Structuur van Computerprogramma’s 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/
Chapter 1 - Introduction
C++ Facts & Figures
Scheme vs. C++

(define (fac n)
  (if (= 0 n)
      1
      (* n (fac (- n 1)))))

(fac 5)

>>>120

A simple matter of getting used to the syntax?

... not really

#include <iostream>
using namespace std;

int fac(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fac(n - 1);
    }
}

int main() {
    cout << fac(5);
    return 0;
}

g++ -Wall -o fac fac.cpp
./fac

>>>120
• **Statically typed** general-purpose programming language

• **Preprocessor, Compiler** and **Linker**

• Focus on **run-time efficiency**
  • Zero-overhead principle

• Focus on **portability**

• Focus on **interoperability**

• **Multiparadigm** programming support
  • structured programming, object-oriented programming, generic programming, data abstraction

“C++ was designed to provide Simula’s facilities for program organization together with C’s efficiency and flexibility for systems programming”

The possibility of automatic garbage collection was considered on several occasions before 1985 and deemed unsuitable for a language already in use for real-time processing and hard-core systems tasks such as device drivers. In those days, garbage collectors were less sophisticated than they are today and the processing power and memory capacity of the average computer were small fractions of what today’s systems offer. My personal experience with Simula and reports of other GC-based systems convinced me that GC was unaffordable by me and my colleagues for the kind of applications we were writing. Had C with Classes (or even C++) been defined to require automatic garbage collection it would have been more elegant, but stillborn.

Option 1: Manual memory management
- Be careful and mindful, don’t introduce memory leaks!

Option II: Use a library-based approach (e.g., libgc)
- Automated protection against memory leaks
- No free’s and delete’s in your program

More about this later...
Take a look at the man pages of gcc:

```
man gcc
```

`g++` is the same as `gcc` but handles the input as C++ (i.e. you can achieve the same result with `gcc` but then you need to set the correct options manually)
Example IDE’s (1)

What's the benefit/drawback?
Example IDE’s (2)

What’s the benefit/drawback?
Brief C++ History

• Originally designed and implemented by Bjarne Stroustrup
  • 1979: start at AT&T Bell Labs, basis was C
  • 1980: “C with Classes” TR in SIGPLAN Notices
  • 1983: officially named C++
  • 1985: first commercial version & book
  • late 80ies: templates & exception handling
  • 1998: standard ISO/IEC 14882 + STL
  • 2003: corrigendum ISO/IEC 14882
  • Today: standard C++0x

Read [STROU93] for a good insight in C++ design considerations
• Standard used in this course
• Reference work
• 786 pages!
• **Save a tree, don’t print it**

**ISO**
International Standard Organization

**IEC**
International Electrotechnical Commission

**ANSI**
American National Standard Institute

What’s the benefit/drawback?
Started in 1979? C++ is old so why should I care...

C++ is statically typed!

C++ for open source development!

C++ is multi-paradigm!

C(++) top-ranked!
Well-known C++ Applications (*)

(source - B. Stroustrup - june 2010 - http://www2.research.att.com/~bs/applications.html)

(*no guarantees about the accuracy, some applications are only partially written in C++
Objectives (Recap)

- Study a language **closer to the hardware and to the operating system**
- Program in a language which requires a **profound knowledge of all the rules and exceptions**
- **No overprotective programmer parenthood** enforced by the language
  - Unsafe areas are open for exploration,
  - Reckless and sloppy programmers will shoot their own feet (yes, both of them!)
- **Debugging beyond “get the algorithm right”**
  - Instead a focus on “**get the right code for the algorithm**”
- Introduce **different programming paradigms**
  - **Structured** programming
  - **Object-Oriented** programming
  - **Generic** programming
- Study an **industry-strength language**
C++ Fundamentals
Overview of Concepts

- Objects, values, and types
- Variables, constants, and expressions
- Lvalue, rvalue
- Object definition, and object manipulation
- Reference variable
- Function definition, function body, formal parameters, actual parameters, return type, return value
- Function calling, function call frame
- Parameter passing, call by value, call by reference
- Identifiers, keywords
- An **object** is a memory area with an **address**
- The contents of an object is a **value**
- An object has a **type** (e.g., `int`) restricting the set of allowed values
- A type implies a **size** used to reserve the amount of memory
  - `sizeof(object) = sizeof(type) = # of bytes to store the object`
- A type supports a set of **operations** (e.g., `+`)
- To create an object of a certain type, you need to:
  - Allocate an object (memory area) of the corresponding size
  - Fill the object with the initial value
Variables, Constants, and Expressions

• How to **reference** a value?
  
