

Introduction to Programming

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

The course in a Nutshell

- We continue from *Introduction to Computer Science*.
- We use the C++ programming language.
- The course consists of:
 - ★ Lecture
 - ★ Programming Assignments

Course Material

- Lecture slides available on the WWW page (after the lecture)
 - Book:
 - J. Liberty.
 - Teach Yourself C++ in 10 Minutes.
 - Sams Publishing, 1999 — or: Second edition, 2002
- in the university library

Examination / Grading

- Written exam: **Wednesday, September 17, 2003**
maximum score: 50 points
- Programming assignments (during the term)
maximum score: 50 points
- With a total of 50 points you have passed the course (grade 4.0).
- With more points, your grade gets better (up to 1.0)

Programming Assignments

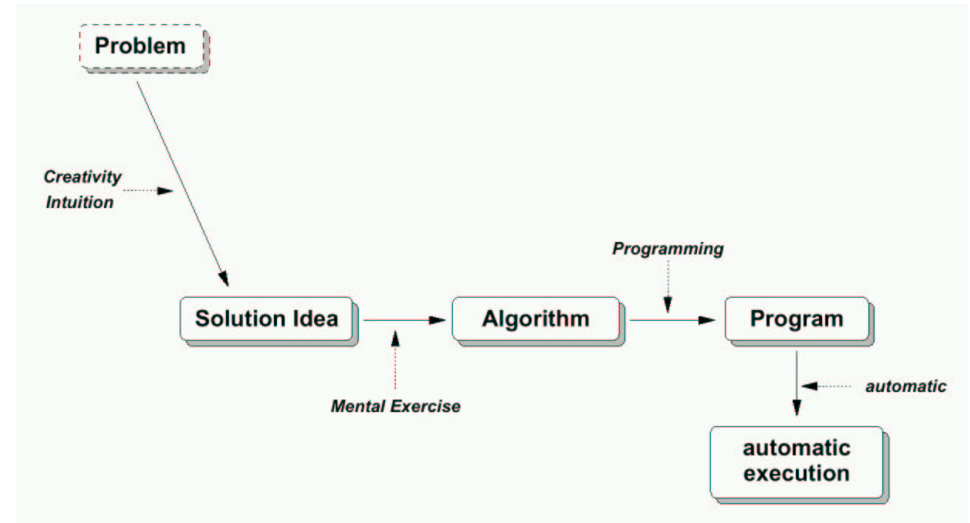
- In H-C 8327, free schedule
- Introduction: **Tue, May 06, 14:15, in H-C 8327**
- Put your name on the list for accounts!
- 5 assignments in total, demonstrate your solutions on the PC's in H-C 8327 (you may use your own computer, but you'll have to bring your program...)
- Get your programs accepted/graded by Frank Thilo!

Lecture

May:	02/05	09/05	16/05	23/05
June:		13/06	20/06	
July:	04/07	11/07	18/07	25/07

Exam: 17/09

From Problem . . . to Solution



Algorithms

An algorithm:

- . . . is a set of instructions
- . . . with a precise statement (\equiv meaning)
- . . . of effective (\equiv actually executable) processing steps
- . . . noted in a precisely defined language

Algorithms

An algorithm:

- ... is a set of instructions
- ... with a precise statement (\equiv meaning)
- ... of effective (\equiv actually executable) processing steps
- ... noted in a precisely defined language

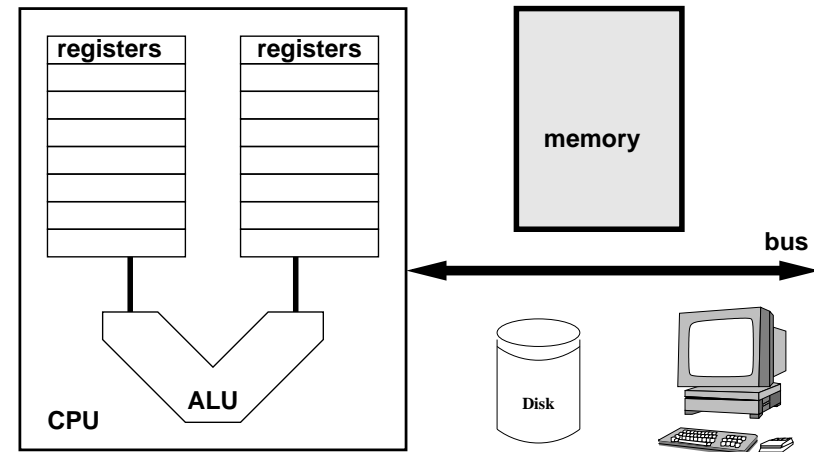
Or:

An algorithm is a prescription **how** to solve a problem using individual, small steps.

Plumcake Algorithm

1. **Get** the ingredients.
2. **First** clean the flour with a sieve.
3. **Then** add sugar, milk, butter, eggs, and salt.
Mix well and stir **until** the dough starts having bubbles.
4. **Meanwhile** wash the plums, cut open, and remove pit.
5. . . .
6. **If** using circulating oven, bake at 180 C for 20 minutes.
Otherwise, heat oven to 210 C . . .

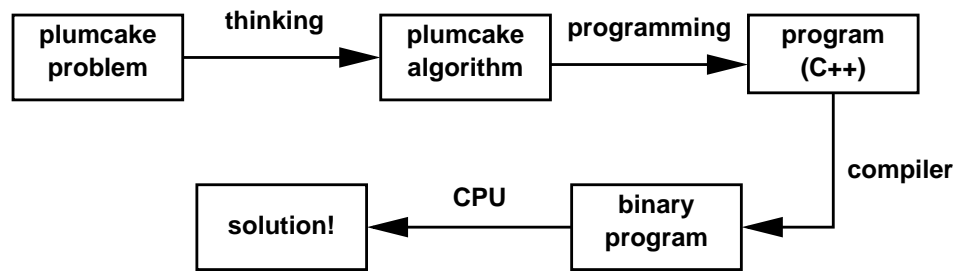
The Computer from Inside (VERY simplistic)



How the CPU works

- Both binary program and data are in the memory, (a sequence of cells e.g., bytes)
- Execution: load OP from memory[PC] to CPU, execute it, increment PC
- Operation:
 - ★ load memory[address] to register
 - ★ perform operation (in the ALU) between registers
 - ★ store register content to memory[address]
 - ★ change PC

From Problem . . . to Solution



“One week of programming can easily save one hour of thinking!”

Or: programming is **not** trial-and-error!

The C++ Programming Language

- Imperative:
Prescribes exactly which statements have to be executed in which order.
- Object-oriented:
Allows the use of objects and classes.

C++ is in between problem description and CPU operation.

(Many say that it is closer to the CPU. . .)

The first C++ Program

```

#include <iostream.h>

using namespace std;

int main(){
    cout << "Hello, world!\n";
    return 0;
}
  
```

All (special) characters are important.

C++ is case sensitive: `return` \neq `Return`
(now try this with the compiler)

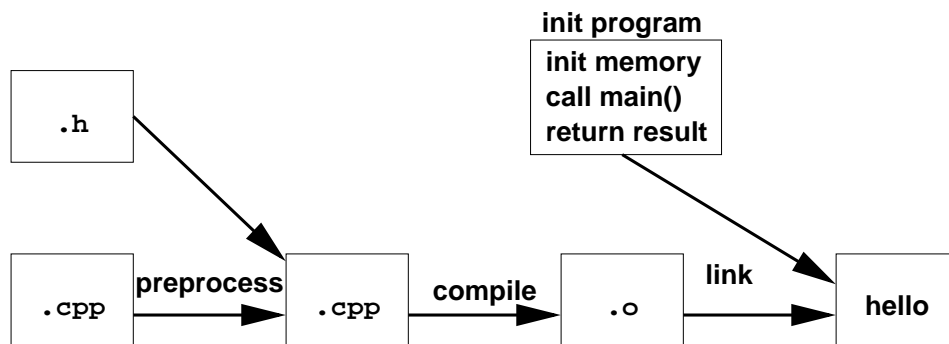
Parts of a Simple Program

- `#include` – preprocessor directive
include the contents of another file
- `using namespace` – where to search unknown names
- `main()` – function
starting point of a program
- statements – prescription what to do
- Comments:
`// comment from here to the end of the line`
`/* comment from here up until */`

The main() Function

- conventional: tells the compiler where the program starts
- is of type int
- return value is given back to operating system after the program has terminated (ended)
- the program executes the statements inside main(), exactly in the given order, from the beginning (top) to the end (bottom)

Compiling a Program



Functions

```

#include <iostream.h>
using namespace std;
int Add ( int x, int y){
    return (x+y);
}
int main(){
    int a,b,c;
    cout << "Enter two numbers: ";
    cin >> a;    cin >> b;
    c = Add(a,b);
    cout << "The sum is " << c << endl;
    return 0;
}
  
```

Functions

```

#include <iostream.h>
using namespace std;
int Add ( int x, int y){
    cout << "In Add(), received " << x << " and "
        << y << "\n";
    return (x+y);
}
  
```

```
int main(){
    cout << "I'm in main()!\n";
    int a,b,c;
    cout << "Enter two numbers: ";
    cin >> a;    cin >> b;
    cout << "\nCalling Add()\n";
    c = Add(a,b);
    cout << "\nBack in main().\n";
    cout << "The sum is " << c << endl;
    cout <<"Exiting...\n\n";
    return 0;
}
```

Functions

- Statements (e.g., in `main()`) are executed in their order in the code.
- When a function is called, the program **branches** to the beginning of it.
- A function consists of header and body.
Header: *type name (parameter list)*
Body: { *list of statements* }
- A function returns when it executes the `return` statement (or when reaching the end of its body).

Course Overview

- data types, variables, operators, functions, parameters
- if, switch, loops, arrays
- classes and objects
- pointers, references, garbage collection
- inheritance and polymorphism
- linked lists, trees, recursive algorithms
- templates, exception handling

Next Lecture

Simple calculations

- variables and data types
- statements, operators
- input

Introduction to Programming, Lesson 2

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

Lecture Schedule

May:	02/05	09/05	12/05	19/05	26/05
June:	02/06		16/06		30/06
July:		14/07	21/07		

Next lecture: monday morning, 8 a.m. (same room)!

Exam: 17/09, 10:00–11:00, room AR - D 5104

Course Outline

02/05 introduction, "Hello, world"
09/05 expressions, statements, if
12/05 exception handling, functions, separate compilation
19/05 switch, loops, arrays
26/05 types, structs, classes, file I/O
02/06 memory, heap/stack, pointer, testing
16/06 constructors, destructors, shallow and deep copies
30/06 inheritance, abstract classes
14/07 lists, queues, trees (as C++ classes)
21/07 templates, container classes, STL

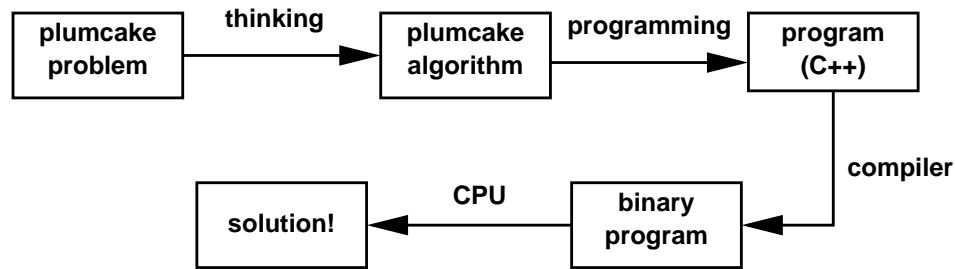
Reminder: The first C++ Program

```
#include <iostream.h>

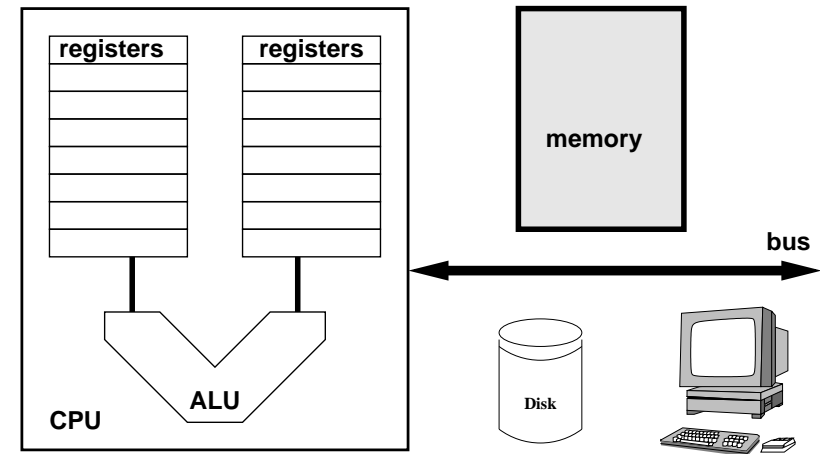
using namespace std;

int main()
{
    cout << "Hello, world!\n";
    return 0;
}
```

Reminder: From Problem . . . to Solution



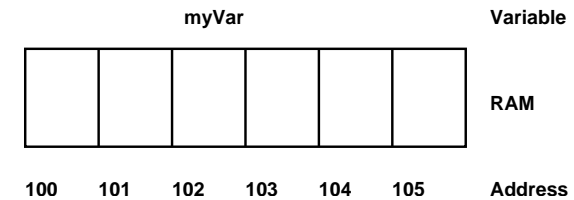
The Computer from Inside (VERY simplistic)



Today:

- variables
- data types
- constants
- statements
- operators
- if statement

Variables



Each memory cell has the size of one byte.
Variables give **name** and **type** to memory locations.

Simple Data Types and their Sizes

int	4 bytes	depends on the machine!
short int	2 bytes	
long int	4 bytes	
float	4 bytes	
double	8 bytes	
char	1 byte	
bool	1 byte	

signed and unsigned Integers

- **int** is a *signed* integer:
... - 3, -2, -1, 0, 1, 2, 3, ...
- **unsigned short int** is an *unsigned* integer:
0, 1, 2, ..., 65534, 65535

Value Ranges

type	size	range
unsigned short int	2 bytes	0 to 65,535
short int	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int	4 bytes	-2,147,483,648 to 2,147,483,647
float	4 bytes	1.2e-38 to 3.4e38
double	8 bytes	2.2e-308 to 1.8e308
char	1 byte	256 (ASCII) character values
wchar_t	4 bytes	large character sets
bool	1 byte	true or false

Caution:

Precision of arithmetic is limited!

- With **int** values, the effects can be handled.
- With **float** or **double** values, the lack of precision can ruin the results of your program!

Don't worry:

all you need is keep this in mind and be careful!

