C++

Alexander Thomas

11 maart 2004

Inhoudsopgave

1 Algemeen (verschillen/uitbreidingen t.o.v. C)

- **Functie-prototypes zijn *verplicht!***
- **Default functie-argumenten**
  Bv. `Void GenerateTone( short frequency = 440 );`
  Liefst in prototype declareren; bij meerdere argumenten moeten alle argumenten na het eerste met default-waarde ook een default-waarde krijgen.
- **Referentie-variabelen**
  Zoals in C worden parameters standaard 'by value' doorgegeven. Referentie-variabelen laten toe om een parameter 'by reference' door te geven zonder pointers te gebruiken. Dit werkt alleen als het type van het argument overeenkomt met dat van de declaratie.
  Bv.: `void DoubleMyValue( short &referenceParam ) { ... }.`
  Je kan een referentie-variabele op elk moment declareren als "*`type &refName = varName;`*", op voorwaarde dat de variabele ook meteen wordt geïnitialiseerd.
- **Function Name Overloading**
  Je kan (zoals in Java) verschillende functies met dezelfde naam maken, als de 'signatuur' (= lijst van parameters) verschilt. Welke functie dan opgeroepen wordt, hangt af van welke parameters je meegeeft (de return-waarde hoort niet bij de signatuur!)
- **Geheugenallocatie**
  `new` & `delete`: zelfde functionaliteit als `malloc` en `free` in C. Het is echter niet aan te raden om deze twee mechanismen te mengen, ttz. gebruik nooit `delete` voor een array die aangemaakt is met `malloc`, of `new` voor een array die in een stuk C-code gebruikt zal worden.
  Bv.: `buffer = new char[1024]; ... delete [] buffer;`

  `new` geeft een pointer naar de gealloceerde geheugenruimte. Als je een array alloceert, moet "`[]`" gebruikt worden bij `delete`, zoniet wordt enkel het eerste element gedealloceerd. Indien de allocatie faalt, is de teruggegeven pointer `NULL` (tip: alloceer in het begin v.h. programma een stuk geheugen dat volstaat om in geval van tekort aan opruiming te doen.) Je kan met `set_new_handler( NewFailedFunction )` een functie *`NewFailedFunction`* laten oproepen telkens new faalt (vereist `#include <new>`).
- **Scope operator**
  Als je '::' voor een variabele zet, kijkt de compiler naar een variabele met die naam buiten het huidige blok.
- **Inline functions**
  Als je een functie als '`inline`' declareert, wordt bij compilatie (indien ondersteund) de inhoud van die functie in plaats van de oproep gekopieerd. Dit vermijdt overhead van subroutines, maar kost meer geheugen (dus enkel gebruiken als snelheid echt kritisch is). De implementatie moet direct bij de declaratie gevoegd worden.

2 Object-georiënteerd Programmeren

2.1 Declaratie van een klasse

```
class ClassName {
// Data members:
   type1 var1;
   ...
// Member functions:
   functionPrototype1;
   ...
};
```

2.2 Een object van een klasse creëren

- Met definitie *(= automatic object)*:
  `ClassName myObject;`
  Het object bestaat dan enkel binnen de scope waarin het gedefinieerd werd.
- Met de new-operator:
  `ClassName *objectPointer;`
  `objectPointer = new ClassName;`
  Het object blijft bestaan totdat het terug vernietigd wordt met `delete`.
  Bv. `delete objectPointer;`

2.3 Verwijzen naar data-velden en member functions van een object

Zoals bij structs (dus met `.` of met `->` voor pointers).

Bv.
```
Employee *employeePtr;
EmployeePtr = new Employee;
EmployeePtr->PrintEmployee();
```

Binnen een member function verwijzen oproepen naar data-velden of member functions naar het huidige object. Om verwarring te vermijden kan er expliciet "`this`" gebruikt worden.

2.4 Functies voor klassen

Syntax: `type ClassName::FunctionName( arguments ) { ... }`
Deze hebben automatisch toegang tot alle data members.

Functies die toegang tot de data members geven, zijn *access functions*. Degene die een waarde opvragen, zijn *getter functions* (*inspectoren* in Java), degene die een waarde instellen, zijn *setter functions* (*mutatoren* in Java).

2.5 Constructor

- Is optioneel
- Heeft dezelfde naam als de klasse en wordt automatisch opgeroepen bij creatie van een object.