  • A **variable** which refers to an object that holds a value
    
    $x \rightarrow$ address $\rightarrow$ value
  
  • A **constant** which refers to a value
    
    123
  
  • An **expression** which refers to a value
    
    $x + 123$

• Two kinds of values:
  
  • An **lvalue** refers to a value that has an address
    
    $x$
  
  • An **rvalue** refers to a value without an address
    
    $x + 123$
ObjectName NameOfVariable(InitialValue);

int x(5); ← initial value provided by a constant
int y(x+7); ← initial value provided by an expression
double z(3.14); ←

Alternative definitions:

int x(5), y(12); ←······ multiple definitions on one line
double z = 3.14; ←······ use of = for assigning initial value

C++ is statically typed!
Object Manipulation: Assignment

LeftExpression = RightExpression

Note: the LeftExpression must always evaluate to an lvalue

```c
int x(5); int y(x+7); double z(3.14);
y = y * 2 + x;
```

![Diagram showing memory and assignment process]

Original memory:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>3.14</td>
</tr>
</tbody>
</table>

After assignment:

```
y = y * 2 + x;
```

Updated memory:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>29</td>
<td>3.14</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
A reference variable holds the address of another object as its value.

The address itself is not accessible to the programmer; the compiler handles the “referencing”.

```c
int x(5);
int& y(x);
y = 6;
```
Reference Variables (2)

NameOfType& NameOfVariable(Expression);

**What if Expression yields an rvalue?**

double& pi(3.14);

1. A fresh object containing the value is created
2. `NameOfVariable` will refer to this object

<table>
<thead>
<tr>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>10030</td>
</tr>
<tr>
<td>pi</td>
</tr>
<tr>
<td>double</td>
</tr>
</tbody>
</table>

What if the object referenced belongs to a different scope?

This is allowed, see functions and parameter passing (a.k.a. **call-by-reference**)

Careful with reference variables and deallocated memory (see later)
Functions

- Basic means for modularizing an implementation
- Structured programming
- Reuse instead of code duplication
- Decrease code size and eliminate potential duplication of errors: easier to maintain!
Function Definition

ReturnType
FunctionName(FormalParamDeclarationList){
    StatementList
}

int x(3);
int y(0);
int z(5);
y = x * x;
y = y + z * z;

int square(int u) {
    return u * u;
}
y = square(x) + square(z);
int x(3);
int y(0);
int z(5);

int square(int u) {
    return u * u;
}

y = square(x + 1);

1. Initialize a fresh **local parameter object** based on the formal parameter declaration and store the value of the actual parameter expression as its value.

2. Execute the statements of the **function body** within a **function call frame** (environment) containing local (u) and non-local names (x, y, z).

3. The return statement is evaluated by creating a fresh object which is initialized with the value of the associated expression.

4. The caller processes the returned object.

5. Deallocate the function call frame.

int u(4);
int tmp
y = 16;
Recursion: Function Call Frame Stack

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

fac(3);
```

1. Every time you enter fac, initialize a fresh **local parameter object** based on the formal parameter declaration and store the value of the actual parameter expression as its value.
2. Execute the statements of the **function body** within a **function call frame** (environment) containing local \((n)\) and non-local names.
3. The return statement is evaluated by creating a fresh object which is initialized with the value of the associated expression.
4. The caller processes the returned object.
5. Deallocate the function call frame.
C++ only supports **call by value**, so the value of the actual parameters is always copied into the local parameter objects.

Problematic if you pass around large objects.

Call by value on parameters of reference type is equivalent to **call by reference** (a.k.a. call by address).

```cpp
int x(3);
int y(4);

void swap(int& a, int& b) {
    int temp(a);
    a = b;
    b = temp;
}
swap(x, y);
```
What happens if we execute the following program?

```c++
int x(3);
int y(4);
int& z(x);

int& square(int u) {
    int result(0);
    result = u * u;
    return result;
}

z = square(y);
swap(square(x), square(y));
```

Once the call frame of `square(x + 1)` is popped from the stack, **the reference to the memory location of the local variable is no longer trustworthy!**

A bug that is **difficult to trace/reproduce**

The compiler generates a **warning** (not an error) not to be ignored!

