

C++

- C++ als een verbetering van C
- Abstracte datatypen met `classes`
- Constructoren en destructoren
- Subklassen
- binding van functies

Commentaar

In C:

```
/* Deze functie berekent
   de omtrek van een cirkel
*/
float omtrek(float r)
{ /* r is de straal */
  return (2*pi*r);
}
```

In C++:

```
// Deze functie berekent
// de omtrek van een cirkel

float omtrek(float r)
{ // r is de straal
  return (2*pi*r);
}
```

Vermijd #define

In C:

```
#define pi      3.14159
#define false  0
#define als    if
```

In C++:

```
const float pi      = 3.14159;
const int  false    = 0;

const int  *pf      = &>false;
```

Ander gebruik van const:

```
char *const s = "hallo";
```

Vermijd #define

In C:

```
#define sq(x) x*x
```

```
sq(1+2)
```

Lapmiddel:

```
#define sq(x) ((x)*(x))
```

In C++ functie-syntax:

```
inline int sq(int x)
{ return x*x;
}
```

Declaraties

In C:
aan het begin van een functie

In ANSI-C:
aan het begin van een blok

In C++:
overal in het blok
(dichtbij het gebruik)

Typedeclaraties

In C:

```
struct punt { int x; int y; }  
void f(struct punt p);
```

Lapmiddel:

```
typedef struct punt { int x; int y; } PUNT;  
void f(PUNT p);
```

In C++:

```
struct punt { int x; int y; }  
void f(punt p);
```

Call by reference

In C:

```
void wissel( int *x; int *y)
{  int h;
   h = *x; *x = y; *y = h;
}
main()
{  int a, b;
   wissel( &a, &b );
}
```

In C++:

```
void wissel( int &x; int &y)
{  int h;
   h = x; x = y; y = h;
}
main()
{  int a, b;
   wissel( a, b );
}
```

Overloading

In C:

```
int iabs(int x)
{ return (x<0 ? -x : x);
}
float fabs(float x)
{ return (x<0.0 ? -x : x);
}
main()
{ ... iabs(3) + fabs(-2.7) ...
}
```

In C++:

```
int abs(int x)
{ return (x<0 ? -x : x);
}
float abs(float x)
{ return (x<0.0 ? -x : x);
}
main()
{ ... abs(3) + abs(-2.7) ...
}
```

Default-parameters

Hé, overloading is handig:

```
float macht(float x; int n)
{  r = 0;
   for (i=0; i<n; i++)
       r *= x;
   return r;
}
float macht(float x)
{  return macht(x,2);
}
```

Maar nog handiger is:

```
float macht(float x; int n=2)
{  r = 0;
   for (i=0; i<n; i++)
       r *= x;
   return r;
}
```

Dynamische allocatie

In C:

```
int *data;  
data = (int*) malloc( n * sizeof(int) );  
f(data[4]);  
free(data);
```

In C++:

```
int *data;  
data = new int[n];  
f(data[4]);  
delete data;
```

Abstracte datatypes in C

```
struct stack
{
    char info[100];
    int top;
};

void reset(stack *s)
{
    s->top = -1;
}

void push(stack *s, char c)
{
    s->top++;
    s->info[s->top] = c;
}

char top(stack *s)
{
    return s->info[s->top];
}

main()
{
    struct stack s;
    /* bedoeld gebruik */
    reset(s);
    push(s, 'a');
    /* onbedoeld gebruik */
    s.info[73] = 'x';
}
```

Abstracte datatypes in C++

```
struct stack
{
private:
    char info[100];
    int top;
public:
    void reset(void) { top = -1;          }
    void push(char c){ top++; info[top]=c; }
    char top(void)   { return info[top];  }
};

main()
{
    stack s;
    s.reset();
    s.push('a');
}
```

structs in C++

Bevatten naast data ook functies
(*memberfuncties*)

Declaraties zijn

- `private`: alleen door memberfuncties te gebruiken
- `public`: bereikbaar met punt-notatie

Implementatie:

`private/public` beïnvloedt alleen scope

memberfuncties worden niet echt opgeslagen

memberfuncties hebben stiekem
extra parameter

classes in C++

Verschil alleen in default-protectie:

- in een struct
declaraties zijn default public
- in een class
declaraties zijn default private

```
class stack
{
    char info[100];
    int top;
public:
    void reset(void) { top = -1;          }
    void push(char c){ top++; info[top]=c; }
    char top(void)   { return info[top];  }
};
```

Object-georiënteerd programmeren

Instances van classes heten *objecten*.

Imperatief programmeren:

Functies (met objecten als parameter)

Object-georiënteerd programmeren:

Objecten (met functies als member)

Ècht object-georieënteerd wordt het pas bij *dy-*
namische binding van memberfuncties.

