Introduction to C++
and Object Oriented Programming

Wouter Verkerke (Nikhef)

v71 – Edition for 2023 Master Course

1

Introduction and Overview

0 Introduction & Overview

© 2006 Wouter Verkerke, NIKHEF

2

Intended audience and scope of course

- This course is targeted to students with some programming experience in procedural (i.e. non-OO) programming languages like Fortran, C, Pascal
  - No specific knowledge of C, C++ is assumed
- This course will cover
  - Basic C/C++ syntax, language features
  - Basics of object oriented programming
- This course has some extra focus on the application of C++ in (High Energy) Physics
  - Organized processing and analysis of data
  - Focus mostly in exercises

© 2006 Wouter Verkerke, NIKHEF

3

Programming, design and complexity

- The goal of software – to solve a particular problem
  - E.g. computation of numeric problems, maintaining an organized database of information, finding the Higgs etc..
- Growing computational power in the last decades has allowed us to tackle more and more complex problems
- As a consequence software has also grown more powerful and complex
  - For example MacOS, last generation video games, often well over 1.000.000 lines of source code
  - Growth also occurs in physics: e.g. collection of software packages for reconstruction/analysis of the BaBar experiment is ~6.4M lines of C++
- How do we deal with such increasing complexity?

© 2006 Wouter Verkerke, NIKHEF

4

Programming philosophies

- Key to successfully coding complex systems is break down code into smaller modules and minimize the dependencies between these modules

- Traditional programming languages (C, Fortran, Pascal) achieve this through procedure orientation
  - Modularity and structure of software revolves around 'functions' encapsulate (sub) algorithms
  - Functions are a major tool in software structuring but leave a few major design headaches

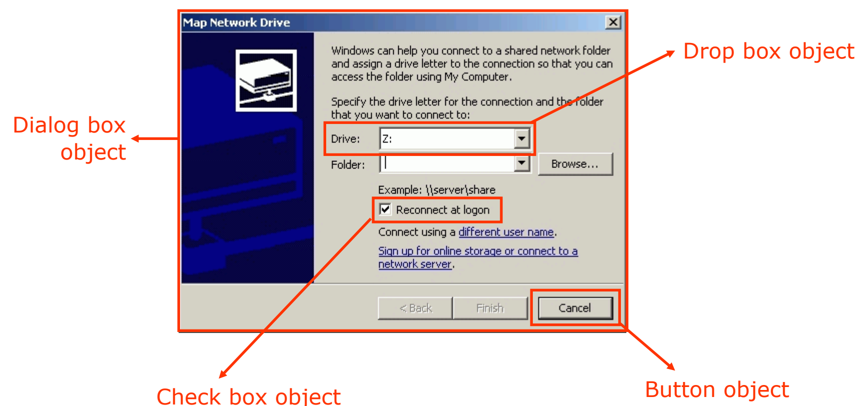
- Object-oriented languages (C++, Java,...) take this several steps further
  - Grouping data and associated functions into objects
  - Profound implications for modularity and dependency reduction

© 2006 Wouter Verkerke, NIKHEF

5

What are objects

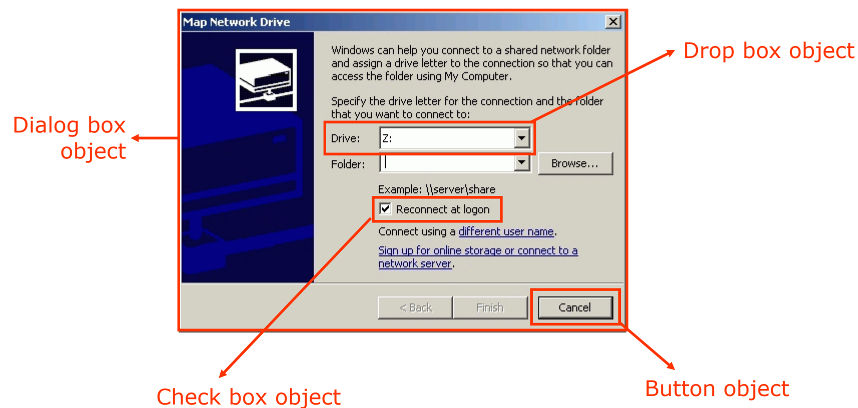
- 'Software objects' are often found naturally in real-life problems
- Object oriented programming → Finding these objects and their role in your problem



6

What are objects

- An object has
  - Properties : position, shape, text label
  - Behavior : if you click on the 'Cancel button' a defined action occurs