Defining a Variable

Type, followed by the name, followed by ';'.

Example: **unsigned long int myAge;**

The name can be any sequence of characters
a...z, A...Z, 0...9, _

Remember: C++ is case-sensitive:

myAge is not the same name as **myage**

Not allowed as names are keywords, like **int**, **if**, **while**

Creating more than one Variable at a Time

```
unsigned int myAge, myWeight; // two unsigned ints
long area, width, length;    // three longs
```

Use names that make sense!

Use names that have sufficient **intellectual distance!!!**

Bad example: **unsigned int x1, x2, x3;**

Good example: **unsigned int myAge, myWeight;**

Assignment and Initialization

Assignment:

```
int Width;
```

```
...
```

```
Width = 5;
```

Initialization (=creation with initial assignment):

```
int Width = 5;
```

Before the first assignment,

the content of a variable is undefined!

Always make sure that variables are initialized **before** you use them!

Literal Constants

- 42
- -23.5e16
- true
- 'a'
- "this is a string\n"

Use literal constants only very carefully!

Symbolic Constants

Literal:

```
students = classes * 15;
```

Symbolic:

```
const unsigned short int studentsPerClass = 15;  
students = classes * studentsPerClass;
```

Advantages:

1. prepare for change
2. compiler can type-check operations on constant

Enumerated Constants

```
enum COLOR { RED, GREEN, BLUE, BLACK,  
            WHITE, CYAN, MAGENTA };  
COLOR c = GREEN;
```

(The enumerated constants get assigned int values. However, this is hardly ever useful.)

Statements

Variables are the program's data.

Statements do something with the data.

- expression: **x = a + b;**
- empty statement: (white space and/or ;)
- compound statement: { statement sequence }

A program executes **main()**'s compound statement, from the beginning to the end.

Expressions

Anything that returns a value is an **expression** in C++.

An **operator** is a symbol that causes the program to perform an operation.

The **assignment operator** (=) changes the value of the left operand to the value of the expression on the right side.

```
x = a + b
```

l-values and r-values

- *l-value*: anything that can be used on the left side of an assignment
- *r-value*: anything that can be used on the right side of an assignment
- (all l-values are r-values)

Example: `35 = x` is *NOT* correct, 35 is not an l-value

Mathematical Operators

- On floats: `+` `-` `*` `/`
- On integers: `+` `-` `*` `/` `%`
 - ★ `21 / 4` \rightarrow 5
 - ★ `21 % 4` \rightarrow 1

Combined Assignment and Arithmetic

- Common example: `myAge = myAge + 2;`
- same as: `myAge += 2;`
- also defined: `--` `*=` `/=` `%=`
beware: *if the meaning of these combined operators is not obvious, then you better write it explicitly!*

Increment and Decrement

- `c = c + 1` is the same as `c++` \Rightarrow language name!
- `c = c - 1` is the same as `c--`

Beware: prefix and postfix operators:

Prefix: `++c` returns the incremented value

Postfix: `c++` returns the old (not yet incremented) value
if `x` has the value 5:

after `a = x++;` `a` has the value 5, and `x` is 6

after `b = ++x;` `b` has the value 7, and `x` is 7

Precedence

- * / % have higher precedence (“priority”) then + -
- arithmetic operators with the same precedence go from left to right
- assignment operators go from right to left
- parantheses () override precedence

Example: $x = 5 + 3 + 8 * 9 + 6 * 4$

Precedence (2)

1. Use precedence only for * / % + - (common sense)
2. Forget about the rest of precedence rules and use parantheses!

Relational Operators

name	op	example	result
equals	==	100 == 50	false
		100 == 100	true
not equals	!=	100 != 50	true
		100 != 100	false
greater than	>	100 > 50	true
		100 > 100	false
greater or equal	>=	100 >= 50	true
		100 >= 100	true
less than	<	100 < 50	false
		100 < 100	false
less or equal	<=	100 <= 50	false
		100 <= 100	true

The if Statement

- Normally, statements are executed in the order they appear in the program (top to bottom).
- Sometimes, statements shall only be executed **if** some condition holds:


```
if ( expression )
    statement;
```
- Example:


```
if ( minimum > new_value )
    minimum = new_value;
```

The else Clause

- Often, some statement shall be executed if an expression holds, and some other code, if not:


```
if ( a < b )
    minimum = a;
if ( b < a )
    minimum = b; // What is wrong here?
```
- if (a < b)


```
minimum = a;
else
    minimum = b; // also if a == b!
```

Expressions used for if (sorry for that)

- every numerical value is “allowed”;
0 means “false”
everything else (usually “1”) means “true”
(That is a “heritage” from the C language. . .)
- However: only use meaningful expressions!
- For example, use `if (minimum > 0)`
rather than `if (minimum) !`

Nested if Statements

```
int main(){
    bool a = true, b = false;
    if ( a )
        if ( b )
            cout << "a and b\n";
    else
        cout << "not a\n";
    return 0;
}
```

What does the program print to cout?

Nested if Statements (2)

```
int main(){
    bool a = true, b = false;
    if ( a ){
        if ( b ){
            cout << "a and b\n";
        }
    }
    else {
        cout << "not a\n";
    }
    return 0; // And what does this program print ??
}
```

Always use { and } for if and else clauses!

More Complex Expressions

Logical Operators:

Operator	Symbol	Example
and	&&	expression1 && expression2
or		expression1 expression2
not	!	!expression

Examples:

```
if ( (x == 5) && (y == 5) )
```

```
if ( (x == 5) || (y == 5) )
```

```
if ( !(x == 5) ), same as if ( x != 5 )
```

Next Lesson

- exception handling
- functions
- separate compilation

Introduction to Programming, Lesson 3

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

Reminder: Statements

Variables are the program's data.

Statements do something with the data.

- expression: **x = a + b;**
- empty statement: (white space and/or ;)
- compound statement: { statement sequence }

A program executes **main()**'s compound statement, from the beginning to the end.

Reminder: if Statement

```
int main(){
    bool a = true, b = false;
    if ( a ){
        if ( b ){
            cout << "a and b\n";
        }
    }
    else {
        cout << "not a\n";
    }
    return 0;
}
```

Today:

- Input, input errors
- Exception handling
- Modules (separate compilation)
- Functions and their parameters
- Recursion

A Simple Calculator (1)

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
    int ReturnCode = 0;

    float Dividend = 0;
    cout << "Dividend: ";
    cin >> Dividend;

    float Divisor = 1;
    cout << "Divisor: ";
    cin >> Divisor;
```

A Simple Calculator (2)

```
float Result = (Dividend/Divisor);
cout << Result << endl;

char StopCharacter;
cout << endl << "Press a key and \"Enter\": ";
cin >> StopCharacter;

return ReturnCode;
}
```

Catching the Error (1)

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
    int ReturnCode = 0;

    float Dividend = 0;
    cout << "Dividend: ";
    cin >> Dividend;

    if ( !cin.fail() ){
        float Divisor = 1;
        cout << "Divisor: ";
        cin >> Divisor;

        float Result = (Dividend/Divisor);
        cout << Result << endl;
    }
```

Catching the Error (2)

```
else{
    cerr << "Input error, not a number" << endl;
    cin.clear();
    char BadInput[80];
    cin >> BadInput;
    ReturnCode = 1;
}
char StopCharacter;
cout << endl << "Press a key and \"Enter\": ";
cin >> StopCharacter;

return ReturnCode;
}
```

Catching both Errors

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
    int ReturnCode = 0;
    float Dividend = 0;
    cout << "Dividend: ";
    cin >> Dividend;
    if ( !cin.fail() ){
        float Divisor = 1;
        cout << "Divisor: ";
        cin >> Divisor;
        if ( !cin.fail() ){
            float Result = (Dividend/Divisor);
            cout << Result << endl;
        }
        else{
            cerr << "Input error, not a number" << endl;

```

```

        cin.clear();
        char BadInput[80];
        cin >> BadInput;
        ReturnCode = 1;
    }
}
else{
    cerr << "Input error, not a number" << endl;
    cin.clear();
    char BadInput[80];
    cin >> BadInput;
    ReturnCode = 1;
}
char StopCharacter;
cout << endl << "Press a key and \"Enter\": ";
cin >> StopCharacter;
return ReturnCode;
}

```

Error Handling

- Professional code needs to be “robust”
- Input errors need to be taken care of
- But: “normal” and error handling code gets mixed, neither is easy to understand afterwards

Exception Handling

```
try{
    // normal code
}
catch (...){
    // error handling
}

```

- modern libraries “throw” exception instead of using return codes
- details need to be taken from library documentation (e.g., for iostream)

Exceptions Example

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
    cin.exceptions(cin.failbit); // <--- enable cin error exeption
    int ReturnCode = 0;
    try{
        float Dividend = 0;
        cout << "Dividend: ";
        cin >> Dividend;
        float Divisor = 1;
        cout << "Divisor: ";
        cin >> Divisor;
        float Result = (Dividend/Divisor);
        cout << Result << endl;
    }
}
```

Exceptions Example (2)

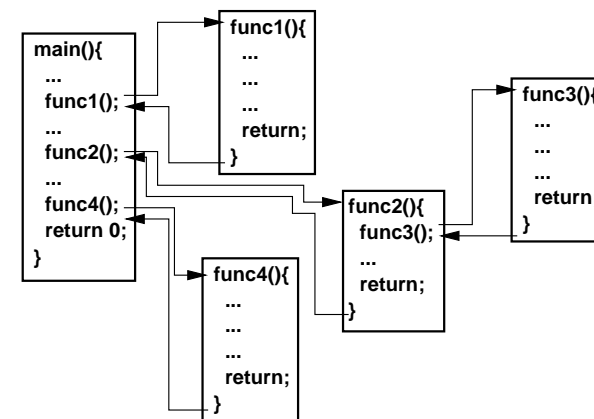
```
catch (...){
    cerr << "Input error, not a number" << endl;
    cin.clear();
    char BadInput[80];
    cin >> BadInput;
    ReturnCode = 1;
}
char StopCharacter;
cout << endl << "Press a key and \"Enter\": ";
cin >> StopCharacter;
return ReturnCode;
}
```

What we have so far

- Variables: data types and names
- Statements: operations on data (e.g., if)
- cout and cin
- error and exception handling
- the main function

Now: functions in detail

Calling Functions



The program **branches** to a called function and **returns** to the statement right after the call.

Designing Functions

- A well-designed function performs a specific task.
- A function does **one thing**, does it well, and returns.
- Complicated tasks should be broken into multiple functions.
- This helps giving your program structure.

Declaring Functions

- Before you can use (call) a function, you have to **declare** it. (“Before” means in the program code.) (Note: This is the same as with variables.)
- Either you write your program “backwards” (main() comes last), or you **declare** your functions, and define (implement) them later in your program.
- If you have a call cycle (a() calls b(), and b() calls a()), then you **must** use declarations.
- Function declarations (**function prototypes**) typically go into header files (→ #include <myfunc.h>)

Declaring and Defining Functions

- Declaration:
return type, name, (parameter list);
Example:

```
int maximum ( int a, int b );
```
- Definition: (→ Implementation)
return type, name, (parameter list) { statements }
Example:

```
int maximum ( int a, int b ){
    if ( a > b ) { return a; }
    else { return b; }
}
```

Return Types

- A function must have a return type.
- Types are either int, float, bool, ... or void, meaning: *no return value*
- A function may have multiple return statements. Beware: this might be difficult to follow!
- In a return statement, every expression (matching the return type) is allowed as return value.

Formal and Actual Parameters

- **Formal** parameters are the parameter names in the declaration/definition.
- **Actual** parameters are the parameters given to an actual call to the function.
- Example:


```
int maximum(int a, int b);
...
x = maximum(42, 44);
```
- The words *parameter* and *argument* are used interchangeably.

Local Variables

- Variables declared inside the function definition (`{ . . . }`) only exist locally within the function. When the function returns, the local variables do not exist any more.
- **Global Variables** are declared outside any function. They are accessible from all functions in the program.
Do NOT use global variables!!!
They are *very* hard to track. . .

Parameters are Local Variables

- In C++, parameters are passed *by value*. This means, a function receives **copies** of the actual parameters.
- With *call-by-value*, variables given as actual parameters are never changed.
- One variable can be simultaneously passed to multiple parameters:


```
y = maximum(x, x)
```

Example: swap()

```
void swap(int x, int y){
    int temp;
    cout << "swap: before swap, x = " << x << " y = " << y << endl;
    temp = x;
    x = y;
    y = temp;
    cout << "swap: after swap, x = " << x << " y = " << y << endl;
}

int main(){
    int x = 5, y = 10;
    cout << "main: before swap(), x = " << x << " y = " << y << endl;
    swap(x,y);
    cout << "main: after swap(), x = " << x << " y = " << y << endl;
    return 0;
}
```

(try it!) (“References” will later fix this problem.)

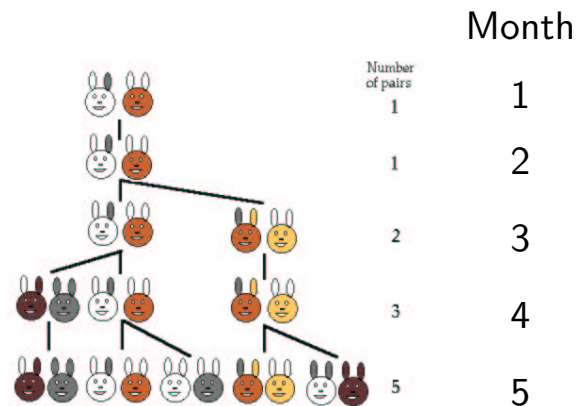
Fibonacci's Rabbits

The Italian mathematician Fibonacci studied in 1202, how fast rabbits could breed under ideal circumstances:

- Rabbits never die.
- A pair of rabbits always produces a male/female pair of offspring.
- A newly born pair of rabbits takes one month to mature.
- Rabbit pregnancy takes one month.



Rabbit Generations



$$\text{Fib}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{Fib}(n - 2) + \text{Fib}(n - 1), & \text{if } n > 1 \end{cases}$$

Recursive Functions

A function is **recursive**, if its definition (implementation) partially contains itself:

$$\text{Fib}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{Fib}(n - 2) + \text{Fib}(n - 1), & \text{if } n > 1 \end{cases}$$

C++ Function Fib()

```
unsigned int Fib(unsigned int n){
    if ( n < 2 ){ // 0 or 1
        return n;
    }
    else {
        unsigned int x, y;
        x = Fib(n-2);
        y = Fib(n-1);
        return x + y;
    }
}
```

Local Variables (dynamically)

- As seen with `Fib()`, each **invocation** of a function gets its own set of local variables!
- (Otherwise, variables would no-longer be local. . .)