```
../src/mathFunctions.cpp:57: warning: reference to local variable 'result' returned
```

A bug that is **difficult to trace/reproduce**

The compiler generates a **warning** (not an error) not to be ignored!

```
../src/mathFunctions.cpp:57: warning: reference to local variable 'result' returned
```
The Anatomy and Execution of a C++ Program
Overview of Concepts

- Translation unit
- Function and variable declarations
- Function and variable definitions
- Preprocessor, compiler, linker
- Header files, source files, object files, executable files
- Directives
- Macro processor, macro language
- #ifndef wrappers
- Controlling/guard macro’s
- Object code relocation
- Scopes, namespaces
- Lexical scoping, dynamic scoping
Program Organization

- A C++ program consists of **translation units**
- A translation unit contains **definitions** and/or **declarations of types, variables, and functions**
- A **declaration** tells the compiler about the existence of something so you can start to use its name (even before it is defined)

```c
extern double pi;
double circle_area(double radius);
```

- A **definition** causes the compiler to:
  - Generate code for a function definition
  - Allocate memory for a variable definition
  - Compute the size and other properties of a newly defined type
An **include file** (a.k.a. header file) contains **declarations** of types, variables, and functions defined in another translation unit. As such it declares the **interface** of a particular translation unit.

A **source file** contains **definitions** of types, variables, and functions.

The **preprocessor** is a macro processor that interprets **directives** such as `#include`, `#define`, `#ifdef`, `#ifndef`.

The **compiler** checks the program for errors and translates it to machine code (a.k.a. object code).

The **linker** connects all the pieces together (e.g. references to libraries).
The C Preprocessor (a.k.a. CPP)

- Implements a **macro language** used to transform C++ programs before they are compiled.

- The **macro processor** will make systematic text replacements in the input files provided.

- The macro language provides a set of constructs to direct this transformation.
  - **Object-like macros**: resemble data objects
    - Similar to a data-object that is being used in code
    - Commonly used to give symbolic names to constants
  - **Function-like macros**: resemble function calls
    - Similar to a function that is being called in code
    - Commonly used to expand larger portions of code


Read it!
CPP: Object-like Macros

#define ObjectLikeMacroName TokenSequence

- A simple identifier that is replaced by a code fragment
- The code fragment is a.k.a. the macro expansion, macro body, or macro replacement list
- There is no restriction, as long as it decomposes into valid preprocessing tokens
- Non-recursive resolution of macro names
  - So if its name occurs in its body then that name will not be expanded
  - Note: other macro names in the body will be expanded!
- Avoids infinite expansion of self-referential macros
- By convention macro names are uppercase

```
#define BUFFER_SIZE 1024

foo = (char *) malloc (BUFFER_SIZE);  \equiv  foo = (char *) malloc (1024);
```
When macro names are expanded, the macro’s expansion replaces the invocation of the macro. After that the expansion is checked for containing macro names itself which are also expanded.
CPP: Function-like Macros

#define FunctionLikeMacroName() TokenSequence

- Function-like macros are only expanded if the macro invocation has parentheses

```c
#define lang_init() c_init()
```

```c
lang_init() ≈ c_init()
```

```c
extern void foo(void);
#define foo() /* optimized version, no function call as body */
...
value = foo();
funcptr = foo;
```

- What happens if you put a space between the name and the parentheses?
  - It is treated as an object-like macro definition whose body starts with `()`
CPP: Header Files

#include <filename>

Used for system header files, lookup starts in a standard list of system directories

#include “filename”

Used for header files of your own program, lookup starts in the current directory, then in the quoted directories, then in the standard list

- CPP will copy the contents of the header file into each source file that requires it
- Avoid redeclarations with a “wrapper #ifndef” (see example later)
- Benefits of working with header files:
  - Changes in the interface can occur in one place
  - All programs that use an interface will include a new version upon recompilation
  - Inconsistencies are avoided
- Naming convention: myfile.h
● **Source files** (translation units) contain definitions
  ● `circle.cpp` contains the **definition** of the `circle_area` function
  ● `pi.cpp` contains the **definition** of the `pi` variable
  ● `prog1.cpp` contains the **definition** of the `main` function

● **Include files** contain declarations
  ● `circle.h` contains the **declaration** of the `circle_area` function
  ● `pi.h` contains the **declaration** of the `pi` variable

● **File dependencies**
  ● `prog1.cpp` needs `circle.h`
  ● `circle.cpp` needs `pi.h` (and `circle.h`)
  ● `pi.cpp` (needs `pi.h`)
Program Organization: Circle Example (2)

**Wrapper**  
`#ifndef`

**Controlling Macro**  
(a.k.a. guard macro)  
`#define`

**Single Line Comments**  
`//`