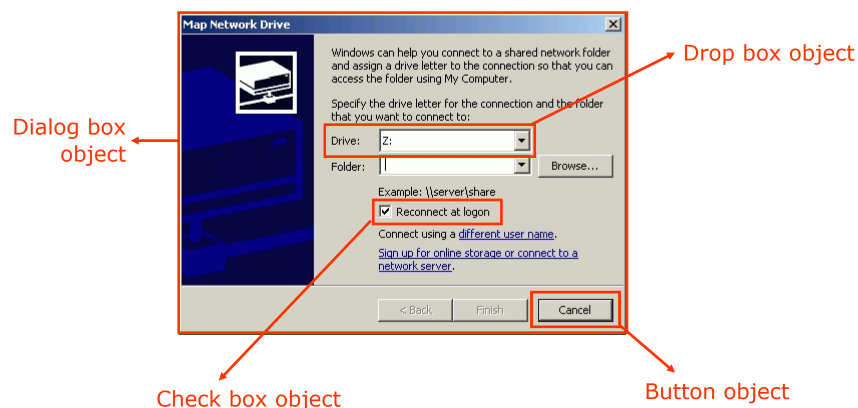


© 2006 Wouter Verkerke, NIKHEF

7

Relating objects

- Object-Oriented Analysis and Design seeks the relation between objects
  - 'Is-A' relationship (a PushButton Is-A ClickableObject)
  - 'Has-A' relationship (a DialogBox Has-A CheckBox)



8

Benefits of Object-Oriented programming

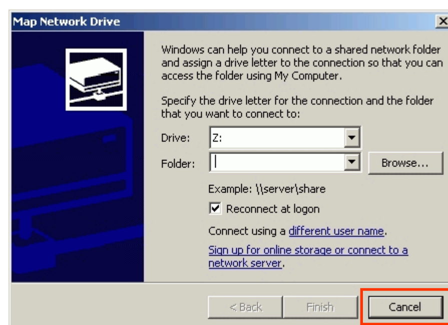
- Benefits of Object-oriented programming
  - Reuse of existing code – objects can represent generic problems
  - Improved maintainability – objects are more self contained than 'subroutines' so code is less entangled
  - Often a 'natural' way to describe a system – see preceding example of dialog box
- But…
  - Object oriented modeling does not substitute for sound thinking
  - OO programming does not *guarantee* high performance, but it doesn't stand in its way either
- Nevertheless
  - *OO programming is currently the best way we know to describe complex systems*

© 2006 Wouter Verkerke, NIKHEF

9

Basic concept of OOAD

- Object-oriented programming revolves around *abstraction* of your problem.
  - Separate *what you do* from *how you do it*
- *Example – PushButton object*



PushButton is a complicated piece of software – Handling of mouse input, drawing of graphics etc..

Nevertheless you can use a PushButton object and don't need to know anything about that. Its public interface can be very simple: My name is 'cancel' and I will call function `doTheCancel()` when I get clicked

© 2006 Wouter Verkerke, NIKHEF

10

Techniques to achieve abstraction

- Abstraction is achieved through

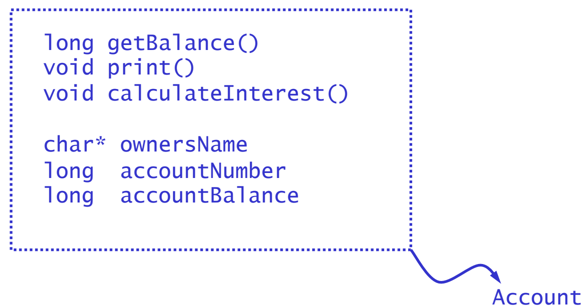
1. **Modularity**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**

© 2006 Wouter Verkerke, NIKHEF

11

Modularity

- Decompose your problem logically in independent units
  - Minimize dependencies between units – **Loose coupling**
  - Group things together that have logical connection – **Strong cohesion**
- Example
  - Grouping actions and properties of a bank account together

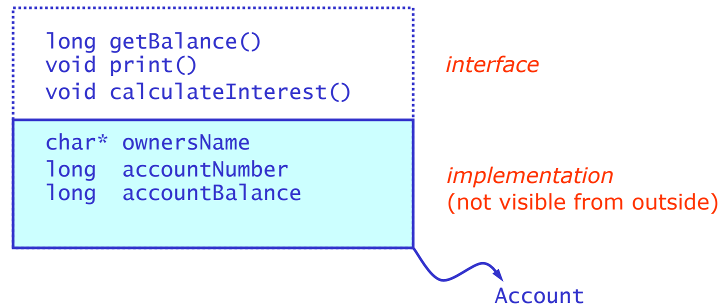


© 2006 Wouter Verkerke, NIKHEF

12

Encapsulation

- Separate interface and implementation and shield implementation from object 'users'

