INFORMATIK
für Mathematiker und Physiker

Eine Einführung in C++

Kurzzusammenfassung zur Vorlesung 252-0847-00
Herbstsemester 2013, ETH Zürich

Bernd Gärtner

Lecture 1: The first program and the C++ syntax and semantics

We saw an overview about what it means to program, how to program, and why it is important to be able to program. The main point I made is that basic programming skills are becoming increasingly important in a world "run" by software (programs).

Then we discussed our first C++ program and the main syntactical and semantical terms necessary to understand it. The following figure summarizes this quite nicely.

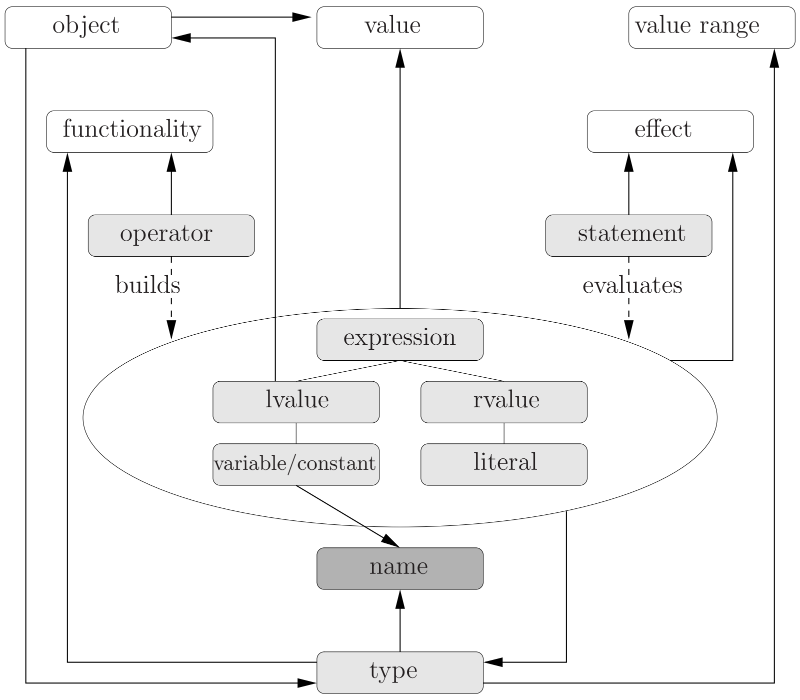


Figure 1: *Syntactical and semantical terms appearing in our first program* `power8.cpp`. *Purely semantical terms appear in white, purely syntactical terms in dark gray. Mixed terms are drawn in light gray. Solid arrows* $A \rightarrow B$ *are to be read as "A has B", while solid lines in the expression part mean that the upper term is more general than the lower one.*

The two most important syntactical concepts are *expressions* and *statements*. An expression represents a computation (for example `b * b`), and expressions can be combined through operators to new expressions (such as in `b = b * b`). Every expression has a type, a value, possibly an effect, and (if it is an lvalue) an address. Value and effect materialize only in an evaluation of the expression. A statement is the basic building block of every program. It has an effect, and often this effect results from evaluating an expression.

In the lecture, we also introduced the type `ifm::integer` for computing with integers of arbitrary length. Such a type is necessary, since the fundamental type `int` has a rather small finite value range and thus frequently "overflows".

Lecture 2: The types `int`, `unsigned int`, arithmetic operators

Here we introduced the arithmetic operators (on the types `int` and `unsigned int`), along with their precedences (*binding power*), associativities (evaluation from left to right, or right to left?), and arities (number of operands).

Description	Operator	Arity	Prec.	Assoc.
post-increment	++	1	17	left
post-decrement	--	1	17	left
pre-increment	++	1	16	right
pre-decrement	--	1	16	right
sign	+	1	16	right
sign	-	1	16	right
multiplication	*	2	14	left
division (integer)	/	2	14	left
modulus	%	2	14	left
addition	+	2	13	left
subtraction	-	2	13	left
assignment	=	2	3	right
mult assignment	*=	2	3	right
div assignment	/=	2	3	right
mod assignment	%=	2	3	right
add assignment	+=	2	3	right
sub assignment	-=	2	3	right

Table 1: *Arithmetic operators*

Most importantly, the division / is the integer division, meaning that 5/2 has value 2, for example, where the remainder 1 can be obtained through the modulus operator (5%2). The three main rules one needs to remember are:

Arithmetic Evaluation Rule 1: Multiplicative operators have higher precedence than additive operators.