  Bv.
  ```
  Employee::Employee( float salary ) {
       employeeSalary = salary;
  }
  ```

- Heeft nooit een return-waarde! Dus ook in de declaratie geen vermelding v.e. return-waarde. Als er een meer ingewikkelde initialisatie nodig is waarbij wel een waarde wordt teruggegeven, is het dus aangeraden een aparte initialisatiefunctie te maken, typisch I+ClassName (= 'two-stage construction'). Dit is dan ook de plaats waar geheugenallocatie of andere initialisatie die kan falen, moet gebeuren.

2.6 Destructor

- Is optioneel
- Notatie: `~ClassName()`, wordt opgeroepen bij destructie van een object. De taak van de destructor is meestal het vrijmaken van geheugen dat gealloceerd werd in de constructor.
- Heeft geen return value noch parameters!
- Als je een array van objecten die een destructor hebben wilt vernietigen, moet je "`delete [] arrayPtr;`" gebruiken, anders wordt enkel de destructor van het eerste element opgeroepen.

2.7 Access Privileges

Elk data member of functie kan een access code krijgen, indien niet gespecifieerd wordt er `private` verondersteld.

- `private:` de data member of functie is enkel toegankelijk voor member functions van dezelfde klasse.
- `public:` de functie of data member is overal toegankelijk.
- `protected:` zelfde als private, maar ook toegankelijk in afgeleide klassen.

Gebruik: zet '`public:`' vóór de functies die publiek moeten zijn, enz. Alle functies tot aan de volgende access code zijn dan publiek. Meestal worden de data members private gehouden.

Een memberfunctie kan ook als constant gedeclareerd worden, bv. `void display() const;` Dit betekent dat de functie geen data van de klasse verandert. Een poging tot wijzigen van de klasse in deze functie, zal een compilatiefout geven.

2.8 Voorbeeld van een volledige klasse-declaratie

```
class Employee
{
// Data members:
  char employeeName[ kMaxNameSize ];
  long employeeID;
  float employeeSalary;

// Member functions:
  public:
       Employee( char *name, long id, float salary );
       ~Employee( void );
  void PrintEmployee( void );
};
```

2.9 Friends

Door een klasse of functie als '`friend`' te declareren van een andere klasse, krijgt deze klasse/functie toegang tot alle data members & functies van deze klasse, ook de '`private`'. Deze declaratie gebeurt in de klasse die toegang geeft, niet in de klasse die ze krijgt:

```
class Employee; // zie verder(*)
class Supervisor {
  ...
  void PayRaise( Employee *luckyPerson);
};
class Employee {
  friend class Supervisor;
  ...
};
```

of:

```
class Employee {
  friend void Supervisor::PayRaise( Employee *luckyPerson );
  ...
};
```

(*): Als twee klassen naar elkaar verwijzen, moeten de klassen waarnaar verwezen wordt, op voorhand gedeclareerd worden.
Ook niet-member functions kunnen als friend gedeclareerd worden, bv.: `friend int main(void)`

2.10 Overerving / Afgeleide klassen

Syntax: `class DerivedClass : accesscode BaseClass { ... };`

Een afgeleide klasse erft alle non-private data members & member functies van zijn basisklasse. *accesscode* kan zijn: `public, protected, private` of kan weggelaten worden (dan wordt `private` verondersteld) en bepaalt wat de privileges van de overgeërfde members zijn.

Bij **constructie** v.e. object van een afgeleide klasse, wordt eerst de constructor van de basisklasse opgeroepen, daarna die van de afgeleide klasse. Voor de destructie is het juist omgekeerd. Als de constructor van de basisklasse **parameters** heeft, moet in de constructor van de afgeleide klasse vermeld worden welke waarde deze parameters krijgen, bv.:

```
Derived::Derived( void ) : Base( 20 ) { }
```

of:

```
Derived::Derived( short derivedParam ) : Base( derivedParam ) { }
```

(Wat na de ':' komt, heet algemeen de *Member initialisation list.*)

2.10.1 Virtuele functies en abstracte klassen

Een functie uit de basisklasse kan vervangen worden door een eigen versie in de afgeleide klasse, als deze functie in de basisklasse als `virtual` is gedeclareerd (mag enkel in de *declaratie* van de klasse!)