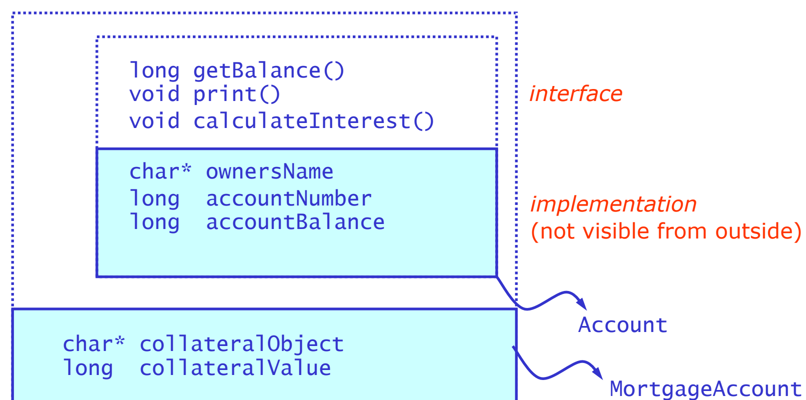


© 2006 Wouter Verkerke, NIKHEF

13

Inheritance

- Describe new objects in terms of existing objects
- Example of mortgage account

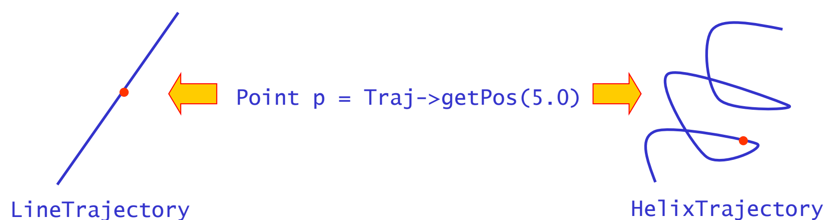


© 2006 Wouter Verkerke, NIKHEF

14

Polymorphism

- Polymorphism is the ability to treat objects of different types the same way
  - You don't know exactly what object you're dealing with but you know that you can interact with it through a standardized interface
  - Requires some function call decisions to be taken at run time
- Example with trajectories
  - Retrieve position at a flight length of 5 cm
  - Same interface works for different objects with identical interface



15

Introduction to C++

- Wide choice of OO-languages – why program in C++?
  - It depends on what you need…
- Advantage of C++ – It is a compiled language
  - When used right the fastest of all OO languages
  - Because OO techniques in C++ are resolved and implemented at compile time rather than runtime so
    - **Maximizes run-time performance**
    - **You don't pay for what you don't use**
- Disadvantage of C++ – syntax more complex
  - Also, realizing performance advantage not always trivial
- C++ best used for large scale projects where performance matters
  - C++ rapidly becoming standard in High Energy Physics for mainstream data processing, online data acquisition etc…
  - Nevertheless, if your program code will be O(100) lines and performance is not critical C, Python, Java may be more efficient

© 2006 Wouter Verkerke, NIKHEF

16

Versions of C++

- C++ is a 'living language' that evolves over time.
- This course is largely based on the 2003 standard of C++
- LHC experiments are now largely adopting C++ compilers that implement the 2011 standard of C++, which brings useful new features
  - E.g. Auto types, range-based for loops, lambdas, constructor delegation, tuples, hash tables and pointer memory management
  - I will cover a subset of these C++2011 features in this course, and explicitly point out the features that are only available in C++2011
- For the GNU compilers (gcc/g++) some of the C++2011 features are implement starting in version 4.4, with almost all features implemented in 4.7
  - In gcc 4.[3456] must add flag '-std=c++0x' to activate
  - In gcc 4.[78] must add flag '-std=c++11' to activate

© 2006 Wouter Verkerke, NIKHEF

17

Outline of the course

1. Introduction and overview
2. Basics of C++
3. Modularity and Encapsulation – Files and Functions
4. Class Basics
5. Object Analysis and Design
6. The Standard Library I – Using IOstreams
7. Generic Programming – Templates
8. The Standard Library II – The template library
9. Object Orientation – Inheritance & Polymorphism
10. Robust programming – Exception handling
11. Where to go from here

© 2006 Wouter Verkerke, NIKHEF

18

The basics of C++

1 **The basics of C++**

© 2006 Wouter Verkerke, NIKHEF

19

"Hello world" in C++

- Lets start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

© 2006 Wouter Verkerke, NIKHEF

20

"Hello world" in C++

- Lets start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

Anything on line after // in C++ is considered a comment

© 2006 Wouter Verkerke, NIKHEF

21

"Hello world" in C++

- Lets start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "He
  return 0;
}
```

Lines starting with # are directives for the preprocessor

Here we include some standard function and type declarations of objects defined by the 'iostream' library

- The preprocessor of a C(++) compiler processes the source code before it is passed to the compiler. It can:
  - Include other source files (using the #include directive)
  - Define and substitute symbolic names (using the #define directive)
  - Conditionally include source code (using the #ifdef, #else, #endif directives)

© 2006 Wouter Verkerke, NIKHEF

22

"Hello world" in C++

- Let start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```


- The main() function is the default function where all C++ programs begin their execution.
  - In this case the main function takes no input arguments and returns an integer value
  - You can also declare the main function to take arguments which will be filled with the command line options given to the program

© 2006 Wouter Verkerke, NIKHEF

23

"Hello world" in C++

- Lets start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```


- The names `std::cout` and `std::endl` are declared in the 'header file' included through the '#include <iostream>' preprocessor directive.
- The std::endl directive represents the 'carriage return / line feed' operation on the terminal

© 2006 Wouter Verkerke, NIKHEF

24

"Hello world" in C++

- Lets start with a very simple C++ program