Arithmetic Evaluation Rule 2: Binary arithmetic operators are left associative.

Arithmetic Evaluation Rule 3: Unary operators + and - have higher precedence than their binary counterparts.

We have also talked about the value ranges of the types `int` and `unsigned int`. Under a b-bit representation (think of $b = 32$), `int` has value range $\{-2^{b-1}, -2^{b-1}+1, \ldots, -1, 0, 1, \ldots, 2^{b-1}-1\} \subset \mathbb{Z}$, while `unsigned int` has value range $\{0, 1, \ldots, 2^b - 1\}$.

Lecture 3: The type `bool` and control statements

We have introduced the type `bool` to work with truth values, along with the logical operators (that combine boolean expressions), and the relational operators (that compare arithmetic expressions and return boolean expressions). A specialty of boolean expressions built from `&&` or `||` is the short circuit evaluation: if the value is already known after evaluation of the left operand, the right operand is not evaluated. This allows for expressions such as `x == 0 || y / x > 2` where the second operand is valid only if $x \neq 0$.

Description	Operator	Arity	Prec.	Assoc.
logical not	!	1	16	right
less	<	2	11	left
greater	>	2	11	left
less or equal	<=	2	11	left
greater or equal	>=	2	11	left
equality	==	2	10	left
inequality	!=	2	10	left
logical and	&&	2	6	left
logical or	||	2	5	left

Table 2: *Logical and relational operators*

The major part of this lecture was about control statements that allow us to deviate from linear control flow (statements are executed from top to bottom in the program). The control statements are

- the `if`- and the `if-else` statement (*selection* statements);
- the `for`-statement (*iteration* statement); this is at the same time the most used, the most important, and most complex control statement;
- the `while` and the `do`-statements (two more *iteration* statements); since they can both easily be simulated using the `for`-statement, they are redundant from a functional point of view, so we called them *syntactic sugar*. However, they still make sense and allow for more readable code;
- `break;` and `continue;` (*jump* statements). The typical use of `break;` is to terminate a loop in the middle, and the use of `continue;` is to skip the remainder of a loop iteration.

In this context, we discussed undecidability of the *halting problem*: there is no C++ program that can check any C++ programs for termination.

We also discussed *blocks* as a means to group several statements into one, and we said that each block is a *scope* in the sense that variables defined in the block start to live (are accessible and get memory assigned) when the program control enters the block and die (get their memory freed and become inaccessible) when the block is left. This is the concept of *automatic storage duration*.

Lecture 4: Floating point numbers

We have discussed how to compute with "real" numbers in C++ using floating point numbers and the types `float` and `double`. Floating point numbers are numbers of the form $\pm s \cdot 2^e$, where both s (the significand) and e (the exponent) are integers. Concrete floating point number systems (such as *IEEE single precision* and *IEEE double precision*) prescribe bounds for s and e.

We have seen that floating point numbers are very useful for a wide range of applications (we have computed the Euler number and Harmonic numbers in the lecture), but that they come with some pitfalls that one needs to know. Most prominently, we discussed that floating point inputs and literals are usually decimal (such as `0.1`, meaning 1/10), but that internally, a binary representation is used. In converting between these two representations, errors are unavoidable, since numbers with finite decimal representation may have infinite binary representation. This can lead to small errors (0.1 internally not being 0.1 but very close to it), and large errors (Excel 2007 bug).

We established three guidelines for safely computing with floating point numbers.

Floating Point Arithmetic Guideline 1: Never compare two floating point numbers for equality, if at least one of them results from inexact floating point computations.

Floating Point Arithmetic Guideline 2: Avoid adding two numbers that considerably differ in size.

Floating Point Arithmetic Guideline 3: Avoid subtracting two numbers of almost equal size, if these numbers are results of other floating point computations.