Je kan objecten van afgeleide klassen toekennen aan variabelen of pointers van de basisklasse (niet omgekeerd). Als het niet de bedoeling is om specifieke objecten van de basisklasse zelf aan te maken, is dit een 'abstracte klasse'. Een functie kan ook als 'puur virtueel' gedeclareerd worden (zonder implementatie) in de basisklasse, bv: `virtual void showArea( void ) = 0;`

2.10.2 Ongebruikte parameters

Als je in een methode parameters voorziet die niet gebruikt worden in de superklasse, maar wel in de subklasse, zal je een waarschuwing 'unused parameter' krijgen. Om dit te vermijden, laat in de definitie de naam van de parameter weg als deze niet gebruikt wordt, bv.:
niet: `int Foo::bar(int arg) { return 0; }`
maar: `int Foo::bar(int) { return 0; }`

3 Operator overloading

Een functie van de vorm *`type`* `operator`*`C++ operator`*`(` *`arguments`* `) { ... }` vervangt de operator *`C++ operator`* door de functie. Het aantal parameters moet overeenkomen met het aantal operanden van de operator. Je kan eenzelfde operator meerdere keren overloaden, op voorwaarde dat de argumenten verschillen.

Als je een member-functie gebruikt, wordt de member-functie van de eerste operand opgeroepen, met als parameter de tweede operand, bv.:

```
float MenuItem::operator+( MenuItem item ) {
  return ( this->GetPrice() + item.GetPrice() );
}
```

De operatoren die overloaded kunnen worden, zijn:

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

Beperkingen:

- De operatoren `.` `.*` `::` `?:` en `sizeof()` kunnen niet overloaded worden.
- Het type van de argumenten van de overloaded operator mag geen voorgedefinieerd type (zoals `int`) zijn.
- Je kan voor `++` en `-` niet nagaan of ze in prefix of postfix gebruikt zijn.
- Je kan de prioriteit van een operator niet veranderen.
- Je kan geen default parameters gebruiken.
- `new, delete, (), [], ->` en `=` kunnen enkel door member-functies overloaded worden.

  Voor `new`: `void *operator new( size_t blobSize );`
  `blobSize` is de grootte v.h. te creëren object. Return type moet `void*` zijn. Voor `size_t` is `#include <stddef.h>` nodig.

  Voor `delete`: `void operator delete( void *blobPtr, size_t blobSize );`
  `blobSize` is optioneel, return type moet `void` zijn.

  Voor `()` *(call operator)*: bv.: `float operator()( float taxRate );`
  Kan gebruikt worden om een verkorte toegang tot een belangrijke data member te geven.

  Voor `[]` *(subscript operator)*: kan gebruikt worden om out-of-bounds indices te detecteren.
  Bv.
```
char& Name::operator[]( short index ) {
   if bounds-checking ...
   else
     return ( nameString[index] );
}
```

Voor `->` *(arrow of member selection operator)*: deze operator kan enkel een object als linkerargument hebben als de klasse van dat object deze operator overloadt. Deze overload-functie wordt dan uitgevoerd en de return-waarde wordt in de plaats van het linkerargument gezet, en de expressie wordt opnieuw uitgevoerd (dit kan zich meerdere keren herhalen, totdat de return-waarde een pointer is.) De `->` operator wordt in dit geval een *'smart pointer'* genoemd.

Voor `=` : nuttig om bv. in een uitdrukking als "`destination = source`" waarbij `source` en `destination` objecten van een klasse `String` zijn, te zorgen dat de inhoud van de strings gekopieerd wordt en niet enkel het adres zoals bij de normale memberwise assignment. Bv. `String &String::operator=( const String& fromString ) { ... }`

Let op: bij uitdrukkingen als `int x = time1 + time2 + time3`, waarbij de som van twee `time` types een `int` teruggeeft, zijn twee verschillende versies van de overloaded operator nodig (want `time1` moet rechts opgeteld worden met een `int`!) Voor dezelfde operanden en een voorgedefinieerd type als return-waarde heb je dus 3 versies nodig.