C++ function `Fib()` (second try)

```
unsigned int Fib(unsigned int n){
    if ( n < 2 ){ // 0 or 1
        return n;
    }
    else {
        return Fib(n-2) + Fib(n-1);
    }
}
```

Default Parameters (ugly details)

- For each formal parameter, an actual parameter must be given.
- . . . unless the function prototype provides a **default parameter**
Example: `void myFunction (int x = 50);`
Call: `myFunction();`
- If one parameter has a default value, then the rest in the parameter list also must have a default value. (otherwise it is unclear what is left out in the call)

Overloading Functions (more ugly details)

- Sometimes, the same functionality is needed on different types:
`int maximum (int a, int b);`
`double maximum (double a, double b);`
- In C++, multiple functions with the same name are allowed, if they take parameters (at least one) of different types.
- This is called “overloading the function name.”

Overloading Functions (2)

Caution:

- Only use this feature for multiple functions of the **same** functionality!
The “intellectual distance” between two functions with the same name is zero.
- You may have a hard time finding out what will be called, if differences are subtle like signed/unsigned, float/double, . . .

Separate Compilation (Modules)

Using functions to structure your programs.
Extract, e.g., error handling to separate file:

errorhandling.h:

```
#ifndef ErrorHandlingH
#define ErrorHandlingH

namespace ErrorHandling{
    void WaitForPrompt(void);
    int HandleNotANumberError(void);
}

#endif
```

errorhandling.cpp

```
#include <iostream>
#include "errorhandling.h"
namespace ErrorHandling{
    using namespace std;
    void WaitForPrompt(void){
        char StopCharacter;
        cout << endl << "Press a key and \"Enter\": ";
        cin >> StopCharacter;
    }
    int HandleNotANumberError(void){
        cerr << "Input error, not a number" << endl;
        cin.clear();
        char BadInput[80];
        cin >> BadInput;
        return 1;
    }
}
```

Using errorhandling.cpp

```
#include <iostream>
#include "errorhandling.h"
using namespace std;
int main(int argc, char *argv[]){
    cin.exceptions(cin.failbit);
    int ReturnCode = 0;
    try{
        float Dividend = 0;
        cout << "Dividend: ";
        cin >> Dividend;
        float Divisor = 1;
        cout << "Divisor: ";
        cin >> Divisor;
        float Result = (Dividend/Divisor);
        cout << Result << endl;
    }
}
```

Using errorhandling.cpp (2)

```
catch (...){  
    ReturnCode = ErrorHandling::HandleNotANumberError();  
}  
ErrorHandling::WaitForPrompt();  
return ReturnCode;  
}
```

Next Lesson:

- The switch statement (a “multi-if”)
- Loops
- Arrays
- `assert()`

Introduction to Programming, Lesson 4

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

Programming Assignments

You can get your code cards for the computer lab:

- Tuesday, May 20, 14:00–15:00
- Office H-B 5404 (Mrs. Tröps)
- Please bring:
 - ★ your student id
 - ★ 10,00 EUR (deposit)

Reminder: Exception Handling

```
try{
    // normal code
}
catch (...){
    // error handling
}
```

- modern libraries “throw” exception instead of using return codes
- details need to be taken from library documentation (e.g., for iostream)

Reminder: Separate Compilation

Using functions to structure your programs.

Extract, e.g., error handling to separate file:

errorhandling.h:

```
#ifndef ErrorHandlingH
#define ErrorHandlingH

namespace ErrorHandling{
    void WaitForPrompt(void);
    int HandleNotANumberError(void);
}

#endif
```

Reminder: Separate Compilation (2)

```
#include <iostream>
#include "errorhandling.h"
namespace ErrorHandler{
    using namespace std;
    void WaitForPrompt(void){
        char StopCharacter;
        cout << endl << "Press a key and \"Enter\": ";
        cin >> StopCharacter;
    }
    int HandleNotANumberError(void){
        cerr << "Input error, not a number" << endl;
        cin.clear();
        char BadInput[80];
        cin >> BadInput;
        return 1;
    }
}
```

Reminder: Functions

```
int main(){
    int x = 5, y = 10;
    swap(x,y);
    return 0;
}

void swap(int x, int y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Reminder: Recursive Functions

```
unsigned int Fib(unsigned int n){
    if ( n < 2 ){ // 0 or 1
        return n;
    }
    else {
        return Fib(n-2) + Fib(n-1);
    }
}
```

Today:

- The switch statement (a “multi-if”)
- More exceptions (the calculator extended)
- Loops
- Arrays
- assert()

Multiple if Statements

```
char command;
cin >> command;
if ( command == '+' ){
    add();
}
if ( command == '-' ){
    subtract();
}
if ( command == '*' ){
    multiply();
}
if ( command == '/' ){
    divide();
}
else {
    cout << "unknown command\n";
}
}
```

The switch Statement

```
char command;    cin >> command;
switch ( command ){
    case '+':
        add();
        break;
    case '-':
        subtract();
        break;
    case '*':
        multiply();
        break;
    case '/':
        divide();
        break;
    default:
        cout << "unknown command\n";
}
}
```

switch Statements

```
switch ( expression ){ // any simple C++ expression
    case constant1: // test for equality
        statements
        break; // leave the switch statement
                // (optional)

    case constant2:
    case constant3:
        statements
        break;

    default: // handle everything else
        statements // (optional)
}
}
```

What is the output?

```
int number;    cin >> number;
switch ( number ){
    case 0:
        cout << "zero\n";
    case 1:
        cout << "one\n";
    case 2:
        cout << "two\n";
    case 3:
        cout << "three\n";
    default:
        cout << "not between zero and three\n";
}
}
```

The Style of switch

- Use break (almost) always.
- If you do not use break, add a comment to make sure that you really mean it:

```
case 1:
    do_something(); // fall through
case 2:
    do_something_else(); // fall through
case 3:
    do_what_is_needed_for_1_2_and_3();
```
- Always use a default clause to handle unforeseen cases. This helps debugging later on!

The Calculator Extended

```
int main(int argc, char *argv[]){
    ErrorHandling::Initialize();
    do{
        try{
            char Operator = GetOperator();
            float Operand = GetOperand();
            cout << Accumulate(Operator,Operand) << endl;
        }
        catch (runtime_error RuntimeError){
            ErrorHandling::HandleRuntimeError(RuntimeError);
        }
        catch (...){ // catch all other exceptions
            ErrorHandling::HandleNotANumberError();
        }
    } while (ErrorHandling::Continue("More? "));
    return 0;
}
```

Exception Handling (extended)

```
try{
    // normal code
}
catch (class exception_object){
    // catch exception of 'class'
    // 'exception_object' has the details
}

// possibly more catch statements

catch (...){
    // catch all other exceptions
    // like the 'default' clause of 'switch'
}
```

Throwing an Exception

```
float Accumulate(const char theOperator, const float theOperand){
    static float myAccumulator = 0;
    switch (theOperator){
        case '+':
            myAccumulator += theOperand;    break;
        case '-':
            myAccumulator -= theOperand;    break;
        case '*':
            myAccumulator *= theOperand;    break;
        case '/':
            myAccumulator /= theOperand;    break;
        default:
            throw runtime_error("Error: Invalid operator");
    }
    return myAccumulator;
}
```

errorhandling.h:

```
#ifndef ErrorHandlingH
#define ErrorHandlingH

#include <stdexcept>      --- or: #include <exception>
namespace ErrorHandling{
    using namespace std;
    int HandleRuntimeError(runtime_error theError);
}
#endif
```

errorhandling.cpp:

```
int HandleRuntimeError(runtime_error theError){
    cerr << theError.what() << endl;
    return 1;
}
```

Loops

- **Iteration:** repeatedly acting on the same data
- In C++, this is achieved with loops.
 - ★ while loops
 - ★ do while loops
 - ★ for loops

while Loops

- Example:

```
cout << "How many hello's?\n";
cin >> number;
```

```
while ( number > 0 ){
    cout << "Hello!\n";
    number--;
}
```

- Expression is evaluated **before** the loop body is entered.
- The body may thus not be entered at all.

do while Loops

- Example:

```
do {
    char c;
    cout << "Do it again (y/n)?\n";
    cin >> c;
} while ( c != 'n' );
```

- Expression is evaluated **after** the loop body is entered.
- The body will always be entered. (at least once)

The Style of `while`

- Statements inside the loop body:
 - ★ `break`;
Exit the loop body without testing the loop expression.
 - ★ `continue`;
Jump back to the top of the loop body.
- Do **not** (hardly) use `break`; or `continue`;!
It is easier to understand if you do the same with `if / else` or exceptions.

Correctness of `while` and `do while`

- The loop **expression** must eventually become false (0).
If not, the loop runs forever.
- This may be non-trivial (e.g., with **if/else** statements)
- Typically, a loop has some initialization, a condition, and a “step” statement (like increment or decrement).

for Loops

- for loops prescribe initialization, loop condition, and step.

- Example:

```
for ( i = 0; i < 10; i++ ) {
    cout << "i = " << i << endl;
}
```

for Loops (2)

```
for ( statement ; expression ; statement )
    statement
```

- Statements may be empty (just the `;`)
- Statements may be complex (bad style):

```
for ( i = 0; i < 10; cout << "i = " << i++ << endl ) ;
```

Nested Loops

Loops may be inside loops:

```
int i,j;
for ( i = 0; i < 5; i++ ){
    for ( j = 0; j < 20; j++ ){
        cout << '.';
    }
    cout << endl;
}
```

Style: Always use { and } for the loop body!

Iteration by Tail Recursion

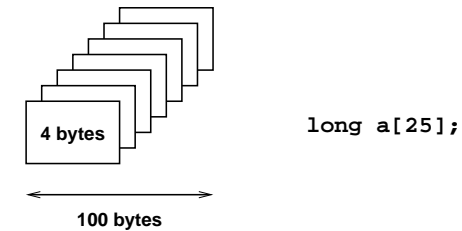
- Example:

```
unsigned int faculty (unsigned int n){
    if ( n < 2 ){
        return 1;
    }
    else{
        return n * faculty(n-1);
    }
}
```

- Avoid tail recursion!
Use loops instead!

What are Arrays?

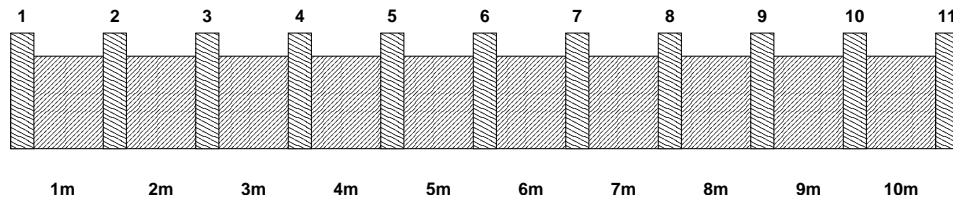
- An array is a collection of data elements, each of the same *type*.
- The number of elements is the *subscript* or *size* of the array.



Array Elements

- Array elements are accessible via their **index**:
 $a[4] = 42;$
 $a[i] = a[i+1];$
- For an array of size n , elements have index $[0 \dots n - 1]$
- Example: `char c[3]` has elements:
`c[0]`, `c[1]`, and `c[2]`

Fence-Post Errors



- For a 10 m fence, you need 11 posts.
- For accessing a[10], you have to declare the array with 11 elements.

Reading/Writing Behind the Array Boundary

- The compiler does **not** check if array indices are correct.
- It is possible to read and write non-existing array elements!
These are other memory locations of your program!
Possibly both variables or program code!
- In the best case, your program crashes immediately.
But you may also get strange effects later on. . .

Initializing Arrays

```
• int intArray[5] = { 1, 3, 4, -2, 7 };
```

```
• int intArray[] = { 1, 3, 4, -2, 7 };
```

Compute the actual size:

```
const unsigned int ArrayLength =
sizeof(intArray)/sizeof(intArray[0]);
```

Beware: this size computation only works when the compiler has sufficient information about the array!

Multidimensional Arrays

```
#include <iostream.h>
int main(){
    short c[2][3] = { { 0 , 1 , 2 } ,
                     { 3 , 4 , 5 } };
    for (int i = 0; i < 2; i++){
        for (int j = 0; j < 3; j++){
            cout << c[i][j] << ' ';
        }    cout << endl;
    }
    cout << sizeof(c) << endl;
    cout << sizeof(c[0]) << endl;
    cout << sizeof(c[0][0]) << endl;
    return 0; }
```

Arrays as Function Parameters

```
void sort(int a[], unsigned int size){
    int i,j;
    for ( i = 0; i < size; i++ ) {
        for ( j = i+1; j < size; j++ ) {
            if ( a[i] > a[j] ){
                int temp;
                temp = a[i];    a[i] = a[j];    a[j] = temp;
            }
        }
    }
}
```

Is this a useful function?

Arrays as Function Parameters (2)

- In C++, array parameters are passed *by reference*. This means, a function receives the actual array parameters (**no copies**)!
- With *call-by-reference*, variables given as actual parameters may be changed.
- (Arrays are implicitly using **references**, which will be explained in later lessons. . .)
- As formal parameters, arrays are declared with unspecified size, for example a[].

assert(), Debugging Aid

- Use `#include <assert.h>`
- Then use `assert(expression);` to include runtime checks of expressions you believe to be true.
- Example: `assert(i < array_size);`
- When your program reaches an `assert()` statement, the expression is checked. When the expression is false (0), then your program halts with an error message.
- **Beware:** do not use side effects, like:
`assert (i++ < X);!`

Character Arrays

- `char Greeting[] = {'H','e','l','l','o','\0'};`
Note the `'\0'` at the end!
- Same as: `char Greeting[] = "Hello";`
- C++ provides **String classes** in the standard library (not in the language).
(Then, strings are objects; later in the course. . .)

Input to Character Arrays

- “Naive” approach:

```
char buffer[80];
cin >> buffer;
```
- Problems:
 - ✦ Reading stops at first 'blank' character.
 - ✦ Possible buffer overrun.

Next Lesson:

- Types
- Structs
- Objects and classes
- File I/O

Input to Character Arrays (2)

- Correct approach:

```
char buffer[80];
cin.get(buffer,79);
```
- Note the **79**! The last array element is for the '**\0**'
- Problems solved:
 - ✦ Reads until the end of the line.
 - ✦ Does not overrun the buffer array.
- Note: `cin` is an object, and `get` is a method.
 (Later. . .)

Introduction to Programming, Lesson 5

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

Reminder: switch Statements