What happens when we violate the second one was demonstrated by a program for computing Harmonic numbers in two ways (forward and backward); only the backward way is approximately correct, while in the forward way, the numbers $1/i$ added last are too small to still contribute.

```
// Forward sum, yields large errors for large n
float fs = 0;
for (unsigned int i = 1; i <= n; ++i)
  fs += 1.0f / i;

// Backward sum, approximately correct also for large n
float bs = 0;
for (unsigned int i = n; i >= 1; --i)
  bs += 1.0f / i;
```

Lecture 5: Functions

We have discussed functions as a way to realize functionality that is frequently used (for example, power computations) in such a way that it is implemented only once (as a function) but can be used very often, and even from different programs. We have seen that the standard library contains many of such functions (e.g. `std::sqrt`).

We have made the point that functions should be properly documented through pre- and postconditions, and that assertions should be used to check pre- but also other conditions at any time during a program.

We have also seen that function calls are expressions, and how the evaluation of a function call proceeds (evaluation of call arguments, initialization of formal arguments, execution of function body, return of function value (optional for `void` functions)).

An important concept is that of *procedural programming* where a program is structured by subdividing its task into small subtasks, each of which is realized by a function. Correctness of the program then easily follows from correctness of all functions, and the program becomes easier to read if meaningful function names are used. Here is an example (computation of perfect numbers between 1 and $50,000$)

```
// POST: return value is the sum of all divisors of i
//       that are smaller than i
unsigned int sum_of_proper_divisors (const unsigned int i)
{
  unsigned int sum = 0;
  for (unsigned int d = 1; d < i; ++d)
    if (i % d == 0) sum += d;
  return sum;
}

// POST: return value is true if and only if i is a
//       perfect number
bool is_perfect (const unsigned int i)
{
  return sum_of_proper_divisors (i) == i;
}

int main()
{
  std::cout << "The following numbers are perfect.\n";
  for (unsigned int i = 1; i <= 50000 ; ++i)
    if (is_perfect (i)) std::cout << i << " ";
  std::cout << "\n";

  return 0;
}
```

Lecture 6: Arrays and vectors

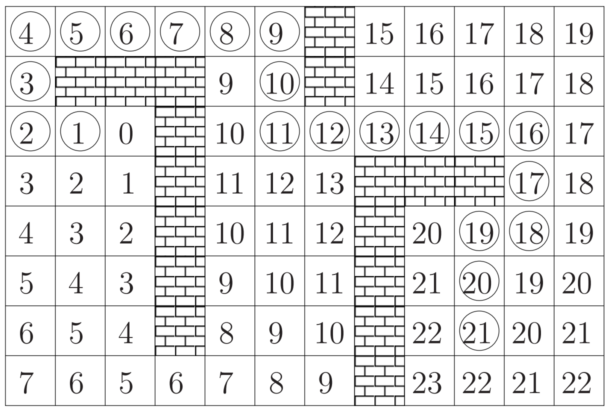
We have seen the concept of an array as a container that stores a sequence of data of the same type, and allows fast random access (by index) to the elements. This is possible since an array occupies a consecutive piece of memory, so the address of any element is easy to compute as $p + si$, where p is the address of the first element, i is the index of the element, and s is the number of memory cells required to store a single element.

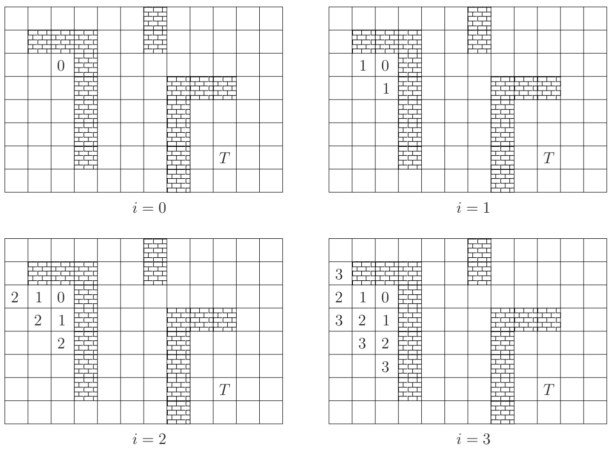
We have seen two models of the array concept: static arrays (the number of elements has to be known at compile time) and vectors (the number of elements can be determined at runtime). Static arrays turn out to be primitive (they originate from C), in the sense that initialization and assignment don't work as for other types, and we have to "copy arrays by hand".