4 Templates

Templates laten toe om data types in een klasse of functie te parametriseren, zodat je dezelfde definitie met verschillende types kan gebruiken zonder de code te moeten kopiëren. Vb.:

```
template <class A, class B> // A en B zijn parametriseerbare types
class MyClass {
  public:
    MyClass( A param ):
    ~MyClass( void );
    B MemberFunction( B param );
};
```

Je kan dan een object creëren met dit template als volgt (= *template instantiation*):

```
MyClass<long, char*> myObject( 250L );
```

Een functie-template ziet er bv. als volgt uit:

```
template <class T, class U>
T MyFunc( T param1, U param2 ) {
  T var1;
  U var2;
  ...
}
```

Beperking: de formele parameters moeten in de signatuur v.d. functie voorkomen.
Instantiatie: de compiler gebruikt de types in de functie-oproep om de formele parameters te bepalen.

Bv.:
```
short i;
long j;
i = MyFunc( i, j ); // Types voor T moeten overeenkomen!
```

Om een member function van een klasse-template te declareren:

```
template <class T>
Array <T>::Array( short size );
```

5 Object-georiënteerd programmeren voor gevorderden

5.1 Static Data Members

Je kan data members of member functions van een klasse als `static` declareren. Dit betekent dat er slechts één versie van wordt gecreëerd, die in alle objecten gebruikt wordt. Het is dus een soort globale variabele, maar dan enkel binnen zijn klasse.

Bv. `static short objectCount;`

Je kan zo'n data member accessen als: `Object::objectCount`

Een static data member moet gedefinieerd worden binnen dezelfde scope als de klassedeclaratie, bv. direct erna. Je kan ook static member functions declareren, maar dan moet je voor elke referentie naar een data member of functie een object gebruiken aangezien `this` niet van toepassing is.

5.2 Multiple Inheritance (Meervoudige overerving)

Het is mogelijk om een subklasse te maken die data members en member functions erft van meerdere klassen tegelijk.

Syntax: `class DerivedClass :` *`accessCode1`* `BaseClass1,` *`accessCode2`* `BaseClass2, ...`

Constructor: zoals bij enkelvoudige overerving, maar dan met een komma-lijst van de constructoren v.d. basisklassen. De volgorde bepaalt in welke volgorde deze constructoren opgeroepen worden, de volgorde v.d. destructoren is juist omgekeerd.

Als meerdere basisklassen een data member hebben met dezelfde naam, moet je de basisklasse specifiëren bij het gebruiken van deze naam: `BaseClassName::varName`

Als je een klasse afleidt van meerdere klassen die op zich van één klasse zijn afgeleid, moet je deze 'root'-klasse als '`virtual`' declareren in deze klassen, zoniet is er ambiguïteit over welk v.d. root-klasse-objecten nu gebruikt wordt. De verschillende constructies van objecten v.d. root-klasse zullen dan tot één gereduceerd worden, nl. enkel de constructor-oproep die het diepst in de 'derivation chain' zit, wordt gebruikt.

Gebruik: zet "`virtual`" vóór de naam van de root-klasse in de declaratie van de ambigue afgeleide klassen. Bv. als `Base1` en `Base2` afgeleid zijn van `Root`, declareer dan `Base1` en `Base2` als: `class Base1 : public virtual Root`
Als `Derived` dan afgeleid is van `Base1, Base2`, en de `Derived` constructor ziet er als volgt uit: `Derived::Derived( short param ) : Root( param );`, zal een eventuele parameter mapping van de `Base1` en `Base2` constructoren naar de `Root` constructor, genegeerd worden: er wordt dus maar één `Root`-object gecreëerd.

5.3 Initialisatie versus Assignment

Initialisatie gebeurt in de Member initialisation list van een constructor, assignment in de body van de constructor. Als je echter een data member als `const` of als een referentie-type wil declareren, moet de waarde bij initialisatie ingesteld worden. Dit kan enkel in de member initialisation list van de constructor, bv.:

```
class MyClass {
    const short kMaxNameLength;
    short &numberAlias;
    short number;
  public:
    MyClass( short constValue );
};
```

```
MyClass::MyClass( short constValue ) :
  kMaxNameLength( constValue ), numberAlias( number ) {
  // kMaxNameLength == constValue en numberAlias wijst naar number
}
```

5.4 Memberwise Initialisation

Dit kan gebruikt worden om een 'kloon' van een object te maken: de inhoud van elke data member wordt gekopieerd. Dit gebeurt standaard bij toekenning van een object naar een ander.