```
// my first program in C++
#include <iostream>

int main () {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

The return statement passes the return value back to the calling function

- The return value of the main() function is passed back to the operating system as the 'process exit code'

© 2006 Wouter Verkerke, NIKHEF

25

Compiling and running 'Hello World'

- Example using Linux, (t)csh and g++ compiler

```
unix> g++ -o hello hello.cc

unix> ./hello
Hello World!

unix> echo $status
0
```

Convert c++ source code into executable

Run executable 'hello'

Print exit code of last run process (=hello)

© 2006 Wouter Verkerke, NIKHEF

26

Outline of this section

- Jumping in: the 'hello world' application
- Review of the basics
  - **Built-in data types**
  - **Operators on built-in types**
  - **Control flow constructs**
  - **More on block {} structures**
  - Dynamic Memory allocation

```
int main() {
  int a  = 3 ;
  float b = 5 ;

  float c = a * b + 5 ;

  if ( c > 10) {
     return 1 ;
  }

  return 0 ;
}
```

© 2006 Wouter Verkerke, NIKHEF

Review of the basics – built-in data types

- C++ has only few built-in data types

type name	type description
`char`	ASCII character, 1 byte
`int`, `signed int, unsigned int, short int, long int`	Integer. Can be signed, unsigned, long or short. Size varies and depends on CPU architecture (2,4,8 bytes)
`float, double`	Floating point number, single and double precision
`bool`	Boolean, can be true or false (1 byte)
`enum`	Integer with limited set of named states `enum fruit { apple,pear,citrus }`, or `enum fruit { apple=0,pear=1,citrus}`

- More complex types are available in the 'Standard Library'
  - A standard collection of tools that is available with every compiler
  - But these types are not fundamental as they're implement using standard C++
  - We will get to this soon

© 2006 Wouter Verkerke, NIKHEF

Defining data objects – variables

- Defining a data object can be done in several ways

```
int main() {
   int j ;     // definition – initial value undefined
   int k = 0 ; // definition with assignment initialization
   int l(0) ;  // definition with constructor initialization

   int m = k + l ; // initializer can be any valid C++ expression

   int a,b=0,c(b+5); // multiple declaration – a,b,c all integers
}
```

- Data objects declared can also be declared constant

```
int main() {
   const float pi = 3.14159268 ; // constant data object
   pi = 2 ; // ERROR – doesn't compile
}
```

© 2006 Wouter Verkerke, NIKHEF

29

Defining data objects – variables

- Const variables must be initialized

```
int main() {
   const float pi ;      // ERROR – forgot to initialize

   const float e = 2.72; // OK
   const float f =  5*e ;// OK – expression is constant
}
```

- Definition can occur at any place in code

```
int main() {
   float pi = 3.14159268 ;
   cout << "pi = " << pi << endl ;

   float result = 0; // 'floating' declaration OK
   result = doCalculation() ;
}
```

  - Style tip: always declare variables as close as possible to point of first use

30

Literal constants for built-in types

- Literal constants for integer types

```
int j = 16 ;   // decimal
int j = 0xF ;  // hexadecimal (leading 0x)
int j = 020 ;  // octal (leading 0)

unsigned int k = 4294967280U ; // unsigned literal
```

  – Hex, octal literals good for bit patterns (hex digit = 4 bits, octal digit = 3 bits)
  – Unsigned literals good for numbers that are too large for signed integers (e.g. between 2^32/2 and 2^32-1)

- Literal constants for character types

```
char ch = 'A' ; // Use single quotes
```

  – Escape sequences exist for special characters →

`\n`	newline
`\r`	carriage return
`\t`	tabulation
`\v`	vertical tabulation
`\b`	backspace
`\f`	page feed
`\a`	alert (beep)
`\'`	single quotes (')
`\"`	double quotes (")
`\?`	question (?)
`\\`	inverted slash (\)

31

Auto declaration type **(C++ 2011)**

- In C++ 2011, you can also omit an explicit type in declarations of objects that are immediately initialized
- In these cases the type is deduced from the initializer

```
auto j = 16 ;    // j is integer
auto j = 2.3 ;   // j is double
auto j = true ;  // j is bool
```

© 2006 Wouter Verkerke, NIKHEF

32

Arrays

- C++ supports 1-dimensional and N-dimensional arrays
  - Definition

```
Type name[size] ;
Type name[size1][size2]…[sizeN] ;
```

  - Array dimensions in definition must be constants

```
float x[3] ;     // OK

const int n=3 ;
float x[n] ;     // OK

int k=5 ;
float x[k] ;     // ERROR!
```

