Structuur van Computerprogramma’s 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/
Chapter 4 - Built-in Type Constructors
Built-in Type Constructors
Overview of Concepts

- user-defined / built-in type constructors
- constant objects, constant reference parameters, constant member functions, constant data members, constant pointers
- physically / logically constant, const_cast operator, mutable qualifier
- pointers (*), null pointers (0)
- dereference operator (*), address operator (&)
- handles
- reference type
- this-pointer
- arrays, array size, array initialization
- pointer arithmetic
- c-style strings
- dangling pointers, memory leaks
- explicit memory management, memory allocation / deallocation, delete, new
A **type constructor** is a compile-time function `construct` that, given a type `t`, returns another type `construct(t)`

**Examples:**

- `&` is a type constructor for reference: `reference : T → T&`
Constants
Constant Objects

The **compiler will ensure** that (after construction) the object referred to by `Variable`, will **not be changed through this variable**

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

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

const int u(7);
int& v(u); // what will happen?
```
Constant Reference Parameters

- Indicates a promise by a function not to change a parameter
- The compiler checks if this promise is kept

```cpp
class Rational {
public:
    Rational(int num, int denom) :
        num_(num), denom_(denom) {
        if (denom == 0)
            abort();
    }

    Rational multiply(const Rational& r) {
        return Rational(num_ * r.num_, denom_ * r.denom_);
    }
private:
    int num_;  // must not be 0!
    int denom_; // must not be 0!
};
```
class Rational {
    public:
    Rational(int num, int denom) :
        num_(num), denom_(denom) { // ... 
    }
    bool isnegative() { return denom() * num() < 0; }
    Rational multiply(const Rational& r) {
        return Rational(num_ * r.num_, denom_ * r.denom_);
    }
    // ...
    private:
    // silly because operations won't work,
    // but illustrates that data members can be const
    ◆ const int num_;
    ◆ const int denom_; // must not be 0!
};

Rational r1(1, 2);
const Rational r2(2, 3);
r1.isnegative();
r1.multiply(r2);
r2.isnegative();
r2.multiply(r1);

How to ensure that a member function doesn't change the data members of the target object?

Constant members can only be initialized in the initialization list of a constructor (not in the body)
A (physically) **constant member function** promises **not to modify** the target object (i.e. everything accessible via *this*)

```cpp
class Rational {
public:
    Rational(int num, int denom) :
        num_(num), denom_(denom) { // ...
    }

    bool isnegative() const
    { return denom() * num() < 0; }

    Rational multiply(const Rational& r) const
    { return Rational(num_ * r.num_, denom_ * r.denom_); }

private:
    const int num_; // must not be 0!
    const int denom_; // must not be 0!
};
```

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

Will only work if *denom()* and *num()* used in *isnegative()* are also declared as constant member functions!
simplify() is a **logically constant** function, whereas inverse(), isnegative(), num(), denom() are **physically constant** functions
Logically Constant vs Physically Constant (2)

calling this discards the const requirement of r, since simplify is not physically constant

```cpp
ostream& operator<<(ostream& os, const Rational& r) {
    r.simplify();
    os << (r.isnegative() ? "-" : " " ) << abs(r.num());
    if (abs(r.denom()) != 1)
        os << "/" << abs(r.denom());
    return os;
}
```

Will this work?

Making simplify() a constant member function is not an option since it needs to “modify” the data members

**Options:**

- Don’t pass r as a constant reference but by value, or
- Use the const_cast operator
  - to “cast away” the constness of r
    ```cpp
    const_cast<NonConstantType>(Expression)
    ```
  - or to turn Rational::simplify into a constant member function, and use
    ```cpp
    const_cast<Rational&>(r).simplify()
    ```
  - or to turn Rational::simplify into a constant member function, and use
    ```cpp
    const_cast<Rational*>(this)->numerator_ /= g; // see later - pointers
    ```
- Use the mutable qualifier to a data member(see book p. 88)
Overloading and \texttt{const}

The usual matching rules apply (\texttt{const T} is a “normal” type)

\begin{verbatim}
int f(const int& i) {
    return i;
}

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

int main() {
    const int c(5);
    int d(5);

    // an exact match, calls f(const int&); prints 5
    std::cout << f(c) << std::endl;

    // two matches, but calls f(int&) which is the closest; prints 6
    std::cout << f(d) << std::endl;
}
\end{verbatim}
Conversion from non-const to const

This is allowed because you do not break any programmer-imposed restrictions:

- non-const implies that you are allowed (but not forced) to change the value.
- const implies that you are never allowed to change its value.

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

int main() {
    int d(5);

    // calls f(const int&);
    // after implicit ‘conversion’ of int& to const int&
    std::cout << f(d) << std::endl;
}
```
Pointers
A **pointer** is an object whose value is the address of another object.

```c
NameOfType* NameOfVariable(InitialValue);
```

- `T a(initVal);` defines a variable `a` of type `T` and initializes it with `initVal`.
- `T* p(&a);` defines a variable `p` of type `T*` (a pointer to `T`) and initializes it with the address (`& address operator`) of `a`.
- `T b(*p);` defines a variable `b` of type `T` and initializes it with the value (`* dereference operator`) stored at the location to which `p` points.