Bv.: `MyClass obj1( 20 );`
`MyClass obj2 = obj1;`

Let op! Als een data member een *pointer* bevat, wordt deze ook gekopieerd en verwijzen origineel en kopie dus naar hetzelfde object! Een oplossing hiervoor is een Memberwise initialisator-functie (*'copy constructor'* genoemd) te voorzien, waarin je dan de pointers realloceert. Dit is een ge-overloade versie van de constructor, van de vorm:

`ClassName( const ClassName& original );`

Je kan natuurlijk ook de '='-operator overloaden om te zorgen dat een uitdrukking als "`myObject = yourObject`" resulteert in een memberwise copy, bv.:

`Name &Name::operator=( const Name& original );`

Vergeet dan wel niet te checken of `original` en `this` niet hetzelfde object zijn alvorens je de oude inhoud van de pointers `deletet`!

5.5 Namespaces

Je kan klassen, functies, variabelen enz. declareren als behorende tot een zekere *namespace*. Dit kan door de declaraties en implementaties te omringen met:

```
namespace MyNamespace {
    ...
}
```

Om deze functies dan op te roepen, moeten ze voorafgegaan worden door "`MyNamespace::`", of moet in de headers "`using namespace MyNamespace;`" gezet worden.

Het voordeel van namespaces is dat je meerdere functies met dezelfde naam kan gebruiken, door ze toe te kennen aan een andere namespace. Dit kan bv. nuttig zijn om conflicten tussen twee libraries op te lossen zonder in de source code van die libraries zelf te moeten gaan prutsen. Je kan in zo'n geval nog steeds '`using`' gebruiken voor beide namespaces tegelijk, maar voor de conflicterende namen moet de scope operator wel gebruikt worden.

6 Werken met Streams: iostream

6.1 Basisklasse = 'ios'

Nodig: `#include <iostream>`
Afgeleide klassen: `istream, ostream` met `cin` (input), `cout` (output), `cerr` en `clog` (error en log) als objecten.

`stream` `<<` *`data`* : schrijf *`data`* naar *`stream`*
`stream` `>>` *`data`* : haal *`data`* uit *`stream`*.

6.2 Functies van istream

- `get()` :
  Leest 1 karakter van de input stream.

  `istream &get( char &destination );` : steekt de eerste byte in `destination` en geeft een referentie terug naar het `istream`-object, bv. `cin.get( c ) >> myShort;`

  `istream &get( char *buffer, int length, char delimiter = '\n' );` : verplaatst de eerste `length-1` bytes naar `buffer`, en sluit `buffer` af met een `NULL`. Als `delimiter` voorkomt, wordt de stroom er juist vóór afgebroken, bv. `cin.get( buffer, 10, '*' ) >> myShort;`

  `int get();` : leest één byte en geeft de waarde terug als een `int`, zodat EOF kan voorgesteld worden als -1.

- `istream &getline( char *buffer, int length, char delimiter = '\n' );`
  zoals de tweede versie van `get()`, maar stopt na `delimiter` (zonder deze mee te kopiëren).

- `istream &ignore( int length = 1, int delimiter = EOF );`
  negeert `length` bytes van de input stream, `delimiter` inbegrepen.

- `int peek();`
  zoals de derde versie van `get()`, maar laat de byte in de stream.

- `istream &putback( char c );`
  steekt `c` terug in de stream.

- `istream& seekg( streampos p );`
  zet de get-pointer op positie `p`.

  `istream& seekg( streamoff offset, relative_to direction );` : `direction` is ofwel `ios::beg` voor het begin v.d. stream, `ios::cur` voor de huidige cursorpositie, of `ios::end` voor het einde.

- `streampos tellg();`
  geeft de positie van de get-pointer.

6.3 Functies van ostream

- `ostream &put( char c )`
  zelfde functie als de `<<` operator voor één byte,
  bv.: `cout.put( 'H' ).put( 'I' ).put( '!'  );`

- `seekp`
  idem als `seekg` maar dan voor de get-pointer.

- `streampos tellp();`
  geeft de positie van de put-pointer.

6.4 Nuttige utilities uit <ctype.h>

`int isalnum( int )`	True if `isalpha( int )` or `isdigit( int )`.
`int isalpha( int )`	Is in range a-z or A-Z?
`int iscntrl( int )`	Is control character?
`int isdigit( int )`	Is char in range 0-9?
`int isgraph( int )`	True if char isalpha() or isdigit() or ispunct().
`int islower( int )`	Is char in range a-z?
`int isprint( int )`	Is char printable ASCII character?
`int ispunct( int )`	Is char in ASCII range 33-47, 58-64, 91-96, 123-126?
`int isspace( int )`	Is char in ASCII range 9-13 or 32?
`int isupper( int )`	Is char in range A-Z?
`int isxdigit( int )`	Is char in ASCII range 0-9, a-f, A-F? (Hex)
`int tolower( int )`	Maps upper case to lower case.
`int toupper( int )`	Maps lower case to upper case.