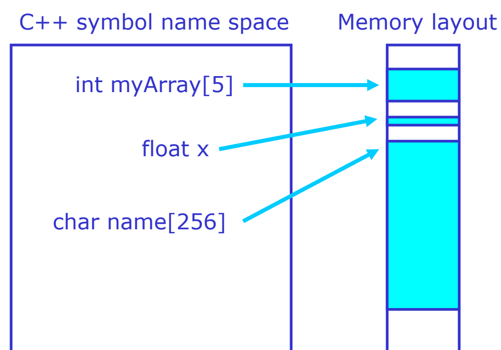
  - *First element's index is always **0***
  - Assignment initialization possible

```
float x[3]     = { 0.0, 5.7 , 2.3 } ;
float y[2][2]  = { 0.0, 1.0, 2.0, 3.0 } ;
float y[3]     = { 1.0 } ; // Incomplete initialization OK
```

33

Declaration versus definition of data

- Important fine point: definition of a variable is two actions
  1. Allocation of memory for object
  2. Assigning a symbolic name to that memory space



  - C++ symbolic name is a way for programs to give understandable names to segments of memory
  - But it is an artifact: no longer exists once the program is compiled

34

References

- C++ allows to create 'alias names', a different symbolic name referencing an already allocated data object
  - Syntax: '`Type& name = othername`'
  - References do not necessarily allocate memory
- Example

```
int x ;        // Allocation of memory for int
               // and declaration of name 'x'
int& y = x ;   // Declaration of alias name 'y'
               // for memory referenced by 'x'

x = 3 ;
cout << x << endl ; // prints '3'
cout << y << endl ; // also prints '3'
```

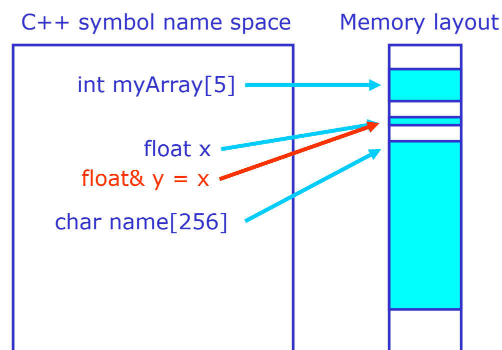
  - Concept of references will become more interesting when we'll talk about functions

© 2006 Wouter Verkerke, NIKHEF

35

References

- Illustration C++ of reference concept
  - Reference is symbolic name that points to same memory as initializer symbol

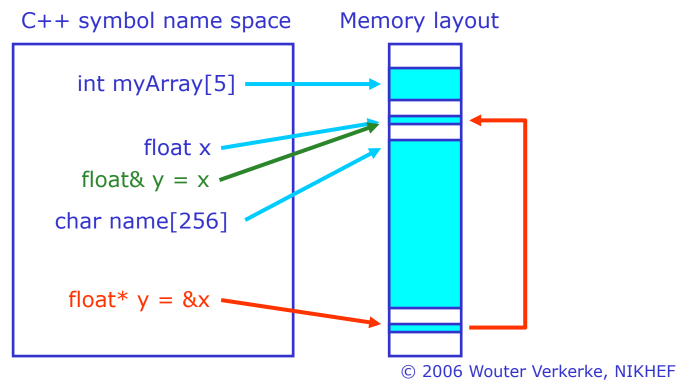


© 2006 Wouter Verkerke, NIKHEF

36

Pointers

- Pointer is a variable that contains a memory address
  - Somewhat similar to a reference in functionality, but fundamentally different in nature: a pointer is always an object in memory itself
  - Definition: '**TYPE* name**' makes pointer to data of type TYPE



© 2006 Wouter Verkerke, NIKHEF

37

Pointers

- Working with pointers
  - Operator & takes memory address of symbol object (=pointer value)
  - Operator * turns memory address (=pointer value) into symbol object
- Creating and reading through pointers

```
int x = 3, y = 4 ;
int* px ;            // allocate px of type 'pointer to integer'
px = &x ;            // assign 'memory address of x' to pointer px

cout << px << endl ; // Prints 0x3564353, memory address of x
cout << *px << endl ;// Prints 3, value of x, object pointed to by px
```

- Modifying pointers and objects pointed to

```
*px = 5 ;            // Change value of object pointed to by px (=x) ;
cout << x << endl ;  // Prints 5 (since changed through px)
px = &y ;            // Reseat pointer to point to symbol named 'y'

cout << px << endl ; // Prints 0x4863813, memory address of y
cout << *px << endl ;// Prints 4, value of y, object pointed to by px
```

© 2006 Wouter Verkerke, NIKHEF

38

Pointers continued

- Pointers are also fundamentally related to arrays

```
int a[3]  = { 1,2,3} ; // Allocates array of 3 integers
int* pa   = &a[0] ;    // Pointer pa now points to a[0]

cout << *pa << endl ;      // Prints '1'
cout << *(pa+1) << endl ; // Prints '2'
```