---

What’s the difference with references? (later)
Pointers Example

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

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

---

**Initializing a pointer with 0 automatically converts it to a null pointer**

**Pointer values can be accessed directly e.g. cout << p; prints the address 100012**
Handles (Pointers to Pointers)

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

**Why are handles useful?**

\[
\text{std::cout} \ll **p + *q \ll "," \ll r \ll "," \ll *q \ll "," \ll **p;
\]

**Member selection from pointers:**

\[
(*\text{Expression}).\text{MemberName} \approx \text{Expression->MemberName}
\]

\[
\text{std::cout} \ll q->\text{add}(*q); // short for (*q).add(*q)
\]
#include <iostream>

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

void
swap_r(int& x, int& y) {
    int tmp(x);
    // copy contents of what y refers to
    // to area that x refers to
    x = y;
    y = tmp;
}
What are the (dis)advantages of using pointers instead of references?

- Parameters need to be explicitly dereferenced inside the function body.
- If a parameter of reference type is passed, then it must always supply a valid (reference) to an object (null is not allowed).
- Check in the function body whether a pointer points to null!
- Pointers allow for passing array parameters (see later).
Pointers and \texttt{const}

Forbidding modification of an object “through” a pointer:

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

Forbidding modification of the pointer itself:

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

Forbidding both:

// constant pointer to constant integer
const int* const pc(&i);
Pointers versus References (Revisited)

int i(3);
int& r(i);    // reference must always be initialized
int* const p(&i); // const used to make pointer “immutable”

A reference is like a constant pointer where dereferencing is automatic

*p = 5; r = 5; // same effect
int j;    // Test “immutability” of our reference and pointer
p = &j;   // ERROR: it is also impossible to make r refer to j

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

int f(List* l); // l may be 0, i.e. not point anywhere
int f(List& l); // l ALWAYS refers to an existing List object
class T {
    ReturnType f(ParameterList) {
        T* const this(PtrToTargetObject);
        // ...
    }

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

See the earlier discussion of constant vs non-constant member functions!
// should return reference to target object
// in order to support x = y = z;

Rational&
Rational::operator=(const Rational& r) {
    num_ = r.num();
    denom_ = r.denom();
    simplify();
    return *this; // return reference to target object
}
const int SIZE = 3;
int a[SIZE]; // array of 3 int objects
    // array indices start from 0 to SIZE-1

for (unsigned int i=0;(i<SIZE);++i)
    a[i] = 2*i;

for (unsigned int i=0;(i<SIZE);++i) // will print 0 2 4
    std::cout << a[i] << " ";
Example: Bubble Sort an Array of Strings - Swap

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

void swap(std::string& x, std::string& y) {
    std::string tmp(x);
    x = y;
    y = tmp;
}
```
Example: Bubble Sort an Array of Strings - Sort

```cpp
const int MAX_WORDS = 10;
std::string words[MAX_WORDS];

int main() {
    // read 10 strings from stdin and bubble-sort them
    for (unsigned int i = 0; (i < MAX_WORDS); ++i)
        std::cin >> words[i];

    for (unsigned int size = MAX_WORDS - 1; (size > 0); --size)
        // find largest element in 0..size range and
        // store it at index size
        for (unsigned int i = 0; (i < size); ++i)
            if (words[i + 1] < words[i])
                swap(words[i + 1], words[i]);

    // print the sorted strings
    for (unsigned int i = 0; (i < MAX_WORDS); ++i)
        std::cout << words[i] << " ";
}
```
#include <iostream>

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

int main() {
    // how to find the number of elements in vat_rates?
    unsigned int size(sizeof(vat_rates) / sizeof(float));

    const char message[] = "VAT rates"; // special case
    std::cout << message;

    for (unsigned int i = 0; (i < size); ++i)
        std::cout << " " << vat_rates[i];

    std::cout << std::endl;
}
Arrays of class objects are initialized using the default constructor (without arguments)

class Rational {
public:
   Rational(int num = 0, int denom = 1) : 
      num_(num), denom_(denom) { }
   // ...
private:
   int num_; 
   int denom_; 
};

// calls Rational::Rational() on each element
Rational rations[3];

// constructors can be used in the initialization
Rational more_rationals[] = { Rational(1, 2), Rational(1, 3) };
int sum(int a[], unsigned int size) {
    int total(0);
    for (unsigned int i = 0; i < size; ++i)
        total += a[i];
    return total;
}

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

• **Arrays are passed by reference**
• The compiler doesn’t care about the size of the array
  • The programmer should check this! (why?)
Arrays versus Pointers

A pointer can be made to point to an array and it can also be indexed