```c
#ifndef PI_H
    // begin conditional inclusion:  
    // read body until #endif ONLY if PI_H is not defined

#define PI_H
    // define variable PI_H, so next time ...

extern double pi;  // declaration of variable pi

#endif  // end of conditional inclusion
```

**Extern** indicates that the definition is elsewhere
Program Organization: Circle Example (3)

#include "pi.h"  // include declaration to ensure that
    // declaration and definition of pi
    // are consistent

double pi(3.14); // definition of variable pi

What if the declaration is different from the definition?

If the declaration and definition are not consistent
then the compiler complains:

../pi.cpp:11: error: conflicting declaration 'float pi'
../pi.h:12: error: 'pi' has a previous declaration as 'double pi'
make: *** [pi.o] Error 1
#ifndef CIRCLE_H
#define CIRCLE_H
// The ifndef define .. endif sequence ensures
// that the following declaration is not read twice
// (see also pi.h)

// declaration of function 'circle_circumference'
double circle_circumference(double radius);

// declaration of function 'circle_area'
double circle_area(double radius);
#endif
#include "pi.h"  // declaration of pi
#include "circle.h"  // declaration of functions
    // that are defined in this file:
    // including them ensures the consistency of
    // the following definitions with the declarations

double   // function definition
circle_circumference(double radius) {
    return 2 * pi * radius;
}

double   // function definition
circle_area(double radius) {
    return pi * radius * radius;
}
#include <iostream> // contains needed declarations,  
    // e.g. for operator<<(ostream&, X)

#include "circle.h" // contains needed declaration  
    // of circle_area

int main() { // function definition
    double r(35); // local variable definition  
        // r will live in the call frame
        // the next expression writes the area of  
        // the circle to standard output

    std::cout << circle_area(r) << std::endl;

    return 0; // all is well: return 0
}

std::cout is the standard output stream (console)  
    << is the insertion operator  
    std::endl represents an endline
When a C++ program is executed, the function `main` is automatically called.

- There can only be one `main` function in your program.
- An `int` is returned when the program finishes its execution.
  - By convention, `0` means that the execution was without errors.
  - Any other value indicates an abnormal program termination.
- Outside the `main` function you can use the `exit(int status)` function to terminate the program.
  - For abnormal termination only, otherwise use exception handling (later).
- The return value is testable from the outside.
  - For example from within a Linux shell:
    ```sh
    ./prog1 || echo " :error level"
    ``
A main function can have two arguments: \texttt{argc} and \texttt{argv}

This enables passing arguments when starting a program

\texttt{argc} : the number of arguments supplied

\texttt{argv} : the values of the arguments provided

\begin{verbatim}
#include <iostream>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        std::cout << "Error: usage -> sayHello aName" << std::endl;
    } else {
        std::cout << "Hello " << argv[1] << std::endl;
    }
    return 0;
}
\end{verbatim}

For example from within a Linux shell:

\texttt{./sayHello "Mister"} \Rightarrow \texttt{Hello Mister}
- Resolves symbols (e.g., `circle_area` in `prog1.o`)
- Relocates object code

**RESOLVING SYMBOLS**

! = defines
? = uses, but does not define

**C++ Compiler**
- Preprocessor
- Compiler
  - `#include`
  - `source file(s)`
  - `object file(s)`
- Linker
  - `executable file`

**object files**
- `circle.o`
  - `!circle_circumference`
  - `!circle_area`
  - `?pi`
- `pi.o`
  - `!pi`
- `prog1.o`
  - `!main`
  - `?circle_area`

**executable file**
- `prog1`
  - `!main`
  - `!circle_circumference`
  - `!circle_area`
  - `!pi`
Circle Example before Linking - prog1.o

00000000 <main>:
  0:  55          pushl  %ebp
...
  1a:  dd 1c 24   fstpl (%esp,1)
  1d:  e8 fc ff ff ff ff  call  1e <main+0x1e>
...

Address in memory, still needs to be relocated

?circle_area

Still needs to be resolved

prog1.o

!main

still needs to be resolved
Circle Example after Linking - prog1.o

- Resolves symbols
  (e.g., `circle_area` in `prog1.o`)
- Relocates object code