- Pointer (pa+1) points to next element of an array
  - This works regardless of the type in the array
  - In fact **a** itself is a pointer of type **int*** pointing to **a[0]**
- The **Basic Rule** for arrays and pointers
  - **a[i] is equivalent to *(a+i)**

© 2006 Wouter Verkerke, NIKHEF

Pointers and arrays of char – strings

- Some special facilities exist for arrays of char
  - char[] holds strings and is therefore most commonly used array
- Initialization of character arrays:
  - String literals in double quotes are of type 'char *', i.e.

```
const char* blah = "querty";
```

  is equivalent to

```
const char tmp[7] = {'q','w','e','r','t','y',0} ;
const char* blah = tmp ;
```

  - Recap: single quoted for a single char, double quotes for a const pointer to an array of chars
- Termination of character arrays
  - Character arrays are by convention ended with a null char (\0)
  - Can detect end of string without access to original definition
    - For example for strings returned by "a literation expression"

© 2006 Wouter Verkerke, NIKHEF

Strings and string manipulation

- Since char[] strings are such a common object
  - the 'Standard Library' provides some convenient manipulation functions
- Most popular char[] manipulation functions

```
//  Length of string
int strlen(const char* str) ;

// Append str2 to str1 (make sure yourself str1 is large enough)
char* strcat(char* str1, const char* str2) ;

// Compares strings, returns 0 if strings are identical
int strcmp(const char* str1, const char* str2) ;
```

- Tip: Standard Library also provides 'class string' with superior handling
  - We'll cover class string later
  - But still need 'const char*' to interact with operating system function calls (open file, close file, etc)

© 2006 Wouter Verkerke, NIKHEF

41

Reading vs. Writing – LValues and RValues

- C++ has two important concepts to distinguish read-only objects and writeable objects
  - An **LValue** is writable and can appear on the left-hand side of an assignment operation
  - An **RValue** is read-only and may only appear on the right-hand side of assignment operations
- Example

```
int i;
char buf[10] ;

i = 5 ; // OK, i is an lvalue
5 = i ; // ERROR, 5 is not an lvalue
        // (it has no memory location)

buf[0] = 'c' ;   // OK buf[0] is an lvalue
buf = "qwerty" ; // ERROR, buf is immutably tied to char[10]
```

© 2006 Wouter Verkerke, NIKHEF

42

Operators and expressions – arithmetic operators

- Arithmetic operators overview

Name	Operator
Unary minus	-x
Multiplication	x * y
Division	x / y

Name	Operator
Modulus	x % y
Addition	x + y
Subtraction	x - y

- Arithmetic operators are evaluated from **left to right**
  - `40 / 4 * 5 = (40 / 4) * 5 = 50` (not 2)

- In case of mixed-type expressions compiler automatically converts integers up to floats

```
int i = 3, j = 5 ;
float x = 1.5 ;

float y = i*x ; // = 4.5 ; int i promoted to float
float z = j/i ; // = 1.0 ; '/' has precedence over '='
```

43

Operators and expressions – increment/decrement operators

- In/Decrement operators

Name	Operator
Prefix increment	++x
Postfix increment	x++
Prefix decrement	--x
Postfix decrement	x--

- Note difference
  - ***Prefix*** operators return value ***after*** operation
  - ***Postfix*** operators return value ***before*** operation
- Examples

```
int x=0 ;
cout << x++ << endl ; // Prints 0
cout << x << endl ;   // Prints 1

cout << ++x << endl ; // Prints 2
cout << x << endl ;   // Prints 2
```

© 2006 Wouter Verkerke, NIKHEF

44

Operators and expressions – relational operators

- Relational operators

Name	Operator
Less than	`x < y`
Less than or equal to	`x <= y`
Greater than or equal to	`x >= y`
Greater than	`x > y`
Equal to	`x == y`
Not equal to	`x != y`

- All relational operators yield `bool` results
- Operators `<`,`<=`,`>=`,`>` have precedence over `==`, `!=`

© 2006 Wouter Verkerke, NIKHEF

Operators and expressions – Logical operators

- Logical operators

Name	Operator
Logical NOT	`!x`
Logical AND	`x>3 && x<5`
Logical OR	`x==3 || x==5`

Do not confuse with bit-wise AND (&) and bit-wise OR (|)

- All logical operators take `bool` arguments and return `bool`
  - If input is not `bool` it is *converted* to `bool`
  - Zero of any type maps to `false`, anything else maps to `true`
- Logical operators are evaluated from left to right
  - *Evaluation is **guaranteed to stop** as soon as outcome is determined*

```
float x, y ;
…
if (y!=0. && x/y < 5.2) ; // safe against divide by zero
```

Operators and expressions – Bitwise operators

- Bitwise operators

Name	Operator	Example
Bitwise complement	~x	0011000 → 1100111
Left shift	x << 2	000001 → 000100
Right shift	x >> 3	111111 → 000111
Bitwise AND	x & y	1100 & 0101 = 0100
Bitwise OR	x | y	1100 | 0101 = 1101
Bitwise XOR	x ^ y	1100 ^ 0101 = 1001