```cpp
#include <iostream>

void f(int x[]) {
    x[0] = 1;
}

int main() {
    int a[] = { 0, 2, 3 };
    int* p(a);
    int* q(&a[0]); // exactly the same as p (i.e. the address of a[0])

    // passing a pointer or an array to f() is the same
    f(p);
    // printing: 1, 1 \newline
    std::cout << *p << ", " << a[0] << std::endl;
    // printing: 1, 1 \newline 2, 2 \newline 3, 3 \newline
    for (unsigned int i = 0; (i < sizeof(a) / sizeof(int)); ++i)
        std::cout << p[i] << ", " << a[i] << std::endl;
}
```
# Pointer Arithmetic

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

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

int main() {
    int* p(a);
    int* q(p + 2);
    int* s(q - 2);

    for (unsigned int i = 0; (i < 3); ++i)
        std::cout << *p++ << "\n";

    std::cout << q - p << std::endl;
}
```

What is the output?

```
3
2
1
```
• String literals can be represented by using `const char*`

• These are **C-style** strings, in C++ one can use the `String` container

```cpp
const char* hi("hello world");

std::cout << "hello world";'  // hello world
std::cout << "hello" "world";'  // helloworld

std::cout << "hello world\n";'  // hello world (with newline)
std::cout << "hello world" << std::endl;' // hello world (with newline)
std::cout << hi;'  // hello world
```

“hello world” returns a pointer (*) to an array of constant characters, which is automatically closed off with a null character `\0`
C-Strings: Example

#include <iostream>

void print(std::ostream& os, const char*p) {
    while (*p) {
        os << *p++;
        os << std::endl;
    }
}

int main() {
    const char* s("hello world");
    print(std::cout, s);
}
Comparing C-Strings

```cpp
#include <iostream>

// returns
// 0 if s1 and s2 are lexicographically equal
// >0 if s1 is lexicographically larger than s2
// <0 if s1 is lexicographically smaller than s2
int strcmp(const char*s1, const char* s2) {
    while (((*s1) && (*s2) && (*s1 == *s2)) {   
        ++s1;
        ++s2;
    } // while loop stops if a \0 character is spotted
    return *s1 - *s2; // difference between ascii of letters
}

int main() {
    const char* s1("abc");
    const char* s2("abcde");
    std::cout << strcmp(s1, s2) << std::endl; // prints -100
```
#include <iostream>
#include <stdlib.h> // for atoi()

// this program computes the sum of its command line args
// usage: sum int...

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

    // atoi(const char*) converts a string to an int
    for (unsigned int i = 1; (i < argc); ++i)
        sum += atoi(argv[i]);
    std::cout << sum << std::endl;
}
Explicit Memory Management
Explicit Memory Allocation

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

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

Explicitly allocated memory does not go away unless the programmer explicitly deallocates it!
Explicit Memory Deallocation

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

delete p; // deallocate memory allocated with new

delete [] q; // deallocate memory allocated with new[]
```

After the delete, the pointers are left “dangling”
The Dangling Pointer Syndrome

```c
int* q = new int[3];
int* p = q;

// q is a dangling pointer!
```

delete[] q;
q = 0; // p is left dangling!
Memory Leaks

```
int* q = new int;
...  // accessing elements q[0], q[1], q[2]
```

The original allocated memory cannot be referenced anymore!

```
int* q(new int[3]);
```

'lost' memory
<table>
<thead>
<tr>
<th></th>
<th>how defined</th>
<th>when initialized</th>
<th>when destroyed</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>static var in function body</td>
<td>first call of function</td>
<td>program exit</td>
</tr>
<tr>
<td></td>
<td>static class member</td>
<td>program startup</td>
<td>program exit</td>
</tr>
<tr>
<td></td>
<td>global variable</td>
<td>program startup</td>
<td>program exit</td>
</tr>
<tr>
<td>automatic</td>
<td>local var in function body</td>
<td>when definition is executed</td>
<td>exit scope</td>
</tr>
<tr>
<td>member</td>
<td>data member of class</td>
<td>just before owner</td>
<td>just after owner</td>
</tr>
<tr>
<td>free</td>
<td>using new/delete</td>
<td>determined by programmer</td>
<td>determined by programmer</td>
</tr>
</tbody>
</table>