```
08048710 <main>:
8048710:  55              pushl   %ebp
...
804872a:  dd 1c 24        fstpl   (%esp,1)
804872d:  e8 36 00 00 00  call    8048768 <circle_area(double)>
...
08048768 <circle_area(double)>:
8048768:  55              pushl   %ebp
```
The Compilation Process: Overview

C++ Compiler

#include

Preprocessor

Compiler

Linker

object file(s)

circle.o

executable file

prog1

source file(s)
circle.cpp

include file(s)

pi.h
C++ Lexical Considerations

- Name of defined type/function/variable: **identifier**:  
  - Start with letter or underscore `_`
  - Contains letters, digits, or underscore `_`
  - Convention: `MyClass, my_variable, my_simple_function, MY_CONSTANT`
  - Case sensitive: `student ≠ Student`

- Not a **keyword** (see list on next slide)

Choosing good names is crucial for understandable programs

- Comments: `//` until end of line
  - Should be consistent with the code
  - Should have an added value

Important documentation for code users and code maintainers (and yourself)
Keywords are reserved as part of the C++ language, they have a fixed meaning which cannot be changed by the programmer.

<table>
<thead>
<tr>
<th>asm</th>
<th>else</th>
<th>operator</th>
<th>throw</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>enum</td>
<td>private</td>
<td>true</td>
</tr>
<tr>
<td>bool</td>
<td>explicit</td>
<td>protected</td>
<td>try</td>
</tr>
<tr>
<td>break</td>
<td>extern</td>
<td>public</td>
<td>typedef</td>
</tr>
<tr>
<td>case</td>
<td>false</td>
<td>register</td>
<td>typeid</td>
</tr>
<tr>
<td>catch</td>
<td>float</td>
<td>reinterpret_cast</td>
<td>typename</td>
</tr>
<tr>
<td>char</td>
<td>for</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>class</td>
<td>friend</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>const</td>
<td>goto</td>
<td>signed</td>
<td>using</td>
</tr>
<tr>
<td>const_cast</td>
<td>if</td>
<td>sizeof</td>
<td>virtual</td>
</tr>
<tr>
<td>continue</td>
<td>inline</td>
<td>static</td>
<td>void</td>
</tr>
<tr>
<td>default</td>
<td>int</td>
<td>static_cast</td>
<td>volatile</td>
</tr>
<tr>
<td>delete</td>
<td>long</td>
<td>struct</td>
<td>wchar_t</td>
</tr>
<tr>
<td>do</td>
<td>mutable</td>
<td>switch</td>
<td>while</td>
</tr>
<tr>
<td>double</td>
<td>namespace</td>
<td>template</td>
<td></td>
</tr>
<tr>
<td>dynamic_cast</td>
<td>new</td>
<td>this</td>
<td></td>
</tr>
</tbody>
</table>
Scopes

- C++ programs may consist of thousands of translation units
  - Name conflicts may arise (e.g., when reusing a library)
- Names are organized hierarchically in (nested) scopes
- Some constructs, e.g., a class type or a function body automatically define a new scope (nested in the encompassing scope)

```cpp
double x(0.12345);
double z(1.2);

int f(int y){
    int x(y+1);
    return x*z;
}

z = f(3) + x;
```
namespace A {
    int x;
    namespace C { int x; }
}

namespace B { int x; }

int main() {
    int z(2);
    //error: which x?
    z = x;
}

// possible correction for z=x
z = A::C::x; // ok
// other possibility:
using A::C::x;
z = x;
namespace A {
    namespace C {
        extern int x;
    } // only a declaration
}

namespace B {
    int x;
}

int A::C::x(5); // definition of name "x" declared in ::A::C
namespace A {
    int x;
} // continuation of namespace A

int main() {
    int z(2);
    using namespace A::C;
    z = x; // thus "x" refers to A::C::x
}
Recommended Reading

  Bjarne Stroustrup, ACM HOPL-II, 1993

  Bjarne Stroustrup, ACM HOPL-III, 2007

- **The C Preprocessor**