- Remarks
  - Bitwise operators cannot be applied to floating point types
  - Mostly used in online, DAQ applications where memory is limited and 'bit packing is common'
  - Do not confuse logical or, and (||,&&) with bitwise or, and (|,&)

© 2006 Wouter Verkerke, NIKHEF

47

Operators and expressions – Assignment operators

- Assignment operators

Name	Operator
Assignment	x = 5
Addition update	x += 5
Subtraction update	x -= 5
Multiplication update	x *= 5
Division update	x /= 5
Modulus update	x %= 5
Left shift update	x <<= 5
Right shift update	x >>= 5
Bitwise AND update	x &= 5
Bitwise OR update	x |= 5
Bitwise XOR update	x ^= 5

© 2006 Wouter Verkerke, NIKHEF

48

Operators and expressions – Assignment operators

- Important details on assignment operators
  - **Left-hand** arguments must be **lvalues** (naturally)
  - Assignment is evaluated **right to left**
  - Assignment operator **returns left-hand** value of expression

- Return value property of assignment has important consequences
  - **Chain** assignment is possible!

```
x = y = z = 5 ;  // OK! x = ( y = ( z = 5 ))
                 //     x = ( y = 5)
                 //     x = 5
```

  - **Inline** assignment is possible

```
int x[5], i ;
x[i=2] = 3 ; // i is set to 2, x[2] is set to 3
```

© 2006 Wouter Verkerke, NIKHEF

49

Operators and expressions – Miscellaneous

- Inline conditional expression: the ternary `?:` operator
  - Executes inline if-then-else conditional expression

```
int x = 4 ;
cout << ( x==4 ? "A" : "B" ) << endl ; // prints "A" ;
```

- The comma operator (`expr1, expr2, expr3`)
  - Evaluates expressions sequentially, returns *rightmost* expression

```
int i=0, j=1, k=2 ;
cout<< (i=5, j=5, k) <<endl ; // Prints '2', but i,j set to 5
```

- The `sizeof` operator
  - Returns size in bytes of operand, argument can be *type* or *symbol*

```
int size1 = sizeof(int) ; // = 4 (on most 32-bit archs)

double x[10] ;
int size2 = sizeof(x) ;   // = 10*sizeof(double)
```

50

Conversion operators

- Automatic conversion
  - All type conversions that can be done 'legally' and without loss of information are done automatically
  - Example: float to double conversion

```
float f = 5 ;
double d = f ; // Automatic conversion occurs here
```

- Non-trivial conversions are also possible, but not automatic
  - Example: `float` to `int`, `signed int` to `unsigned int`
  - If conversion is non-trivial, conversion is not automatic → you must request it with a ***conversion operator***
- C++ has a variety of ways to accomplish conversions
  - C++ term for type conversion is '**cast**'
  - Will focus on 'modern' methods and ignore 'heritage' methods

© 2006 Wouter Verkerke, NIKHEF

51

Conversion operators – Explicit casts

- For conversions that are 'legal' but may result in truncation, loss of precision etc…: **static_cast**

```
float f = 3.1 ;
int i = static_cast<int>(f) ; // OK, i=3 (loss of precision)
int* i = static_cast<int*>(f) ; // ERROR float != pointer
```

- For conversions from 'const X' to 'X', i.e. to override a logical const declaration: **const_cast**

```
float f = 3.1 ;
const float& g = f ;
g = 5.3 ;                      // ERROR not allowed, g is const
float& h = const_cast<float&>(g) ; // OK g and h of same type
h = 5.3 ;                      // OK, h is not const
```

© 2006 Wouter Verkerke, NIKHEF

52

Conversion operators – Explicit casts

- Your last resort: **reinterpret_cast**

```
float* f ;
int* i = reintepret_cast<int*>(f) ; // OK, but you take
                                     // responsibility for the ensuing mess…
```

- You may need more than one cast to do your conversion

```
const float f = 3.1 ;
int i = static_cast<int>(f) ; // ERROR static_cast cannot
                              // convert const into non-const

const float f = 3.1 ;
int i = static_cast<int>( const_cast<float>(f) ) ; // OK
```

  – It may look verbose but it helps you to understand your code as all aspects of the conversion are explicitly spelled out

© 2006 Wouter Verkerke, NIKHEF

53

Control flow constructs – if/else

- The `if` construct has three formats
  – Parentheses around expression required
  – Brackets optional if there is only one statement (but put them anyway)

```
if (expr) {
   statements ; // evaluated if expr is true
}
```

```
if (expr) {
  statements ; // evaluated if expr is true
} else {
  statements ; // evaluated if expr is false
}
```

```
if (expr1) {
  statements ;     // evaluated if expr1 is true
} else if (expr2) {
  statements ;     // evaluated if expr2 is true
} else {
  statements ;     // evaluated if neither expr is true
}
```