Declaratie van memberfuncties

Direct in de klasse-declaratie:

```
class c
{ int x;
public:
    int f(void) { return 2*x; }
};
```

In de klasse-declaratie alleen prototype, functiedeclaratie later apart:

```
class c
{ int x;
public:
    int f(void);
};
int c::f(void) { return 2*x; }
```

Notatie `c::f` heet *scope-resolutie*

static variabelen in C

static lokale variabelen van functies worden gedeeld door alle aanroepen

```
void f(void)
{
    static int n=0;
    printf("ik ben %d keer\
          aangeropen", n++);
}
```

Implementatie:

globale variabele met beperkte scope

static members in C++

static members van classes
worden gedeeld door alle instanties

```
class c
{
    static int n;
    int x; int y;
public:
    void f() { n++; }
    void g() { n++; }
    void h() { printf
                ("%d keer memberfunctie\
                 gebruikt"
                 , n); }
};
```

Implementatie alweer:
globale variabele met beperkte scope

Constructors

Initialisatie van een instance van een klasse door *constructorfunctie*

Heeft dezelfde naam als de klasse, en geen resultaattype

```
class stack
{ char info[100];
  int top;
public:
  stack(void) { top = -1; }
};
```

Constructorfunctie wordt automatisch aangeroepen bij creatie van de variabele

```
main()
{ stack s, *p;
  ...
  p = new stack;
}
```

Constructors met parameters

Handig voor initialisatie

Parameters worden meegegeven bij declaratie of dynamische creatie

```
class punt
{  int x;
   int y;
public:
   punt(int x0, int y0)
   {  x=x0; y=y0; }
};
main()
{  punt hier(12,5), *p;
   ...
   p = new punt(2,6);
}
```

Dynamisch geheugen als members

Allocatie in constructorfunctie

De-allocatie in *destructorfunctie*

```
class stack
{  char *info;
   int top;
public:
   stack(int n)
   {  info = new char[n];
      top = -1;
   }
   ~stack(void)
   {  delete info;
   }
};
```

Destructor wordt automatisch aangeroepen
als het object verdwijnt

Classes voor open/sluit-constructies

In C:

```
File f;  
f = open("aap");  
seek(f, pos);  
read(f, data);  
close(f);
```

Met classes in C++:

```
{ File f("aap");  
  f.seek(pos);  
  f.read(data);  
}
```

Nog een open/sluit-constructie

In het MS-Windows programma van vorige week

In C:

```
data = GlobalLock(handle);  
... data->x ...  
GlobalUnlock(handle);
```

Met classes in C++ zou dit zo kunnen:

```
{ Lock data(handle)  
  ... data.x ...  
}
```

met in de constructor GlobalLock
en in de destructor GlobalUnlock

Toepassing: veilige arrays

```
class Vector
{   int *p;
    int size;
public:
    Vector(int n=10) { size=n; p=new int[n]; }
    ~Vector(void)    { delete p; }
    int &elem(int);
};

int & Vector::elem(int i)
{   if (i<0 || i>=size) printf("bound error");
    return p[i];
}

main()
{   Vector a(10), b(5);
    a.elem(1) = 17;
    b.elem(1) = a.elem(1) + 9;
}
```

Members die zelf objecten zijn

Probleem:

```
class Twee
{
    Vector heen;
    Vector terug;
public:
    Twee(int i)
    {
    }
};
```

wat is de parameter van de constructoren van de members?

Oplossing: speciale syntax hiervoor:

```
Twee(int i) : heen(i), terug(i)
{
}
```

Pointer naar this object

```
class klasse;  
void f(klasse *obj);
```

Pointer naar object is impliciete parameter van member-functies:

```
class klasse  
{ int x;  
public:  
    void g(void);  
    void h(void)  
    { // impliciete parameter  
      // gebruikt voor toegang  
      // tot andere members  
      ... x    ...  
      ... g() ...  
      // moet expliciet genoemd  
      // om 'dit' object door te  
      // geven aan externe functies  
      ... f(this) ...  
    }  
};
```

Derived classes

Derived classes breiden een klasse uit met extra members.

Memberfuncties mag je herdefiniëren.

```
class Persoon
{   public:
    char naam[20];
    int  gebJaar;
    int  leeftijd(void);
    void print(void);
};
class Student : Persoon
{   public:
    char studie[10];
    void print(void);
};
main()
{   Student s;
    ... s.naam    ...
    ... s.studie ...
    s.print();
}
```

Constructoren voor derived classes

Voorbeeld: array met ondergrens

```
class Vector
{
public:
    Vector(int);
    ~Vector(void)
    int &elem(int);
};
```

```
class BndVector : Vector
{ int eerste;
public:
    BndVector(int lo, int hi) : Vector(hi-lo)
    { eerste = lo; }
    int &elem(int i)
    { return Vector::elem(i-eerste); }
};
```

Binding van hergedefinieerde functies

Binding van functies gebeurt statisch, dus op grond van het compile-time type:

```
class A
{ public:
  int f(void){ return 1; }
};
class B : A
{ public:
  int f(void){ return 2; }
};
main()
{ A a;
  B b;
  a.f(); // levert 1
  b.f(); // levert 2
}
```

Derived klassen zijn subtypen

Binding blijft statisch

```
main()
{  A a, *pa;
   B b;

   pa = &a;    // dit mag
   pa->f();    // levert 1

   pa = &b;    // dit mag ook!
   pa->f();    // wat levert dit?
}
```

virtual functies

Dynamische binding van member-functies kan met virtual functies:

```
class A
{ public:
    virtual int f(void){ return 1; }
};
class B : A
{ public:
    int f(void){ return 2; }
    // type moet hetzelfde zijn als A::f
};

main()
{   A a, *pa;   B b;

    pa = &a; pa->f();    // levert 1
    pa = &b; pa->f();    // levert nu 2!
}
```

Implementatie van `virtual`

Memberfuncties die `virtual` zijn, worden wèl in de datastructuur opgeslagen.

Bij creatie is het type, en dus de gewenste functie bekend.

bij aanroep wordt de gewenste functie in de datastructuur opgezocht.

Kost iets meer tijd en ruimte, maar voor de programmeur erg makkelijk.