largest position where some chararacter of s occurs
// (does not occur) in this string
int find_last_of(const string&s, int pos = npos) const;
int find_last_of(char *s, int pos = npos) const;
int find_last_not_of(const string&s, int pos = npos) const;
int find_last_not_of(const string&s, int pos = npos) const;
```

For more functions and detailed information, see the class documentation.

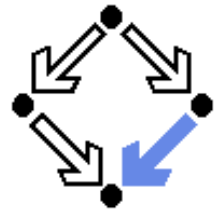
Examples



```
string lower = "abc..z";
string upper = "ABC..Z",
string letters = lower + upper;

cout << letters[26];              // 'A'
cout << letters.length();         // 52
cout << letters.substring(2, 3); // "cde"
cout << letters.find("cde");      // 2
cout << letters.find("xxx");      // npos

letters.insert(0, "<");
letters.append(">");
cout << letters;                  // "<abc..zABC..z>"

String line;                     // empty string
bool okay = getline(cin, line); // read one text line
if (!okay) return;               // check for end of input
```

Flexible construction and manipulation of strings.

Example: Text Processing



Write a program that reads a string which contains multiple words (sequences of letters) separated by other characters. The program then prints all words of the text in separate lines in the order of their occurrence in the text. For instance, the input

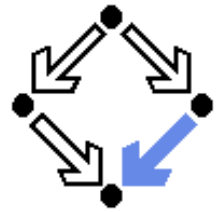
```
One, two, and three!
```

shall result in output

```
One
two
and
three
```

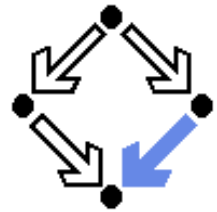
A simple example of text processing.

Example: Text Processing



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

using namespace std;

void printWords(const string& text);

int main()
{
  while (true)
  {
    string line;
    bool okay = getline(cin, line);
    if (!okay) break;
    printWords(line);
  }
}
```

Example: Text Processing



```
const char LETTERS[] =
  "abcdefghijklmnopqrstuvwxyz"
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

void printWords(const string& text)
{
  int i = 0;
  int end = text.length();
  while (i < end)
  {
    int a = text.find_first_of(LETTERS, i);
    if (a == string::npos) break;
    int b = text.find_first_not_of(LETTERS, a);
    if (b == string::npos) b = end;
    cout << text.substr(a, b-a) << "\n";
    i = b+1;
  }
}
```

Easy with the help of the existing string methods.