6.5 Werken met bestanden

Nodig: `#include <fstream>`
`ifstream` en `ofstream` zijn subklassen van `istream` en `ostream` respectievelijk (dus al het bovenstaande geldt).

6.5.1 Constructor

`ifstream( const char* name, int mode = ios::in );` *(analoog voor* `ofstream`*)*
`name` is de bestandsnaam, `mode` is één of een combinatie (d.m.v. `|`) van de volgende:

`ios::in`	input toegelaten,
`ios::out`	output toegelaten,
`ios::ate`	ga naar EOF bij openen,
`ios::app`	output toegelaten maar enkel achter de huidige data (append),
`ios::trunc`	output toegelaten, vervang huidige inhoud,
`ios::nocreate`	open faalt als het bestand niet bestaat,
`ios::noreplace`	open faalt als het bestand al bestaat.

Vb.:
```
ifstream readMe( "My File");
char c;
while( readMe.get( c ) )  // zie state bits: good() wordt gebruikt
   cout << c;
```

Om een file tegelijk voor lezen & schrijven te openen, gebruik `fstream`:
`fstream inAndOut( "Read and Write", ios::in | ios::out );` (zie fstream.h !)

Een bestand wordt automatisch gesloten als zijn object vernietigd wordt, maar kan ook expliciet gesloten worden met `close()`, bv.: `writeMe.close();`. Je kan ook eerst een stream declareren en dan pas een bestand openen met `open`, bv.:

```
ifstream readMe;
readMe.open( "File 1" );
...
readMe.close(); }
```

6.5.2 State bits

Elk `istream` en `ostream` object heeft 4 state bits, deze kunnen als volgt ge-accessed worden:

`int good();`	is `true` als de stream klaar is voor I/O.
`int eof();`	is `true` als de laatste I/O operatie het einde v.d. file bereikt heeft.
`int fail();`	is `true` als de laatste operatie gefaald is (bv. letter ingelezen terwijl het een `short` moest zijn).
`int bad();`	is `true` als de laatste operatie gefaald is en de stream corrupt is.

`void clear( int newState = 0 );` : zet de state bits op `newState`; meestal gewoon `clear()` gebruiken om terug op initiële waarden te zetten. Dit is nodig om opnieuw te kunnen lezen/schrijven wanneer `good() != 1`.

6.5.3 Nuttige istream/ostream functies voor files:

- `istream &read( void *data, int size );`
  leest `size` bytes en steekt ze in de buffer `data`. Als een EOF tegengekomen wordt, wordt de `fail` bit op 1 gezet en geeft de functie `size_t istream::gcount()` het aantal succesvol gelezen bytes.
- `ostream &write( const void *data, size_t size );`
  omgekeerd, `size_t ostream::pcount()` geeft het aantal bytes geschreven door de voorgaande `write()` oproep.

6.6 Tips voor gevorderden

6.6.1 Overloaden van >> en <<

Het linkerargument van `>>` is altijd een `istream` object en van `<<` een `ostream` object. Om sequenties van deze operatoren te ondersteunen, moet de functie dus altijd deze eerste parameter teruggeven.

Bv.:
```
istream &operator>>( istream &is, MenuItem &item ) {
  ...
  return( is );
}
```

6.6.2 ios Flags

Met `ios` flags kan het formaat van de output veranderd worden. Een ios flag kan gezet of verwijderd worden met `setf()` en `unsetf()` respectievelijk.

Vb.: `cin.setf( ios::skipws );`

Sommige flags zijn gegroepeerd: door de groep als tweede parameter mee te geven aan `setf()`, worden al de andere flags van de groep op 0 gezet, bv. `cout.setf(ios::hex, ios::basefield);`

Alle ios flags kunnen opgevraagd worden als één getal met `long ios::flags();` en ingesteld worden met `long ios::flags( long flags );`.