```
switch ( expression ){ // any correct C++ expression
  case constant1: // test for equality
    statements
    break; // leave the switch statement
             // (optional)

  case constant2:
  case constant3:
    statements
    break;

  default: // handle everything else
    statements // (optional)
}
```

Reminder: Loops

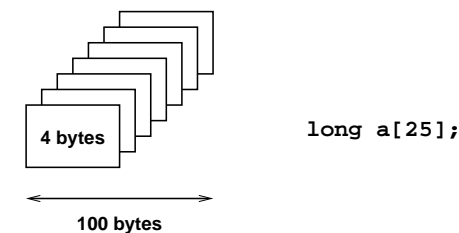
```
while ( expression )
  statement
```

```
do statement
while ( expression )
```

```
for ( statement ; expression ; statement )
  statement
```

Reminder: Arrays

- An array is a collection of data elements, each of the same *type*.
- Array elements are accessible via their **index**:
 $a[i] = a[i+1]$;



Reminder: `assert()`

- Use `#include <assert.h>`
- Example: `assert(i < array_size);`
- When your program reaches an `assert()` statement, the expression is checked. When the expression is false (0), then your program halts with an error message.

Today:

- Objects and classes
- Simple string objects
- Simple file I/O

Variables and Types

- Types (like **int**, **float**, **bool**) tell the compiler both the size of the variables (e.g., 4, 8, 1 bytes), and the possible operations on them.
- Example:
If **height** and **width** are variables of type **int**, then the compiler knows that for each of them 4 bytes need to be reserved, and that **height * width** is a legal operation.

Built-In Types

- The types in C++ like **int**, **float**, **bool** are called **elementary**, **simple**, or **primitive** types.
- Real programs need to solve real-world problems: engineering computations, book keeping, airline reservation. . .
- Primitive types can hardly represent the data of real applications.
- **Arrays** are only a very weak form of aggregating primitive types to more useful units.

(Reminder:) Function Return Types

- A function can return at most one value.
- A function must have a return type.
- Types are either `int`, `float`, `bool`, ... or `void`, meaning: *no return value*
- Types may also be self defined: **enum**, **struct**, **class**

Structure Types (C heritage)

```
struct complex { double re, im; };
```

Using a complex variable:

```
complex c;
c.re = 1.0;
c.im = 0.0;
```

- a `struct` variable forms a unit
- but the internals are open, no restriction to “useful” operations

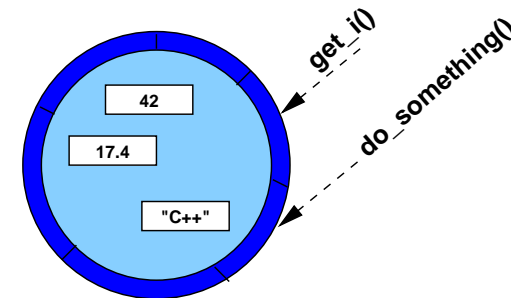
Classes: Creating new Types

A **class** defines a new type:

- a collection of variables (**data members**)
- a set of related functions (**methods, member functions**)
They define the **interface** to objects of the class.

An **object** is an instance (a variable) of a class.

Encapsulation



The bundling together of all the information, capabilities, and responsibilities into a single object.

Declaring a Class

```
class Cat{
public:
    unsigned int itsAge;
    unsigned long itsWeight;
    void Meow();
};
```

Defining an object:

```
unsigned long GrossWeight; // define an unsigned long
Cat Frisky;                // define a Cat
```

Accessing Class Members

Use the . operator:

- **Frisky.itsWeight = 50;**
- **Frisky.Meow();**

Private vs. Public

Class members can be marked as private or as public:

- **private** members can be accessed only within methods of the class itself.
- **public** members can also be accessed from other classes (“code outside the class itself”)
- The keywords **public:** and **private:** indicate: “from here on, members are public/private”
- The default is “private”.

Cat Class (2)

```
class Cat{
    unsigned int itsAge;
    unsigned long itsWeight;
    void Meow();
};

Cat Boots; // define a Cat
Boots.itsAge = 5; // Error! cannot access private data!
Boots.Meow(); // Error! cannot call private method!
```

Make Data Members Private!

```
class Cat{
public:
    void Meow();
    void SetAge(unsigned int age);
    unsigned int GetAge();
    void SetWeight(unsigned long weight);
    unsigned long Getweight();
private:
    unsigned int itsAge;
    unsigned long itsWeight;
};
```

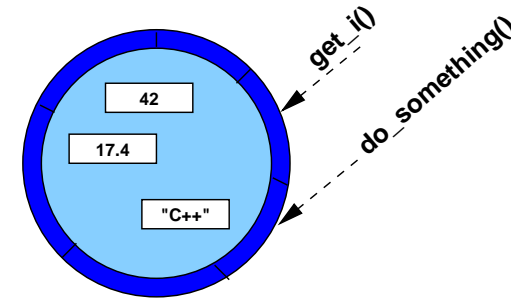
Structure Types Revisited

```
struct complex { double re, im; };
```

A struct is like a class

- with all members public
- without well-defined operations

Accessor Methods



Accessor Methods (like GetAge and SetWeight) encapsulate object data.

A class interface describes the “**what**”, not the “**how**”.

Implementing Class Methods

Use the :: operator:

```
void Cat::Meow(){
    cout << "Meow.\n";
}
unsigned int Cat::GetAge(){
    return itsAge;
}
void Cat::SetWeight(unsigned long weight){
    itsWeight = weight;
}
```

Quiz

Can one **Cat** access another **Cat's itsAge** member?

E.g., is the following correct?

```
void Cat::Quiz(){
    Cat Kitten;
    Kitten.itsAge = 0;
}
```

(itsAge is a private member of class Cat.)

Constructors and Destructors

Variables can be declared and later initialized:

```
int i;
...
i = 42;
```

or they can be initialized within the declaration:

```
int i = 42;
```

Now, the variable **i** **always** has a well-defined value.

A **constructor** does the same for objects.

Constructors

A constructor is a method with the same name as the class:

```
Cat::Cat(unsigned int age){
    itsAge = age;
}
```

A constructor has no return value; not even “void”.

Use as in: `Cat Frisky(5);`

Multiple Constructors

A class can have multiple constructors, if they can be distinguished by their signature (the parameter types).

This is the same as with function overloading.

Beware: remember “intellectual distance”:

```
Cat::Cat(unsigned int age)
Cat::Cat(unsigned long weight)
```

→ this is asking for trouble!

Better use:

```
Cat::Cat(unsigned int age)
Cat::Cat(unsigned int age, unsigned long weight)
```

The Default Constructor

The parameterless constructor is called **default** constructor:

```
Cat::Cat()
```

For each class without declared constructors, the compiler automatically generates a default constructor:

```
Cat::Cat() {};
```

Destructors

A desctructor is called whenever an object gets destroyed.

Example with an object as local variable:

```
void my_function(){
    Cat Frisky(5);    // constructor is called

    ...

    // here, the destructor is called
}
```

Destructors (2)

A destructor is a method with the same name as the class, preceded by a `~`:

```
Cat::~~Cat(){
    // do something useful
}
```

A destructor has no return value; not even “void”.
A destructor does not take parameters.

If not implemented, the compiler generates an empty (default) destructor.

What should a Destructor do?

All kinds of cleanup:

- free memory (next week)
- close files (streams)
- do application-specific cleanup (e.g., statistics)

Programming Style:

If you implement constructors, also implement a destructor!

Even if it is empty; document you mean it like that!
(Same as with a missing **break** in a **switch** statement.)

const Member Functions

```
class Cat{
public:
    void Meow();                // does not change object
    void SetAge(unsigned int age);
    unsigned int GetAge();      // does not change object
    void SetWeight(unsigned long weight);
    unsigned long Getweight(); // does not change object
private:
    unsigned int itsAge;
    unsigned long itsWeight;
};
```

const Member Functions (2)

```
class Cat{
public:
    void Meow() const;
    void SetAge(unsigned int age);
    unsigned int GetAge() const;
    void SetWeight(unsigned long weight);
    unsigned long Getweight() const;
private:
    unsigned int itsAge;
    unsigned long itsWeight;
};
```

1. you document that a method just reads
2. and the compiler enforces this for you!

Use const Functions Whenever Possible!

- You add information to the class interface. For others (or yourself) to use it properly.
- The compiler gives you error messages, if you still try to modify the object.
- Compile-time errors are better than runtime errors! (With testing, you may miss the case in which it happens. . .)

Class Interface vs. Implementation

- The class interface defines a “contract” with clients of the class (other classes).
- The interface should be put in a separate header file (like **Cat.h**)
- Other classes might then do: `#include "Cat.h"`
- The data members in the class declaration might be seen as a design problem of C++.
(They are needed by the compiler for computing the size of objects.)

Cat.h: Class Interface

```
class Cat{
public:
    void Meow() const;
    void SetAge(unsigned int age);
    unsigned int GetAge() const;
    void SetWeight(unsigned long weight);
    unsigned long Getweight() const;
private:
    unsigned int itsAge;
    unsigned long itsWeight;
};
```

Cat.cpp: Class Implementation

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

void Cat::Meow(){
    cout << "Meow.\n";
}

unsigned int Cat::GetAge(){
    return itsAge;
}

void Cat::SetWeight(unsigned long weight){
    itsWeight = weight;
}
```

main.cpp: Main Program

```
#include "Cat.h"           // in the same directory
#include <iostream.h>      // in system directories

int main(){
    Cat Frisky;
    Frisky.SetAge(5);
    Frisky.Meow();
    cout << "Frisky is a cat who is ";
    cout << Frisky.GetAge() << " years old.\n";
    Frisky.Meow();
    return 0;
}
```

The makefile

```
Cat: main.o Cat.o
    g++ -o Cat main.o Cat.o

Cat.o: Cat.cpp Cat.h

main.o: main.cpp Cat.h

(make "knows" how to compile .cpp to .o)
```

Simple string Objects

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

int main() {
    string s1, s2; // Empty strings
    string s3 = "Hello, World."; // Initialized
    string s4("I am"); // Also initialized
    s2 = "Today"; // Assigning to a string
    s1 = s3 + " " + s4; // Combining strings
    s1 += " 8 "; // Appending to a string
    cout << s1 + s2 + "!" << endl;
    return 0;
}
```

Simple File I/O

```
#include <string>
#include <fstream>
using namespace std;
int main() {
    ifstream in("filecopy.cpp"); // Open for reading
    ofstream out("filecopy2.cpp"); // Open for writing
    string s;
    while (getline(in, s)){
        out << s << endl ; // ... must add back the endl
    }
    return 0;
} // destructor closes the files
```

Simple File I/O (2)

```
ifstream in;
ofstream out;
stream s;
try {
    in.exceptions(in.failbit);
    out.exceptions(out.failbit);
    in.open("filecopy.cpp",ios_base::in);
    out.open("filecopy2.cpp",ios_base::out);
    while (getline(in,s)){
        out << s << endl;
    }
    in.close();
    out.close();
} catch ( ios_base::failure IOError ){
    cerr << IOError.what() << endl;
}
```

Next Lesson:

- Pointers
- More about memory
- Garbage collection

Introduction to Programming, Lesson 6

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

Reminder: Classes and Objects

A **class** defines a new type:

- a collection of variables (**data members**)
- a set of related functions (**methods, member functions**)
They define the **interface** to objects of the class.

An **object** is an instance (a variable) of a class.

Reminder: Constructors

A constructor is a method with the same name as the class:

```
Cat::Cat(unsigned int age){
    itsAge = age;
}
```

A constructor has no return value; not even “void”.

Use as in: `Cat Frisky(5);`

Reminder: Destructors

A destructor is a method with the same name as the class, preceded by a `~`:

```
Cat::~~Cat(){
    // do something useful
}
```

A destructor has no return value; not even “void”.

A destructor does not take parameters.

Reminder: Class Interface vs. Implementation

- The class interface defines a “contract” with clients of the class (other classes).
- The interface should be put in a separate header file (like **Cat.h**)
- Other classes might then do: `#include "Cat.h"`

Reminder: Cat.h: Class Interface

```
class Cat{
public:
    void Meow() const;
    void SetAge(unsigned int age);
    unsigned int GetAge() const;
    void SetWeight(unsigned long weight);
    unsigned long Getweight() const;
private:
    unsigned int itsAge;
    unsigned long itsWeight;
};
```

Reminder: Cat.cpp: Class Implementation

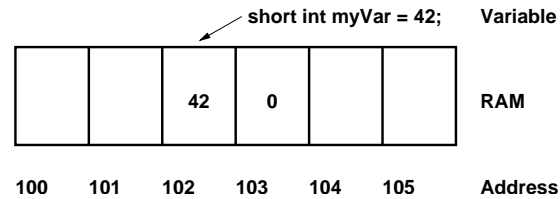
```
#include "Cat.h"
#include <iostream.h>

void Cat::Meow(){
    cout << "Meow.\n";
}
unsigned int Cat::GetAge(){
    return itsAge;
}
void Cat::SetWeight(unsigned long weight){
    itsWeight = weight;
}
```

Today:

- Pointers
- More about memory
- Garbage collection
- Pointers and arrays
- References

A Variable in Memory



Each memory cell has the size of one byte.
Variables give **name** and **type** to memory locations.
Here:

- myVar is located at address 102.
- myVar has a size of 2 bytes. (short int)

Pointers

A variable stores a value according to its type.
A pointer is a variable that stores a memory address.