As applications for arrays, we have considered (i) Eratosthenes' sieve for computing all prime number up to a given upper bound, and (ii) string matching: find a given pattern in a given text.

For the latter, we also needed the types `char` and `std::string` to represent single characters and texts, and we have presented another simple text processing task (en- and decryption of texts using the Caesar cipher).

Then we have discussed multidimensional arrays (arrays of arrays of arrays...), with one (twodimensional) application: finding shortest paths for a robot on a factory floor from a start to a goal position that avoids all obstacles.


$i = 23$



The main idea in the solution was to label every cell with the length of the shortest path from the start to the cell (a twodimensional array stores these labels), and use an expanding wavefront to assign labels in increasing order. In the left picture, the cell labeled 0 is the start, and the cell labeled 21 is the goal. The indicated path of increasing labels is the shortest path. A single step of expanding the wavefront to label i is really simple (right picture): all unlabeled neighbors of cells with label $i - 1$ receive label i.

Lecture 7: Pointers, Algorithms, Iterators, Containers

We have discussed how to pass arrays to functions: via two *pointers*, one pointing to the first element, and one *behind* the last element. A pointer of type T* stores the address of an object of type T. A static array is automatically converted to a pointer to its first element in every expression, and using *pointer arithmetic*, we can get pointers to other elements. For example, to set all 5 elements of an array a to 1, we can use

```
// a points to first element, a+5 behind the last
std::fill (a, a+5, 1);
```

Here, std::fill is an *algorithm* from the standard library. In realizing such a function for an array of length n, we have discussed iteration by random access, and natural iteration:

```
// iteration by random access: read book by
// opening it on page 1, reading two pages, closing book,
// opening it on page 3, reading two pages, closing book, ...
for (int i=0; i<n; ++i)
   a[i] = 1;

// natural iteration: read book by
// opening it on page 1, reading two pages, turning page,
//                       reading two pages, turning page,...
for (int* p = a; p < a+n; ++p)
   *p = 1;
```

For vectors, we cannot use pointers, but have to work with *vector iterators* that can be thought of as "pointers" to vector elements. To set all elements of a vector, we use

```
// v.begin() points to first element, v.end() behind the last
std::fill (v.begin(), v.end(), 1);
```

An iterator is a type whose expressions behaves like pointers, meaning that an expression it of iterator type can at least be dereferenced to yield the element it points to (*it), and advanced to point to the next element (++it). Some iterators (for example, pointers and vector iterators) also support random access (it[k]).

A *container* is an object that can store other objects (its elements), and that offers some ways of accessing these elements. Every standard container allows access through iterators. If c is a standard container of type C, the following loop will output all its elements.

```
for (C::const_iterator it = c.begin(); it != c.end(); ++it)
   std::cout << *it;
```

Concretely, we have seen this loop for C equal to std::set<char>, a standard container to represent sets of characters. Set iterators do not allow random access, since from a mathematical point of view, a set is not ordered: there is no natural way to define the i-th element.

Lecture 8: Recursion

We have discussed *recursive* functions, functions that call themselves within their body. Here is an example for computing the Fibonacci numbers

$$\begin{aligned} F_0 &:= 0, \\ F_1 &:= 1, \\ F_n &:= F_{n-1} + F_{n-2}, \quad n > 1. \end{aligned}$$

This recursive mathematical definition has a direct counterpart in C++:

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib (const unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

For a recursive function, we separately have to prove *correctness* and *termination.*

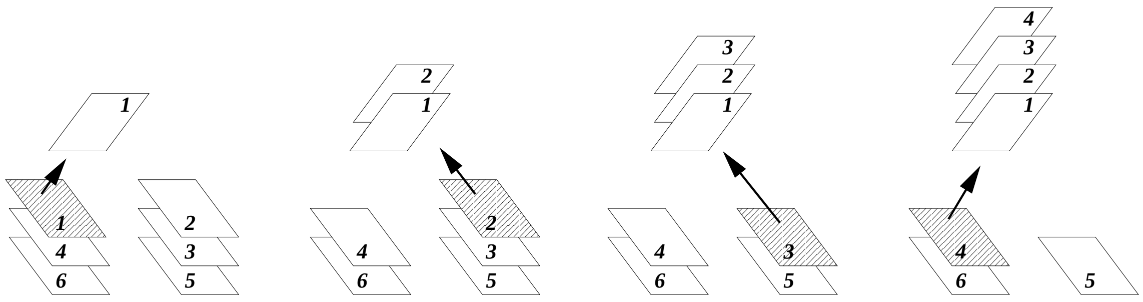
The evaluation of a recursive function call may take very long (Examples: `fib(50)`, or—taken to the extreme—Ackermann function values). In many cases (in particular under *tail-end recursion*, but also for Fibonacci numbers), there are efficient iterative (non-recursive) ways of writing the same function. Here is a fast iterative function for Fibonacci numbers:

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (const unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    const unsigned int a_prev = a;  // F_{i-2}
    a = b;                          // F_{i-1}
    b += a_prev;                    // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

We have discussed the *call stack* as a way to deal with several formal arguments that are in use simultaneously during the evaluation of nested function calls.

Lecture 9: Mergesort and Lindenmayer systems

We have seen two applications of recursive functions: efficient sorting, and fractals. *Mergesort* follows the paradigm *Divide-and-Conquer*: divide the sequence to be sorted into two equal parts, and recursively sort the parts separately. Then merge the two sorted parts into one sorted part, as visualized in the following picture:



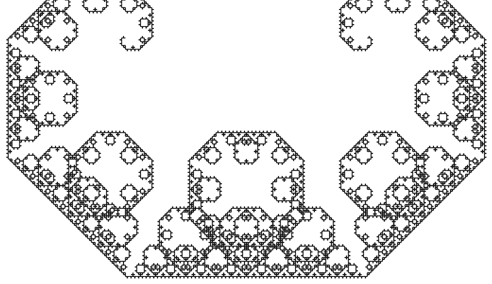
We have counted the total number $T(n)$ of comparisons between elements when we sort n numbers, and we have argued that the total runtime will be proportional to this number. The comparisons happen in the merge steps when we need to decide which of the two current top cards is the smaller one. We have $T(0) = T(1) = 0$ and for $n \geq 2$, $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1$. We proved by induction that this yields $T(n) \leq (n-1)\lceil \log_2 n \rceil$, and we mentioned that no comparison-based sorting algorithm can be much faster.

A Lindenmayer system yields a sequence of words, where each word results from the previous one by replacing each symbol according to a production rule. For example, with initial word F and replacement of F with F + F+ (other symbols are replaced by themselves), we get

$$w_0 = \text{F}, w_1 = \text{F+F+}, w_2 = \text{F+F++F+F++}, w_3 = \text{F+F++F+F+++F+F++F+F+++}, \ldots$$

A word described by a Lindenmayer system can be interpretated as a sequence of *turtle graphic* commands, where F means *forward* (one step, and leave a trace), and $+/-$ mean *turn (counter)clockwise* (by 90 degrees). As words can be generated recursively, we can write simple programs for getting the turtle graphic drawing. In our example, this looks as follows (drawing for $i = 14$).

```
// POST: the word w_i^F is drawn
void f (const unsigned int i) {
  if (i == 0)
    ifm::forward();    // F
  else {
    f(i-1);            // w_{i-1}^F
    ifm::left(90);     // +
    f(i-1);            // w_{i-1}^F
    ifm::left(90);     // +
  }
}
```



Lecture 10: Structs and Reference Types

We have seen *structs* as a way to define new types whose value range is the Cartesian product of existing types. Our running example was that of rational numbers:

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

We have seen how to provide functionality for a new type (such as `rational`), in order to make it behave like existing types. The concept here is *operator overloading*. For example, to enable us to write `r + s` if `r` and `s` are of type rational, we overload the binary addition operator:

```
// POST: return value is the sum of a and b
rational operator+ (const rational a, const rational b)
{
  rational result;
  result.n = a.n * b.d + a.d * b.n;
  result.d = a.d * b.d;
  return result;
}
```

We can do the same with all other arithmetic operators, relational operators, input and output operators. To realize the arithmetic assignment operators (`r += s`), we needed *reference types* in order to allow functions to change the values of their call arguments. With reference types, operator+= looks as follows.

```
// POST: b has been added to a; return value is the new value of a
rational& operator+= (rational& a, const rational b)
{
  a.n = a.n * b.d + a.d * b.n;
  a.d *= b.d;
  return a;
}
```