Formatting flags:

- `skipws` : geeft aan of white space verwijderd wordt tijdens extractie-operaties.
- `dec, hex, oct` ∈ `basefield` : bepalen de radix van de getallen in de output.
- `left, right, internal` ∈ `adjustfield` : worden samen met `width()` gebruikt, wat het min. aantal karakters bepaalt in de volgende output (string of getal). Als bv. `left` aan is, wordt het getal links gealigneerd. De rest wordt opgevuld met het karakter gespecifieerd met `fill()`. `internal` plaatst de opvulkarakters tussen een eventueel teken en het getal

zelf. Na elk getal/string wordt `width()` terug op 0 gezet.
Bv.:
```
cout.width( 5 );
cout.fill( '0' );
cout.setf( ios::right, ios::adjustfield );
cout << 202;
```
geeft "00202".

- `showbase` : bepaalt of octale getallen met een 0 en hex getallen met 0x beginnen.
- `showpoint` : bepaalt of nullen a.h. einde van floating-point getallen getoond worden (bv. `0.123000`).
- `uppercase` : bepaalt of `E` dan wel `e` gebruikt wordt in wetenschappelijke notatie en `X` dan wel `x` voor hex.
- `showpos` : bepaalt of positieve getallen expliciet door '+' voorafgegaan worden.
- `scientific, fixed` ∈ `floatfield` : bepaalt of wetenschappelijke of vaste-komma notatie altijd gebruikt wordt. Als beide vlaggen op 0 staan (met `cout.set( 0, ios::floatfield );`), wordt wetensch. notatie enkel voor zeer grote of kleine getallen gebruikt. `precision()` bepaalt het aantal beduidende cijfers in het getal.
- `unitbuf` : bepaalt of de output buffer ge-flushed wordt na elke output-operatie.
- `stdio` : (voor C I/O) doet hetzelfde voor `stdout` en `stderr`.

6.6.3 iostream Manipulators

Deze laten toe een I/O operatie in het midden van een insertie of extractie te doen.
Bv.: `cout << "Enter a number: " << flush;`
Een manipulator kan ook opgeroepen worden als `cout.flush();` of `flush( cout );` (voor manipulators met parameters is `#include <iomanip>` nodig).

`flush()`	maakt de buffer van de stream leeg.
`dec(), oct(), hex()`	zetten de overeenkomstige flag van de `basefield` groep aan.
`endl()`	zet een `'\n'` in zijn output stream en flusht dan de stream.
`ends()`	zet een `NULL` in zijn output stream en flusht dan de stream.
`ws()`	vreet alle white space op in zijn input stream tot aan EOF of een niet-whitespace karakter.
`setbase( int b )`	zet de radix op 8, 10 of 16.
`setfill( int f )`	zie `fill()` bij ios flags.
`setprecision( int p )`	zie `precision()`.
`setw( int w )`	zie `width()`.
`setiosflags( long f )`	zie `setf()`.
`resetiosflags( long f )`	zie `unsetf()`.

6.6.4 istringstream & ostringstream

Nodig: `#include <sstream>`

`istringstream` en `ostringstream` zijn subklassen van istream en ostream en laten toe om I/O-functies toe te passen op een buffer, zoals de C-functie `sprintf()`, maar dan met de flexibiliteit van streams. (Deze klassen vervangen `strstream` met `istrstream` en `ostrstream`. In tegenstelling tot `strstream` doet `sstream` al het geheugenbeheer zelf.)

Constructoren:
```
ostringstream( openmode mode = out );
ostringstream( const string &str, openmode mode = out );
```

`istringstream( openmode mode = in );`
`istringstream( const string &str, openmode mode = in );`
Mogelijkheden voor `mode`:

`app`	(append) Seek to the end of the stream before each output operation.
`ate`	(at end) Seek to the end of the stream when opening.
`binary`	Consider stream as binary rather than text.
`in`	Allow input operations on a stream.
`out`	Allow output operations on a stream.
`trunc`	(truncate) Truncate file to zero when opening.

I.p.v. een `string` object mag `str` ook een pointer zijn naar een zero terminated `char` buffer.

Met elk van deze objecten is een `stringbuf` object geassocieerd, wat opgevraagd kan worden met de functie `stringbuf* rdbuf()`. Met de functie `string str()` en de methode `void str( string &s )` kan een kopie van het `string` object van de buffer opgevraagd worden, respectievelijk een `string` naar de buffer gekopieerd worden.