```
int myAge = 25; // an int
int *pAge;    // a pointer to int
pAge = &myAge; // pAge now points to myAge
*pAge = 37;
```

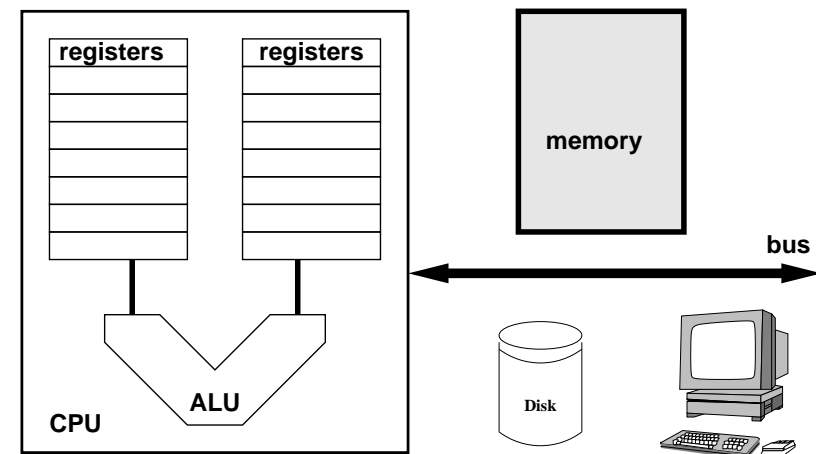
Pointers are typed! (e.g., int*, double*, ...)
Accessing ***pAge** is called “dereferencing” **pAge**.

Pointers are Dangerous!

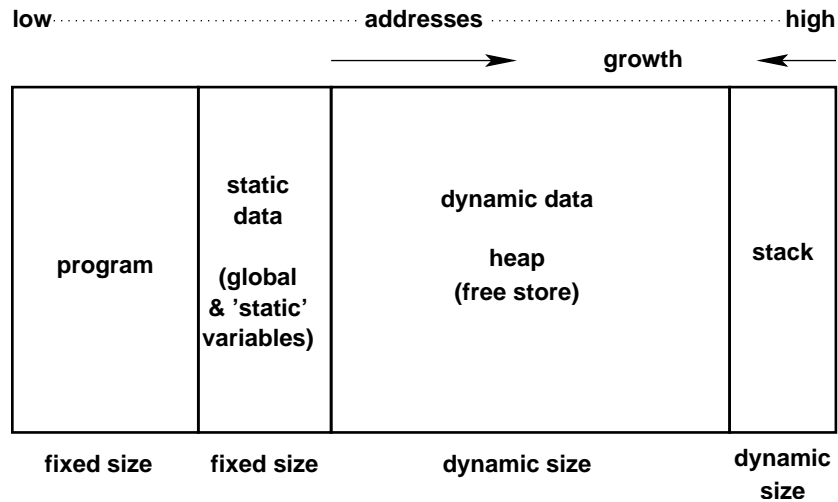
- Via pointers, you can read/write **any** memory location.
- This means, you can overwrite other variables, or the program itself. . .
- As with array bounds violations, your program may crash immediately, or one hour later. . .

Play safe: always initialize your pointers!

Reminder: The Computer from Inside



Segments in Memory



Static Data

- All global variables (outside any function) live in the static data memory segment.
- The same is true for variables declared as `static`
- Their size has to be known at compile time (e.g., for arrays).

A Stack (pseudo code)

```
class Stack{
public:
    void Push(void *data); // put data on top of the stack
    void *Top() const;    // return the data from the top
                        // of the stack
    void Pop();           // remove the topmost data item
}
```

Data on the Stack

- local variables
- function parameters
- (return address for functions)

Data on the stack has limited lifetime.
The stack might be limited in size.

Example: Fibonacci

```

unsigned int Fib(unsigned int n){
    if ( n < 2 ){ // 0 or 1
        return n;
    }
    else {
        return Fib(n-2) + Fib(n-1);
    }
}
int main(){
    unsigned int N = 2;
    cout << fib(N) << endl;
    return 0;
}

```

Data on the Heap

- The *heap* is a large memory segment where we can allocate parts of it dynamically (at runtime). We do not need to know in advance how much data will be needed.
- The heap can grow until the memory of the computer is exceeded.
- Memory on the heap can (should!) also be freed (un-allocated) for later re-use.

new and delete

```

int *pointer;

pointer = new int; // allocate variable on the heap

*pointer = 42;
    // write the value 42 into the heap variable

delete pointer; // free memory on the heap

```

Forgetting delete: memory leaks!

```

void evil(){
    int *pointer = new int;
}

```

The `int` on the heap “leaked out of the program,” because it is no longer reachable. (the only pointer is gone) . . . and can not be freed any more. This memory is lost until the program ends.

delete only once!

- delete-ing a variable that already has been deleted, immediately crashes your program!
(the memory management data gets corrupted)
- Assign 0 to a deleted pointer!
delete pointer; pointer = 0;
It is defined to be safe deleting a 0-pointer.
- A non-initialized or deleted-but-not-zeroed pointer is called “dangling pointer”.
(A “time bomb”, exploding when dereferenced.)

Excursion: Garbage Collection

- Modern object-oriented languages (like Java) do the delete-business for you!
- The number of references (variables) to a heap object is counted. (reference counting)
- When the number of references becomes zero, the object is marked for deletion.
- Once in a while, a *garbage collector* frees all deleted objects.
- This requires careful language design
(e.g., pointer assignment)

Objects on the Heap

```
void ShortLivedCat(){
    Cat *pCat = new Cat(5); // creates Cat with age 5
                          // new calls a constructor

    delete pCat;          // delete calls the destructor
}
```

Accessing Members on Heap Objects

```
void SlightlyLongerLivedCat(){
    Cat *pCat = new Cat(5); // new calls a constructor

    (*pCat).SetWeight(7); // . operator applied to the
                          // Cat object *pCat

    pCat -> SetWeight(7); // -> does the same, is simpler

    long weight = pCat->itsWeight;

    delete pCat;          // delete calls the destructor
}
```

Reminder: Arrays

- Array elements are accessible via their **index**:
`a[4] = 42;`
`a[i] = a[i+1];`
- For an array of size n , elements have index $[0 \dots n - 1]$
- Example: `char c[3]` has elements:
`c[0]`, `c[1]`, and `c[2]`

Arrays of Objects

- Any object can be stored in an array:
`Cat myPets[42];`
- The class (here: **Cat**) must have a default constructor (without parameters) for creating the objects along with the array.
- Accessing members:
`myPets[17].meow();`

Arrays of Pointers

```
void my_function(){
    Cat *Family[500];
    int i;
    for ( i=0; i<500; i++ ){
        Family[i] = new Cat;
        Family[i]->SetAge(2*i+1);
    }
    // delete the objects one by one...
}
```

The array is on the stack.

The objects are on the heap.

Arrays on the Heap

```
void my_function(){
    Cat *Family = new Cat[500];
    int i;
    for ( i=0; i<500; i++ ){
        Family[i].SetAge(2*i+1);
    }
    delete Family; // what are we deleting here??
}
```

The array with all the objects is on the heap.

Deleting Arrays on the Heap

```
void my_function(){
    Cat *Family = new Cat[500];
    int i;
    for ( i=0; i<500; i++ ){
        Family[i].SetAge(2*i+1);
    }
    delete [] Family;
}
```

The operator **delete []** deletes an array!
and calls the destructor on all objects,
e.g. further cleanup. . .

A Pointer to an Array vs. An Array of Pointers

In the following, which is which?

```
Cat FamilyOne[500];
Cat *FamilyTwo[500];
Cat *FamilyThree = new Cat[500];
```

For each three, give the syntax to call SetAge(10) on the Cat with index 10!

Pointers and Array Names

```
Cat FamilyOne[500];
Cat *FamilyTwo[500];
Cat *FamilyThree = new Cat[500];

FamilyOne[10].SetAge(10);
FamilyTwo[10]->SetAge(10);
FamilyThree[10].SetAge(10);
```

In C++, an array name is a const pointer to the first element of the array.

References

A **reference** is an alias (a second name) for an object.

A **reference** is **not** a pointer.

(But it can achieve similar goals with cleaner syntax.)

```
int &someRef = someInt;
```

A **reference** can only be initialized, but not re-assigned.

Assigning to a Reference

```
int intOne;
int &SomeRef = intOne;

intOne = 5;
cout << intOne << " " << SomeRef << endl;

int intTwo = 8;
SomeRef = intTwo; // what is this ???
cout << intOne << " " << SomeRef << " " << intTwo << endl;
```

What is the output?

Reminder: swap()

```
void swap(int x, int y){
    int temp;
    cout << "swap: before swap, x = " << x << " y = " << y << endl;
    temp = x;
    x = y;
    y = temp;
    cout << "swap: after swap, x = " << x << " y = " << y << endl;
}

int main(){
    int x = 5, y = 10;
    cout << "main: before swap(), x = " << x << " y = " << y << endl;
    swap(x,y);
    cout << "main: after swap(), x = " << x << " y = " << y << endl;
    return 0;
}
```

Making swap() work with Pointers

```
void swap(int *x, int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int x = 5, y = 10;
    cout << "main: before swap(), x = " << x << " y = " << y << endl;
    swap(&x,&y);
    cout << "main: after swap(), x = " << x << " y = " << y << endl;
    return 0;
}
```

swap() now works, but the syntax is spoiled. . .

swap() with References!

```
void swap(int &x, int &y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main(){
    int x = 5, y = 10;
    cout << "main: before swap(), x = " << x << " y = " << y << endl;
    swap(x,y);
    cout << "main: after swap(), x = " << x << " y = " << y << endl;
    return 0;
}
```

Et voilà!

Next Lesson

- More about objects, pointers, references, . . .
- **June 16!**

Introduction to Programming, Lesson 6

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

Reminder: Classes and Objects

A **class** defines a new type:

- a collection of variables (**data members**)
- a set of related functions (**methods, member functions**)
They define the **interface** to objects of the class.

An **object** is an instance (a variable) of a class.

Reminder: Constructors

A constructor is a method with the same name as the class:

```
Cat::Cat(unsigned int age){
    itsAge = age;
}
```

A constructor has no return value; not even “void”.

Use as in: `Cat Frisky(5);`

Reminder: Destructors

A destructor is a method with the same name as the class, preceded by a `~`:

```
Cat::~~Cat(){
    // do something useful
}
```

A destructor has no return value; not even “void”.

A destructor does not take parameters.

Reminder: Class Interface vs. Implementation

- The class interface defines a “contract” with clients of the class (other classes).
- The interface should be put in a separate header file (like **Cat.h**)
- Other classes might then do: `#include "Cat.h"`

Reminder: Cat.h: Class Interface

```
class Cat{
public:
    void Meow() const;
    void SetAge(unsigned int age);
    unsigned int GetAge() const;
    void SetWeight(unsigned long weight);
    unsigned long Getweight() const;
private:
    unsigned int itsAge;
    unsigned long itsWeight;
};
```

Reminder: Cat.cpp: Class Implementation

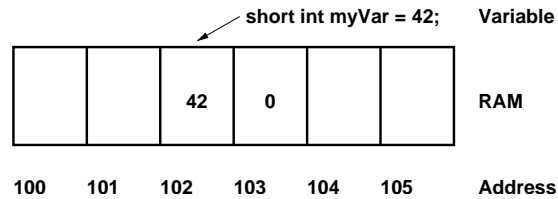
```
#include "Cat.h"
#include <iostream.h>

void Cat::Meow(){
    cout << "Meow.\n";
}
unsigned int Cat::GetAge(){
    return itsAge;
}
void Cat::SetWeight(unsigned long weight){
    itsWeight = weight;
}
```

Today:

- Pointers
- More about memory
- Garbage collection
- Pointers and arrays
- References

A Variable in Memory



Each memory cell has the size of one byte.
Variables give **name** and **type** to memory locations.
Here:

- myVar is located at address 102.
- myVar has a size of 2 bytes. (short int)

Pointers

A variable stores a value according to its type.
A pointer is a variable that stores a memory address.

```
int myAge = 25; // an int
int *pAge;     // a pointer to int
pAge = &myAge; // pAge now points to myAge
*pAge = 37;
```

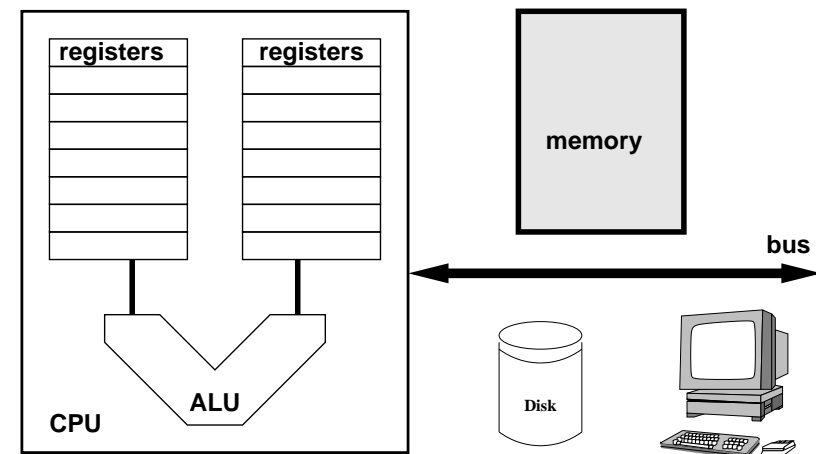
Pointers are typed! (e.g., int*, double*, ...)
Accessing ***pAge** is called “dereferencing” **pAge**.

Pointers are Dangerous!

- Via pointers, you can read/write **any** memory location.
- This means, you can overwrite other variables, or the program itself. . .
- As with array bounds violations, your program may crash immediately, or one hour later. . .

Play safe: always initialize your pointers!

Reminder: The Computer from Inside



Example: Fibonacci

```

unsigned int Fib(unsigned int n){
    if ( n < 2 ){ // 0 or 1
        return n;
    }
    else {
        return Fib(n-2) + Fib(n-1);
    }
}
int main(){
    unsigned int N = 2;
    cout << fib(N) << endl;
    return 0;
}

```

Data on the Heap

- The *heap* is a large memory segment where we can allocate parts of it dynamically (at runtime). We do not need to know in advance how much data will be needed.
- The heap can grow until the memory of the computer is exceeded.
- Memory on the heap can (should!) also be freed (un-allocated) for later re-use.

new and delete

```

int *pointer;

pointer = new int; // allocate variable on the heap

*pointer = 42;
    // write the value 42 into the heap variable

delete pointer; // free memory on the heap

```

Forgetting delete: memory leaks!