If a formal argument is of reference type, we have *call by reference*, otherwise *call by value*. For large types, call by value can be inefficient, since the whole value is copied. We can avoid this by using *const references* instead and are still able to pass rvalues. For operator+, this would look as follows.

```
rational operator+ (const rational& a, const rational& b)
{
  rational result;
  result.n = a.n * b.d + a.d * b.n;
  result.d = a.d * b.d;
  return result;
}
```

Lecture 11: Classes

We have addressed the main drawback of structs so far, namely that the internal representation (`.n` and `.d` for rational numbers) is revealed to the user ("customer") of the struct, meaning that it is hard to change it later. With classes, we hide the internal representation by making it *private*, and allow access to it only through *public member functions*. This means, we promise a certain functionality, but not a concrete realization of it. In case of rational numbers, this looks as follows.

```
class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const
   {
     return n;
   }
   // POST: return value is the denominator of *this
   int denominator () const
   {
     return d;
   }
private:
   int n;
   int d; // INV: d!= 0
};
```

In order to allow objects of class types to be initialized, we have discussed the concept of *constructors* as member functions, for example

```
// PRE: den != 0
// POST: *this is initialized with num / den
rational (const int num, const int den)
  : n (num), d (den)
{
  assert (d != 0);
}
```

To use this constructor, we write

```
rational r (1,2); // initializes r with value 1/2
```

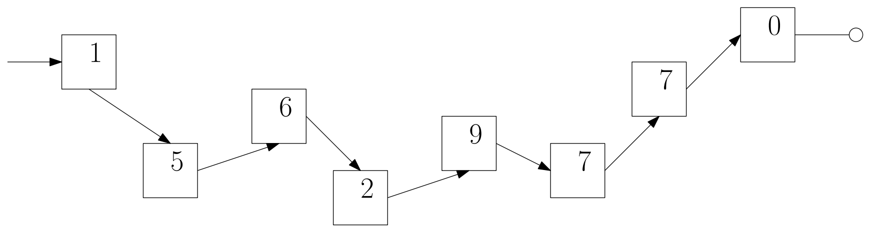
As an application, we have developed a class for computing a sequence of pseudorandom numbers according to the *linear congruential method*: Given a *multiplier* $a \in \mathbb{N}$, an *offset* $c \in \mathbb{N}$, a *modulus* $m \in \mathbb{N}$ and a *seed* $x_0 \in \mathbb{N}$, the sequence $x_1, x_2, \ldots$ of natural numbers defined by the rule

$$x_i = (ax_{i-1} + c) \bmod m, \quad i > 0$$

is a sequence of pseudorandom numbers.

Lecture 12: Lists and dynamic data types

We have seen that for arrays, it is difficult to insert or remove elements "in the middle", and we have proposed *lists* as a solution that may store their elements anywhere in memory (and not consecutively as arrays do it). In addition to the *key*, each list element also stores a pointer to its successor:



The main new concept we needed in order to insert new elements was *dynamic memory allocation* from the heap using `new` (and a later `delete` to free the memory again). Here is the `push_front` member function of the class list that dynamically allocates a new list node, initializes it using an appropriate constructor of the class `Node`, and makes it the new first element of the list:

```
void list::push_front (int key)
{
  head_ = new node (key, head_);
}
```

We have also seen how one traverses lists, using a "running pointer" `p` that is advanced in each step according to `p = p->get_next();`

In order to override default initialization and assignment of lists that would just copy the pointer to the first element (making the new list an alias of the old one), we needed to provide user-defined *copy constructor* and *assignment operator*. To properly clean up the list (`delete`!) when it runs out of scope, we also need to provide a user-defined *destructor*, calling a clear method:

```
void List::clear ()
{
  Node* p = head_;
  while (p != 0) {
    p = p->get_next();
    delete head_;
    head_ = p;
  }
  head_ = 0;  // list is now empty
}
```

Having constructors, destructor, copy constructor and assignment operator, our type is a *dynamic data type*. The customer can work with list objects of automatic storage duration, and all the dynamic memory management happens in the background.