```

void evil(){
    int *pointer = new int;
}

```

The `int` on the heap “leaked out of the program,” because it is no longer reachable. (the only pointer is gone) . . . and can not be freed any more. This memory is lost until the program ends.

delete only once!

- delete-ing a variable that already has been deleted, immediately crashes your program!
(the memory management data gets corrupted)
- Assign 0 to a deleted pointer!
delete pointer; pointer = 0;
It is defined to be safe deleting a 0-pointer.
- A non-initialized or deleted-but-not-zeroed pointer is called “dangling pointer”.
(A “time bomb”, exploding when dereferenced.)

Excursion: Garbage Collection

- Modern object-oriented languages (like Java) do the delete-business for you!
- The number of references (variables) to a heap object is counted. (reference counting)
- When the number of references becomes zero, the object is marked for deletion.
- Once in a while, a *garbage collector* frees all deleted objects.
- This requires careful language design
(e.g., pointer assignment)

Objects on the Heap

```
void ShortLivedCat(){
    Cat *pCat = new Cat(5); // creates Cat with age 5
                          // new calls a constructor

    delete pCat;          // delete calls the destructor
}
```

Accessing Members on Heap Objects

```
void SlightlyLongerLivedCat(){
    Cat *pCat = new Cat(5); // new calls a constructor

    (*pCat).SetWeight(7); // . operator applied to the
                          // Cat object *pCat

    pCat -> SetWeight(7); // -> does the same, is simpler

    long weight = pCat->itsWeight;

    delete pCat;          // delete calls the destructor
}
```

Reminder: Arrays

- Array elements are accessible via their **index**:
`a[4] = 42;`
`a[i] = a[i+1];`
- For an array of size n , elements have index $[0 \dots n - 1]$
- Example: `char c[3]` has elements:
`c[0]`, `c[1]`, and `c[2]`

Arrays of Objects

- Any object can be stored in an array:
`Cat myPets[42];`
- The class (here: **`Cat`**) must have a default constructor (without parameters) for creating the objects along with the array.
- Accessing members:
`myPets[17].meow();`

Arrays of Pointers

```
void my_function(){
    Cat *Family[500];
    int i;
    for ( i=0; i<500; i++ ){
        Family[i] = new Cat;
        Family[i]->SetAge(2*i+1);
    }
    // delete the objects one by one...
}
```

The array is on the stack.

The objects are on the heap.

Arrays on the Heap

```
void my_function(){
    Cat *Family = new Cat[500];
    int i;
    for ( i=0; i<500; i++ ){
        Family[i].SetAge(2*i+1);
    }
    delete Family; // what are we deleting here??
}
```

The array with all the objects is on the heap.

Deleting Arrays on the Heap

```
void my_function(){
    Cat *Family = new Cat[500];
    int i;
    for ( i=0; i<500; i++ ){
        Family[i].SetAge(2*i+1);
    }
    delete [] Family;
}
```

The operator **delete []** deletes an array!
and calls the destructor on all objects,
e.g. further cleanup. . .

A Pointer to an Array vs. An Array of Pointers

In the following, which is which?

```
Cat FamilyOne[500];
Cat *FamilyTwo[500];
Cat *FamilyThree = new Cat[500];
```

For each three, give the syntax to call SetAge(10) on the
Cat with index 10!

Pointers and Array Names

```
Cat FamilyOne[500];
Cat *FamilyTwo[500];
Cat *FamilyThree = new Cat[500];

FamilyOne[10].SetAge(10);
FamilyTwo[10]->SetAge(10);
FamilyThree[10].SetAge(10);
```

In C++, an array name is a **const** pointer to the first
element of the array.

References

A **reference** is an alias (a second name) for an object.

A **reference** is **not** a pointer.

(But it can achieve similar goals with cleaner syntax.)

```
int &someRef = someInt;
```

A **reference** can only be initialized, but not re-assigned.

Assigning to a Reference

```
int intOne;
int &SomeRef = intOne;

intOne = 5;
cout << intOne << " " << SomeRef << endl;

int intTwo = 8;
SomeRef = intTwo; // what is this ???
cout << intOne << " " << SomeRef << " " << intTwo << endl;
```

What is the output?

Reminder: swap()

```
void swap(int x, int y){
    int temp;
    cout << "swap: before swap, x = " << x << " y = " << y << endl;
    temp = x;
    x = y;
    y = temp;
    cout << "swap: after swap, x = " << x << " y = " << y << endl;
}

int main(){
    int x = 5, y = 10;
    cout << "main: before swap(), x = " << x << " y = " << y << endl;
    swap(x,y);
    cout << "main: after swap(), x = " << x << " y = " << y << endl;
    return 0;
}
```

Making swap() work with Pointers

```
void swap(int *x, int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int x = 5, y = 10;
    cout << "main: before swap(), x = " << x << " y = " << y << endl;
    swap(&x,&y);
    cout << "main: after swap(), x = " << x << " y = " << y << endl;
    return 0;
}
```

swap() now works, but the syntax is spoiled. . .

swap() with References!

```
void swap(int &x, int &y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main(){
    int x = 5, y = 10;
    cout << "main: before swap(), x = " << x << " y = " << y << endl;
    swap(x,y);
    cout << "main: after swap(), x = " << x << " y = " << y << endl;
    return 0;
}
```

Et voilà!

Next Lesson

- More about objects, pointers, references, . . .
- **June 16!**

Reminder: Members of Heap Objects

```
void SlightlyLongerLivedCat(){
    Cat *pCat = new Cat(5); // new calls a constructor

    (*pCat).SetWeight(7); // .-operator applied to the
                        // Cat object at *pCat

    pCat -> SetWeight(7); // -> does the same, is simpler

    long weight = pCat->itsWeight;

    delete pCat; // delete calls the destructor
}
```

Reminder: Pointers and Array Names

```
Cat FamilyOne[500];
Cat *FamilyTwo[500];
Cat *FamilyThree = new Cat[500];

FamilyOne[10].SetAge(10);
FamilyTwo[10]->SetAge(10);
FamilyThree[10].SetAge(10);
```

In C++, an array name is a const pointer to the first element of the array.

Reminder: References

A **reference** is an alias (a second name) for an object.

A **reference** is **not** a pointer.

(But it can achieve similar goals with cleaner syntax.)

int &someRef = someInt;

A **reference** can only be initialized, but not re-assigned.

Reminder: using References

```
void swap(int &x, int &y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main(){
    int x = 5, y = 10;
    cout << "main: before swap(), x = " << x << " y = " << y << endl;
    swap(x,y);
    cout << "main: after swap(), x = " << x << " y = " << y << endl;
    return 0;
}
```

Today:

- The “this” pointer, const pointers
- Call-by-value vs. Call-by-reference
- Pitfalls with references
- The copy-constructor
shallow and deep copies of objects
- Operator overloading

The “this” Pointer

Each method (member function) has a hidden local variable, a pointer to the object itself, called **this**.

```
void Cat::SetWeight(int w){
    itsWeight = w;          // normal use
}
```

```
void Cat::SetWeight(int w){
    this->itsWeight = w;    // via 'this' pointer
}
```

Why have a “this” Pointer?

- Method might return a pointer to the object itself
- Method might pass a pointer to the object itself
- Useful for assignment (=) operator

Reminder: Constants

- **const int studentsPerClass = 15;**
Defines an integer constant.
- A constant can only be initialized.
- A new value can **not** be assigned to a constant.

Reminder: const Member Functions

```
class Cat{
public:
    void Meow() const;
    void SetAge(unsigned int age);
    unsigned int GetAge() const;
    void SetWeight(unsigned long weight);
    unsigned long Getweight() const;
private:
    unsigned int itsAge;
    unsigned long itsWeight;
};
```

What was the meaning of a **const** method?

const Pointers

const int *pOne	pointer to const int
int *const pTwo	const pointer to int
const int *const pThree	const pointer to const int

Not allowed:

```
*pOne = 5;
pTwo = &x;
*pThree = 42;
pThree = &y;
```

In a **const** method, the **this** pointer refers to a **const object**!

Call-by-Value vs. Call-by-Reference

- With Call-by-Value, **copies** of the parameters are given to the function.
- With Call-by-Reference, **references** (or pointers) to the parameters are given to the function.
- Remember: An array name is a const pointer to the first element of the array.

Call-by-Value: Copying Parameters

Objects can be parameters. . .
How do we copy objects???

Copy constructor:

```
Cat::Cat(Cat &source){
    cout << "Cat Copy Constructor\n";
}
```

A copy constructor makes an object from another object of the same class.

How many Cat's are constructed/deconstructed?

```
Cat FunctionOne(Cat theCat){
    cout << "FunctionOne. Returning...\n";
    return theCat;
}
int main(){
    cout << "Making a Cat\n";
    Cat Frisky;
    cout << "Calling FunctionOne\n";
    FunctionOne(Frisky);
    return 0;
}
```

. . . and with Call-by-Reference?

```
Cat &FunctionTwo(Cat &theCat){
    cout << "FunctionTwo. Returning...\n";
    return theCat;
}
int main(){
    cout << "Making a Cat\n";
    Cat Frisky;
    cout << "Calling FunctionTwo\n";
    FunctionTwo(Frisky);
    return 0;
}
```

Don't Return a Reference to an Object Out-of-Scope!

```
Cat &TheFunction(){
    Cat Frisky(5,9);
    return Frisky;
}
```

At the end of TheFunction, Frisky goes out of scope and disappears.

The returned reference is a "stray pointer"!

Returning a Reference to an Object on the Heap

```
Cat &TheFunction(){
    Cat *Frisky = new Cat(5,9);
    return *Frisky;
}
```

And how do we **delete** this object?

Returning a Reference to an Object on the Heap (2)

```
Cat &TheFunction(){  Cat *Frisky = new Cat(5,9);
    return *Frisky;  }

int main(){
    Cat & myCat = TheFunction();
    Cat *pCat = &myCat;
    delete pCat;
    // myCat ??? -> stray pointer :-(
    return 0;
}
```

Responsibility for Pointers

- Better do not return objects that were created in a function.
- Better let the client create them, and pass by reference.
- This gives a single point of responsibility for creating/deleting objects.
- . . . and all that just because C++ does not provide garbage collection.

Initializing Objects (Constructors)

Constructors consist of two phases:

```
Cat::Cat() : // initialization starts here
    itsAge(5) , itsWeight(8)
    { /* body of constructor */ }
```

Assignment of member variables can be in method body and/or initialization list.

Initialization necessary for **references** and **constants**.

Copying Objects with call-by-value

```
Cat FunctionOne(Cat theCat){
    cout << "FunctionOne. Returning...\n";
    return theCat;
}

int main(){
    cout << "Making a Cat\n";
    Cat Frisky;
    cout << "Calling FunctionOne\n";
    Cat Twinky = FunctionOne(Frisky);
    return 0;
}
```

Twinky now is a **copy of a copy** of Frisky.

Twinky is a “clone” (of a clone) of Frisky



The picture shows CC (“copy cat”), the world’s first cloned cat, with Rainbow, her genome donor.

Unlike CC, we need clones that can not be distinguished from their C++ donor object.

Cloning Cats: the Copy Constructor

Unless implemented, the compiler provides a default copy constructor, performing a member-wise copy.

(shallow copy)

Example:

```
Cat::Cat(const Cat &theCat){
    itsAge = theCat.itsAge;
    itsWeight = theCat.itsWeight;
}
```

A copy constructor modifies a newly created “**this**” object.

A Class for “Dogs”

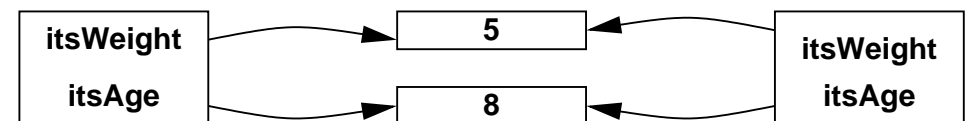
```
class Dog{
public:
    Dog();
    Dog(int age);
    Dog(Dog &source);
    ~Dog();
private:
    int *itsAge;
    long *itsWeight;
}
```

Dog objects store their weight and age on the heap.

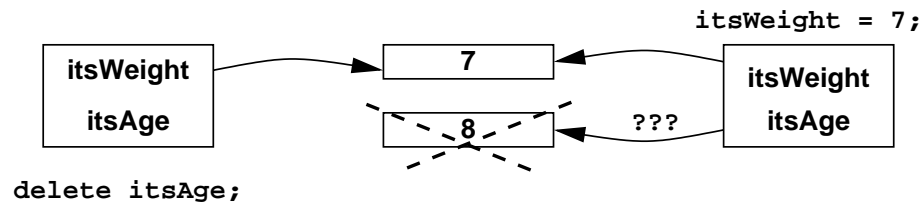
A Copy Constructor for Dogs

The standard, shallow copy:

```
Dog::Dog(const Dog &theDog){
    itsAge = theDog.itsAge;
    itsWeight = theDog.itsWeight;
}
```



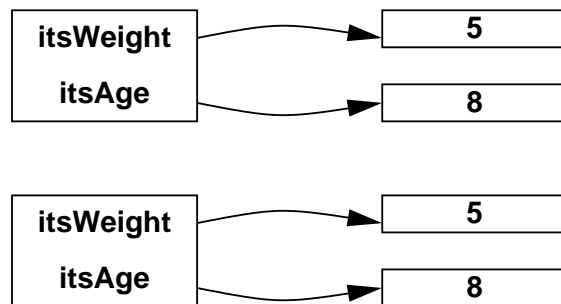
(Obvious) Problems with Shallow Copies



A “Deep” Copy Constructor for Dogs

```
Dog::Dog(const Dog &theDog){
    itsAge = new int;
    itsWeight = new long;
    *itsAge = *(theDog.itsAge);
    *itsWeight = *(theDog.itsWeight);
}
```

Deep Copy



For objects with pointers/references to other objects, **deep** copies have to be made.

Data Types and Operators

- A range of data values, and a set of operations on instances of the type.
- int, long, float: + - */%
- Operations on basic data types are expressed by **operators**.

Classes Define Data Types

- A range of data values, and a set of operations on instances of the type.
- **Cat:**
 - ★ Values: all combinations of itsAge and itsWeight
 - ★ Operations: **methods** (setWeight, getAge, . . .)
 - ★ **operators** like =, and user-defined ones

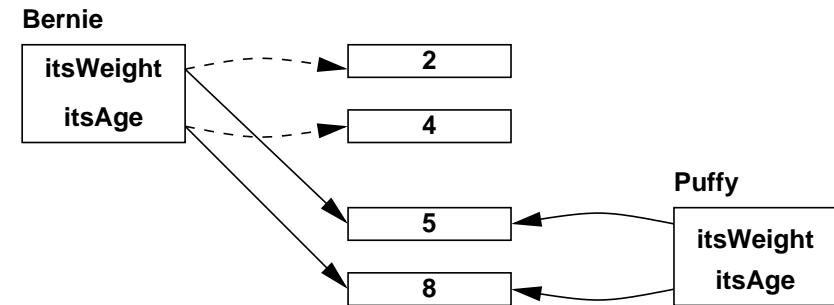
The Assignment Operator '='

- What is the meaning of **Dog Bernie = Puffy;**?
- Like the constructors, a '=' operator is compiler-generated unless implemented explicitly:

```
Dog Dog::operator=(const Dog &source){
    itsAge = source.itsAge;
    itsWeight = source.itsWeight;
    return *this;
}
```

- Is this a copy constructor?
- Is this assignment like we would expect it?

“Shallow” Assignment



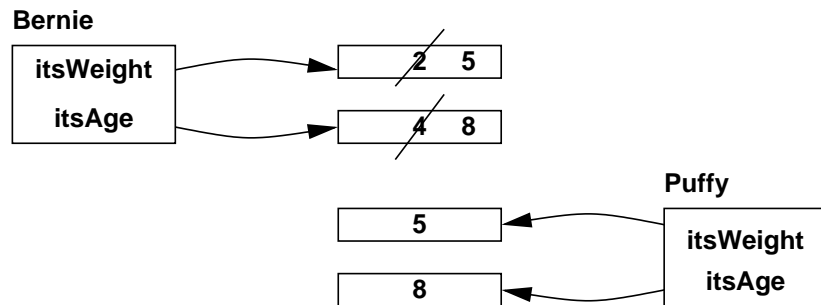
Whenever you implement your own copy constructor, you will also need your own assignment operator!

User-defined Dog Assignment

```
Dog Dog::operator=(const Dog &source){
    *itsAge = *(source.itsAge);
    *itsWeight = *(source.itsWeight);
    return *this;
}
```

For class data members, you may need to delete/new the objects first.

Deep Assignment



Overloading

- **Methods:**
Have multiple methods (functions) with the same name, that can deal with parameters of different types.
- **Operators:**
Have multiple operators with the same name, that can deal with parameters of different types.
The set of operators in C++ is fixed:
(+, -, ++, *, ...)

Why Operator Overloading?

- Sometimes, it is necessary (like '=').
- Mostly, it is “syntactic sugar”:
Operators make reading programs simpler.
Caution: Only use your own operators if their meaning is obvious, otherwise you defeat the purpose of simplicity!
- For those “syntactic sugar” operators, you can achieve the same effect with a normal method.

Syntactic Sugar: iostreams

With operator:

```
cout << 42 << " string\n";
```

With methods (in theory):

```
cout.write(42);
cout.write(" string\n");
```

Next Lesson

- Inheritance
- Polymorphism
- Abstract classes
- **June 30!**

Introduction to Programming, Lesson 8

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

Reminder: Initializing Objects (Constructors)

Constructors consist of two phases:

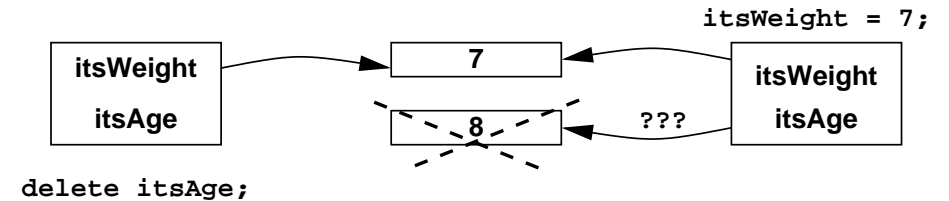
```
Cat::Cat() : // initialization starts here
  itsAge(5) , itsWeight(8)
  { /* body of constructor */ }
```

Assignment of member variables can be in method body and/or initialization list.

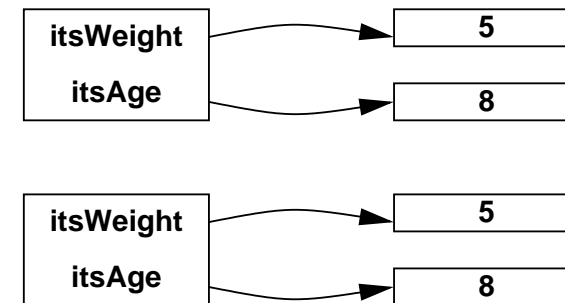
Initialization necessary for **references** and **constants**.

Reminder: Shallow Copies

```
Dog::Dog(const Dog &theDog){
  itsAge = theDog.itsAge;
  itsWeight = theDog.itsWeight;
}
```



Reminder: Deep Copies



For objects with pointers/references to other objects, **deep** copies have to be made.

Reminder:

A “Deep” Copy Constructor for Dogs

```

Dog::Dog(const Dog &theDog){
    itsAge = new int;
    itsWeight = new long;
    *itsAge = theDog->itsAge;
    *itsWeight = theDog->itsWeight;
}

```

Reminder: Assignment Operator

```

Dog Dog::operator=(const Dog &source){
    *itsAge = source->itsAge;
    *itsWeight = source->itsWeight;
    return *this;
}

```

This is a deep assignment, like a deep copy.

Today

- Inheritance
- Polymorphism

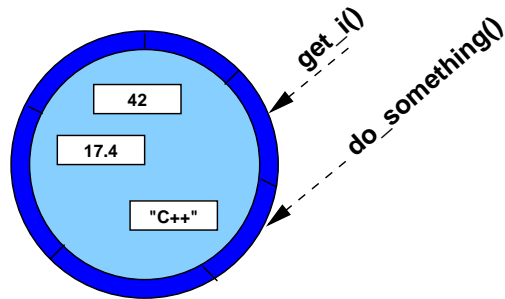
Reminder: Classes Create new Types

A **class** defines a new type:

- a collection of variables (**data members**)
- a set of related functions (**methods, member functions**)
They define the **interface** to objects of the class.

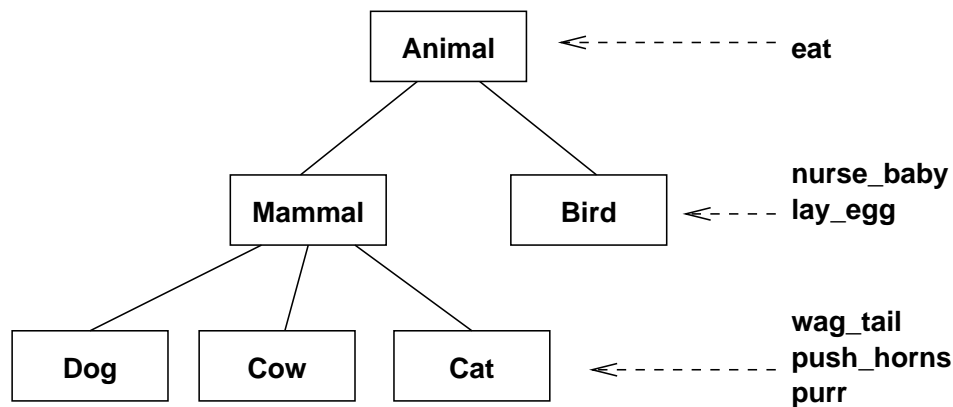
An **object** is an instance (a variable) of a class.

Reminder: Encapsulation



We distinguish the class **interface** from the encapsulated **implementation**.

Inheritance: Refinement/Specialization



Classes are defined according to the needs of an application.

Specialization by Inheritance

Relation between classes:

- Mammal inherits from Animal, and Dog inherits from Mammal.
- A Dog **is a** Mammal which **is an** Animal.
- Animal is a **base class** of Mammal. Animal and Mammal are **base classes** of Dog.
- Mammal is a **derived class** of Animal. Dog and Mammal are **derived classes** of Animal.

Specialization by Inheritance (2)

Class interfaces:

- An Animal can eat.
- A Mammal can eat and nurse a baby.
- A Dog can eat, nurse a baby, and wag its tail.

Uses of Inheritance

1. Conceptual Modelling:

Define a hierarchy of classes that represent the concepts of the data.

This mostly defines hierarchies of interfaces.

2. Code re-use:

Take a given class, and extend it by a little code to get what you need.

A Mammal Class

```
class Mammal{
public:
    Mammal(); ~Mammal();
    int GetAge() const;
    void SetAge(int);
    int GetWeight() const;
    void SetWeight(int);
    void Speak() const;
    void Sleep() const;
protected:
    int itsAge;
    int itsWeight;
};
```

A Dog Class

```
enum BREED{
    YORKIE, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB
};
class Dog : public Mammal{
public:
    Dog(); ~Dog();
    BREED GetBreed() const;
    void SetBreed(BREED);
    void WagTail() const;
    void BegForFood() const;
protected:
    BREED itsBreed;
};
```

Private vs. Public vs. Protected

Class members can be private, public, or protected:

- **private** members can be accessed only within methods of the class itself.
- **protected** members can be accessed from within methods of the class itself, and of its derived classes.
- **public** members can also be accessed from other classes (“code outside the class itself”)

Private vs. Public vs. Protected (2)

Class inheritance can be:

- **public**, the public members of the base class are also public for the derived class.
This is the only case we use!
- **private**, the public members of the base class become private for the derived class.
- **protected**, the public members of the base class become protected for the derived class.

Dog Interface (accumulated)

```
int GetAge() const;
void SetAge(int);
int GetWeight() const;
void SetWeight(int);
void Speak() const;
void Sleep() const;
Dog(); ~Dog();
BREED GetBreed() const;
void SetBreed(BREED);
void WagTail() const;
void BegForFood() const;
```

Dog Data Members (accumulated)

```
int itsAge;
int itsWeight;
BREED itsBreed;
```

Constructors/destructors must deal with data from all (inherited) classes.

Creating a Dog, first the Mammal constructor is called, then the Dog constructor.

Deleting a Dog, first the Dog destructor is called, then the Mammal destructor.

Overloaded Constructors

```
Mammal();
Mammal(int age);
Dog();
Dog(int age);
Dog(int age, int weight);
Dog(int age, BREED breed);
Dog(int age, int weight, BREED breed);
```

Mammal/Dog Constructors

```
Mammal::Mammal() : itsAge(1), itsWeight(5) {}
Mammal::Mammal(int age) : itsAge(age), itsWeight(5) {}

Dog::Dog() : Mammal(), itsBreed(YORKIE) {}
Dog::Dog(int age) : Mammal(age), itsBreed(YORKIE) {}
Dog::Dog(int age, int weight) : Mammal(age),
                               itsBread(YORKIE)
                               { itsWeight = weight; }

Dog::Dog(int age, BREED breed) : Mammal(age),
                               itsBread(breed) {}
Dog::Dog(int age, int weight, BREED breed) : Mammal(age),
                                             itsBreed(breed)
                                             { itsWeight = weight; }
```

Overriding Methods

```
void Mammal::Speak() const{
    cout << "Mammal sound \n";
}

void Dog::Speak() const{
    cout << "Woof! \n";
}
```

The class Dog overrides the Speak method of Mammal.

```
Mammal m.Speak() --> Mammal sound
Dog    d.Speak() --> Woof!
```

Overriding vs. Overloading

- **Overloading:**
Define multiple methods with the same name, but different signature.
- **Overriding:**
Define multiple methods with the same name and signature, but in different classes.
- **Caution:** Overriding means Hiding!
If you override one of many, overloaded methods (with the same name), they you hide all of them!

Hiding, Calling Base Methods

```
void Mammal::Move(int distance) {...}
void Mammal::Move(int distance, int angle) {...}

void Dog::Move(int distance) {...}
    // now, you can not call Dog d.Move(2,4); !!!

    // but you can:
    Dog d.Mammal::Move(2,4); !!
```

Polymorphism

```
Mammal *Bonzo = new Dog();
```

Is it OK to have a `Mammal*` point to a `Dog`?

Polymorphism (2)

What is the output of the following?

```
Dog *Fido = new Dog();
Mammal *Bonzo = Fido;
```

```
Fido->Speak();
Bonzo->Speak();
```

Virtual Methods

```
class Mammal{
public:
    Mammal();
    virtual ~Mammal();
    ..
    virtual void Speak() const;
    ..
};
class Dog : public Mammal {
    ..
    void Speak() const;
    ..
}
```

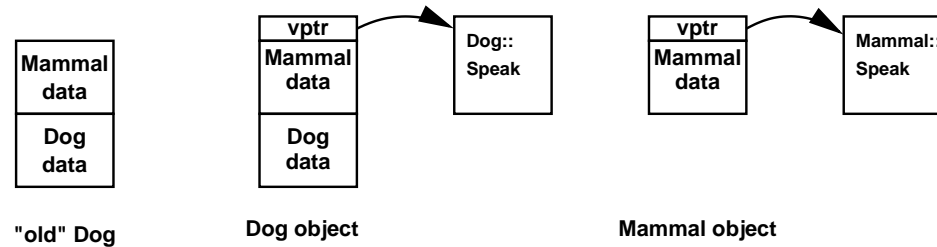
Virtual Methods (2)

And what is now the output of the following?

```
Dog *Fido = new Dog();
Mammal *Bonzo = Fido;
```

```
Fido->Speak();
Bonzo->Speak();
```

V-tables: how Virtual Methods Work



Each object of a class with virtual methods carries a vptr to a v-table of all virtual methods of its **actual** class.

(Objects get bigger, method calls get slower.)

Next Lesson

- Abstract classes
- Container Classes (lists, trees, . . .)
- Templates
- **July 14**

Virtual Destructors

- You would always want to call the destructor of an object's **actual** class.
E.g., call `~Dog()`, even via a pointer to a `Mammal`.
- In fact, destructors should always be virtual.
But this makes all objects bigger and slower.
- **Rule of thumb:**
If you have any virtual methods, then also make the destructor virtual. In this case, you assume to have pointers to base classes.

Introduction to Programming, Lesson 9

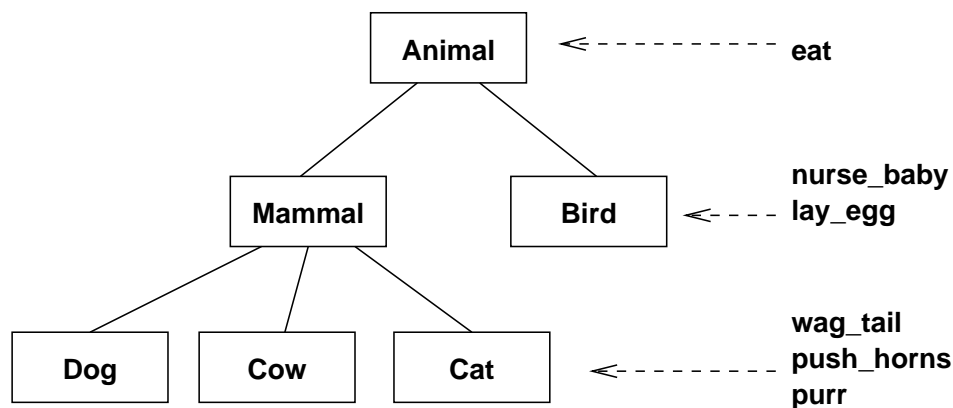
Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

Reminder: Inheritance:



Classes are defined according to the needs of an application.

Reminder: A Dog Class

```

enum BREED{
    YORKIE, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB
};
class Dog : public Mammal{
public:
    Dog(); ~Dog();
    BREED GetBreed() const;
    void SetBreed(BREED);
    void WagTail() const;
    void BegForFood() const;
protected:
    BREED itsBreed;
};
  
```

Reminder: Private vs. Public vs. Protected

Class members can be private, public, or protected:

- **private** members can be accessed only within methods of the class itself.
- **protected** members can be accessed from within methods of the class itself, and of its derived classes.
- **public** members can also be accessed from other classes (“code outside the class itself”)

Reminder: Polymorphism

What is the output of the following?

```
Dog    *Fido = new Dog();
Mammal *Bonzo = Fido;

Fido->Speak(); // Woof!
Bonzo->Speak(); // Mammal sound!
```

Reminder: Virtual Methods

```
class Mammal{
public:
    Mammal();
    virtual ~Mammal();
    ..
    virtual void Speak() const;
    ..
};
class Dog : public Mammal {
    ..
    void Speak() const;
    ..
}
```

Reminder: Virtual Methods (2)

And what is now the output of the following?

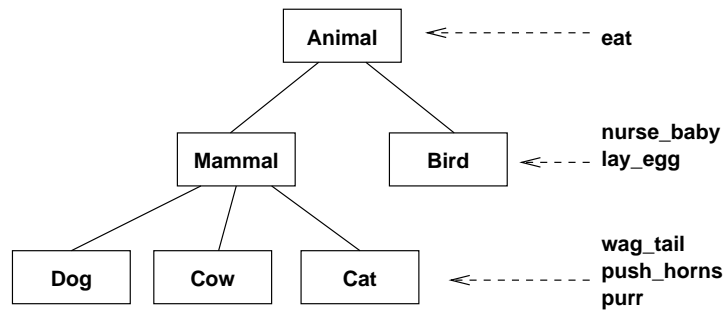
```
Dog    *Fido = new Dog();
Mammal *Bonzo = Fido;

Fido->Speak(); // Woof!
Bonzo->Speak(); // Woof!
```

Today

- Abstract classes
- Container Classes
 - ★ stack
 - ★ queue
 - ★ list
 - ★ tree

Abstract Classes



Animal and Mammal provide useful interfaces, but it does not really make sense having a Mammal object. (Think of the “mammal sound” in Speak(). . .)

Abstract Classes (2)

- In C++, a class is abstract, if it has at least one **pure virtual** method.
- An abstract class in C++ can not be instantiated. (You cannot create an object of an abstract class.)
- However, abstract C++ classes may implement some methods.
- In Java, interfaces serve a similar purpose.

Pure Virtual Methods

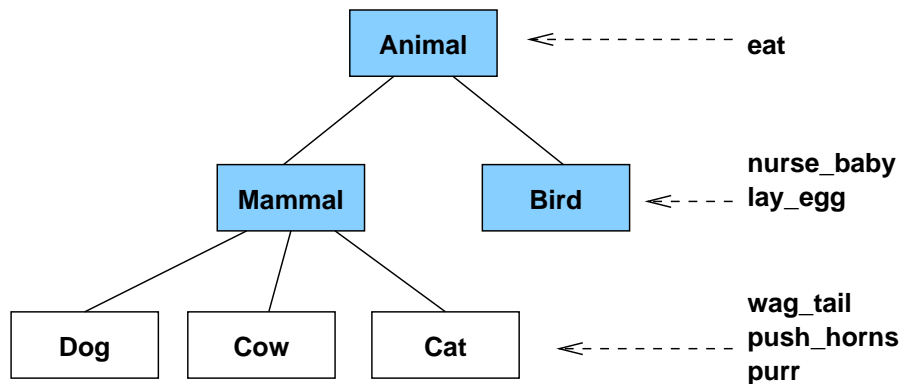
```

class Mammal{
public:
    Mammal();
    virtual ~Mammal();
    ..
    virtual void Speak() const = 0; // pure virtual
    ..
};
class Dog : public Mammal {
    ..
    void Speak() const{ cout << "Woof!\n"; }
    ..
}
  
```

Hierarchies of Abstract/Concrete Classes

- We can define Animal and Mammal as abstract classes. They now simply provide common interfaces for all their subclasses.
- If we inherit Dog, Cow, Cat from Mammal, then we make these classes concrete, so we can have Dog objects that implement the Mammal interface.

Abstract/Concrete Hierarchies (2)



Example: a Hierarchy for Graphics Elements

From last year's exam:

Define an *abstract class* `Element` for graphics elements. The abstract class shall have a method `Circumference` that returns the circumference of the object as a double value.

Class Element

```

class Element{
public:
    virtual ~Element(){}
    virtual double Circumference() const = 0;
};
  
```

Derive real Classes

Implement three classes of graphics elements which are derived from `Element`. For each class, you need to implement a constructor, `Circumference`, and the data members.

1. `Circle`, defined via its radius r .
2. `Rectangle`, defined via two sides a and b .
3. `Square`, derived from `Rectangle`, using $a = b$.

Class Circle

```
class Circle : public Element{
private:
    double myRadius;
public:
    Circle(double radius) : myRadius(radius) {};
    double Circumference() const {
        return 2 * PI * myRadius;
    }
};
```

Class Rectangle

```
class Rectangle : public Element{
private:
    double myA, myB;
public:
    Rectangle(double a, double b)
        : myA(a), myB(b) {};
    double Circumference() const {
        return 2 * (myA+myB);
    }
};
```

Class Square

```
class Square : public Rectangle{
public:
    Square(double a) : Rectangle(a,a) {};
};
```

Use the Element Interface

Implement a function:

```
double sum(const Element *array[],
           const unsigned int len)
```

that computes the sum of the circumferences of the Element objects in the array.

len gives the length of the array.

sum()

```
double sum(const Element *array[],
          const unsigned int len){
    double retval = 0.0;
    unsigned int i;
    for (i=0; i < len; i++){
        retval += array[i]->Circumference();
    }
    return retval;
}
```

Container Classes

- data structures with well-defined access patterns, independent of the type of objects stored inside
- array – simple example
- stack – well-known example
- queue, list, tree – more sophisticated examples

A Stack Class

- sequence of items
- insertions and deletions are done at the top
- main operations: push and pop
- others: full, empty, clear, size

A Queue Class

- sequence of items
- addition at the tail, removal from the head
- main operations: put and get
- others: full, empty, clear, size

Summary

- Inheritance revisited
- Abstract classes
- Container classes (queue example)
- Next lesson:
 - ★ More about containers
 - ★ Templates
 - ★ Exam info

Introduction to Programming, Lesson 10

Uni Siegen, Summer 2003

Lecture: PD Dr.-Ing. Thilo Kielmann
kielmann@cs.vu.nl
Office: H-A 8110

Assignments: Dipl.-Ing. Frank Thilo
thilo@informatik.uni-siegen.de
Office: H-B 8404

<http://www.informatik.uni-siegen.de/kielmann/ip/>

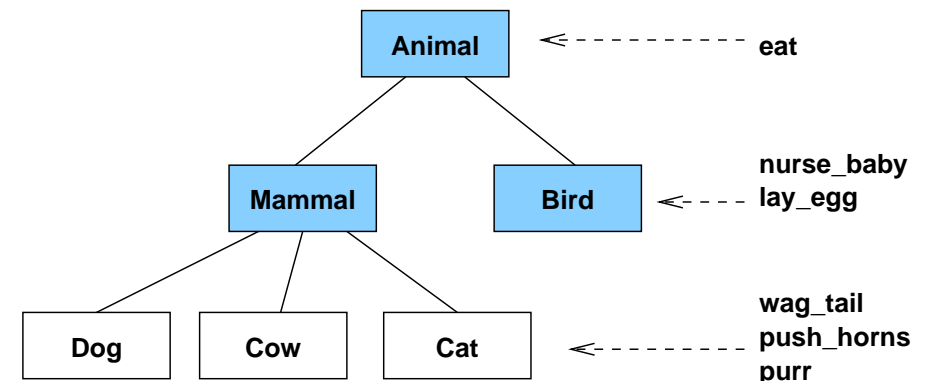
Reminder: Abstract Classes

- In C++, a class is abstract, if it has at least one **pure virtual** method.
- An abstract class in C++ can not be instantiated. (You cannot create an object of an abstract class.)
- However, abstract C++ classes may implement some methods.
- In Java, interfaces serve a similar purpose.

Reminder: Pure Virtual Methods

```
class Mammal{
public:
    Mammal();
    virtual ~Mammal();
    ..
    virtual void Speak() const = 0; // pure virtual
    ..
};
class Dog : public Mammal {
    ..
    void Speak() const{ cout << "Woof!\n"; }
    ..
}
```

Reminder: Abstract/Concrete Hierarchies



Reminder: Container Classes

- data structures with well-defined access patterns, independent of the type of objects stored inside
- array – simple example
- stack – well-known example
- queue, list, tree – more sophisticated examples

Reminder: A Queue Class

- sequence of items
- addition at the tail, removal from the head
- main operations: put and get
- others: full, empty, clear, size

Today

- The Queue Class Revisited
- Templates
- Other Container Classes (lists, trees, . . .)
- All about the exam

The Queue Class Revisited

```
class Queue{
public:
    Queue(unsigned int size);
    ~Queue();
    void put(int i);
    int get();
    bool full() const;
    bool empty() const;
    void clear();
    int size() const;
private:
    int *items;    // array of int
    int max_size; // size of items[]
    int tail;     // position to put
    int head;     // position to get from
    int filled;  // elements in the queue
};
```

Adding Exception Handling

```
#include <stdexcept>
void Queue::put(int i){
    if (filled < max_size){
        items[tail] = i;    tail = (tail+1)%max_size;
        filled++;
    }
    else { throw runtime_error("queue: full on put"); }
}
int Queue::get(){
    int retval;
    if (filled > 0){
        retval = items[head];    head = (head+1)%max_size;
        filled--;
    }
    else{ throw runtime_error("queue: empty on get"); }
    return retval;
}
```

Motivation: Templates

- A Queue is characterized by its interface and behavior (semantics).
- This is independent of the type/class of elements stored inside.
- However, our Queue class can only store **int** values.
- We could rewrite (copy + search-replace) the Queue class for each new class of data elements.
- Or, we could write a **Template** for the Queue class

Template Classes

```
template <class Data> class Queue{
public:
    void put(Data i);
    Data get();
private:
    Data *items; // array of Data
};
```

Use template type parameter as placeholder for final type.

Template Class Queue

```
template <class Data> class Queue{
public:
    Queue(unsigned int size);
    ~Queue();
    void put(Data i);
    Data get();
    bool full() const;
    bool empty() const;
    void clear();
    int size() const;
private:
    Data *items; // array of Data
    int max_size; // size of items[]
    int tail;    // position to put
    int head;    // position to get from
    int filled; // elements in the queue
};
```

Using a Template Class

```
// without template:
Queue *q = new Queue(5);

// with template:
Queue<int> *q = new Queue<int>(5);

Queue<Cat> *qc = new Queue<Cat>(42);
```

The use (instantiation, etc.) of a template is called **specialization**.

What is a Template?

- A Template is just a template (a blueprint) for many possible (specialized) classes.
- A Template is like a (type) definition for the compiler.
- Templates are put into header files.
- Only when they are specialized, the compiler creates actual classes.

queue.h continued

```
template <class Data> Queue<Data>::Queue(unsigned int size){
    items = new Data[size];
    max_size = size;
    tail = 0;
    head = 0;
    filled = 0;
}

template <class Data> void Queue<Data>::put(Data i){
    if (filled < max_size){
        items[tail] = i;
        tail = (tail+1)%max_size;
        filled++;
    }
    else { throw runtime_error("queue: full on put");
    }
}
```

More about Templates

- Templates can have more than one parameter:

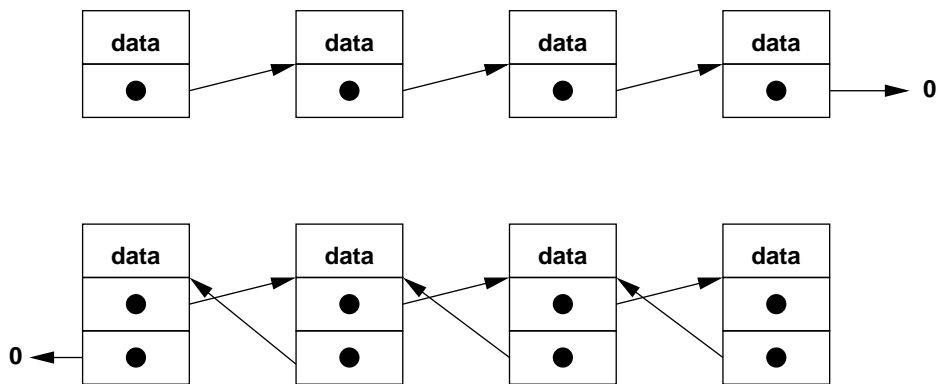

```
template <class type1, class type2, ...>
class MyMultiTemplate
```
- Templates can inherit from other templates and/or from non-template classes
- Non-template classes can inherit from templates:


```
class intQueue : public Queue<int>
```

Other Container Classes

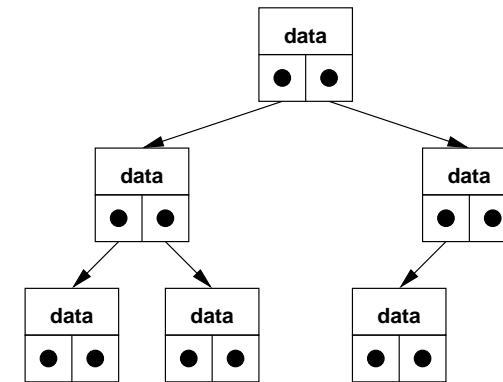
- You already know Stacks and Queues
- In general, a **container** class defines objects that can store (contain) other objects, while having a specific interface for accessing the contained objects.
- Containers for sorting elements:
linked lists and binary trees

Linked Lists



With N elements, access time is $O(N)$.

Binary Trees



With N (sorted) elements, access time is $O(\log_2 N)$.
(if the tree is balanced)

Container Classes

- For all containers, there is a class that holds a data item, plus one or more pointers to other data items.
- There is possibly also a “wrapper” class that provides the container interface, and hides the internal data nodes.
- The interface and implementation is mostly independent of the type of the data items.
- A comprehensive discussion of containers needs another course on “algorithms and data structures” . . .

The Standard Template Library (STL)

- Collection of many data structures and algorithms, written as C++ templates
- Introduced 1994, by Alex Stepanov (HP)
- By now, (modified and) included in the standard C++ libraries
- Documentation:
 - ★ <http://www.sgi.com/tech/stl/>
 - ★ “Thinking in C++” (Bruce Eckel), via class web site

All about the Exam

- Wednesday, September 17, 10:00–11:00 (sharp!), AR-D 5104
- “Open book” exam: all written material allowed (no electronic devices, like computers or phones. . .)
- Bring your own paper, write your name and student number (“Matrikelnummer”) on each sheet that you hand in!
- Have some photo ID with you.

Exam Topics

- If you managed to do the programming assignments, then you will find the exam quite easy.
- Topics: C++ statements, loops, switch, objects, classes, inheritance, polymorphism, templates, . . .
- Results: will be published via the WWW site and the mechatronics blackboard
- Thank you for being here!