54

Intermezzo – coding style

- C++ is free-form so there are no rules
- But style matters for readability, some suggestions
  - One statement per line
  - **Always put {} brackets** even if statement is single line
  - Common indentation styles for {} blocks

```
if (foo==bar) {
    statements ;
} else {
    statements ;
}
```

```
if (foo==bar)
{
    statements ;
}
else
{
    statements ;
}
```

Try to learn yourself this style, it is more compact and more readable (especially when you're more experienced)

© 2006 Wouter Verkerke, NIKHEF

55

Control flow constructs – while

- The while construct

```
while (expression) {
  statements ;
}
```

  - Statements will be executed if expression is true
  - At end, expression is re-evaluated. If again true, statements are again executed

- The do/while construct

```
do {
  statements ;
} while (expression) ;
```

  - Similar to **while** construct except that statements are always executed once before expression is evaluated for the first time

© 2006 Wouter Verkerke, NIKHEF

56

Control flow constructs – for

- The for construct

```
for (expression1 ; expression2 ; expression3) {
  statements ;
}
```

  - is equivalent to

```
expression1 ;
while (expression2) {
  statements ;
  expression3 ;
}
```

- Most common looping construct

```
int i ;
for (i=0 ; i<5 ; i++) {
  // Executes with i=0,1,2,3 and 4
}
```

© 2006 Wouter Verkerke, NIKHEF

57

Control flow constructs – for

- Expressions may be empty

```
for (;;) {
  cout << "Forever more" << endl ;
}
```

- Comma operator can be useful to combine multiple operations in expressions

```
int i,j;
for (i=0,j=0 ; i<3 ; i++,j+=2) {
  // execute with i=0,j=0, i=1,j=2, i=2,j=4
}
```

© 2006 Wouter Verkerke, NIKHEF

58

Control flow constructs – break and continue

- Sometimes you need to stop iterating a do, do/while or for loop prematurely
  - Use break and continue statements to modify control flow

- The break statement
  - Terminate loop construct *immediately*

```
int i = 3 ;
while(true) { // no scheduled exit from loop
  i -= 1 ;
  if (i<0) break ; // exit loop
  cout << i << endl ;
}
```

  - Example prints '2', '1' and '0'. Print statement for i=-1 never executed

© 2006 Wouter Verkerke, NIKHEF

59

Control flow constructs – break and continue

- The continue statement
  - Continue stops execution of loops statements and *returns to evaluation of conditional expression*

```
char buf[12] = "abc,def,ghi" ;
for (int i=0 ; i<12 ; i++) {
  if (buf[i]==',') continue ; // return to for()
                              // if ',' is encountered
  cout << buf[i] ;
}
cout << endl ;
```

  - Output of example 'abcdefghi'
  - Do not confuse with FORTRAN 'continue' statement -- Very different meaning!

- Both break and continue only affect the *innermost* loop
  - When you are using nested loops

© 2006 Wouter Verkerke, NIKHEF

60

Control flow constructs – switch

- The switch construct

```
switch (expr) {
  case constant1:
    statements ; // Evaluated if expr==const1
    break ;

  case constant2:
  case constant3:
    statements ; // Evaluated if expr==const2 or const3
    break ;

  default:
    statements ; // Evaluated expression matched none
    break ;
}
```

  - Most useful for decision tree algorithms
- If break is omitted execution continues with next case evaluation
  - Usually you don't want this, so watch the breaks

61

Control flow constructs – switch (example)

- switch works very elegantly with enum types
  - enum naturally has finite set of states
- case expressions must be constant but can be any valid expression
  - Example:

```
enum color { red=1, green=2, blue=4 };
color paint = getcolor() ;
switch (paint) {
  case red:
  case green:
  case blue:
    cout << "primary color" << endl ;
    break ;

  case red+green:
    cout << "yellow" << endl ;
    break ;

  case red+blue:
    cout << "magenta" << endl ;
    break ;

  case blue+green:
    cout << "cyan" << endl ;
    break ;

  default:
    cout << "white" << endl ;
    break ;
}
```

62

Some details on the block {} statements

- Be sure to understand all consequences of a block {}
  - The lifetime of automatic variables inside the block is limited to the end of the block (i.e up to the point where the } is encountered)

```
int main() {
  int i = 1 ;

  if (x>0) {
    int i = 0 ;
    // code
  } else {
    // code
  }
}
```


  - A block introduces a new scope : it is a separate namespace in which you can define new symbols, even if those names already existed in the enclosing block

© 2006 Wouter Verkerke, NIKHEF

63

Scope – more symbol visibility in {} blocks

- Basic C++ scope rules for variable definitions
  - In given location all variables defined in local scope are visible
  - All variables defined in enclosing scopes are visible
  - Global variables are always visible
  - Example

```
int a ;
int main() {
   int b=0 ;

   if (b==0) {
      int c = 1;
   }

}
```


© 2006 Wouter Verkerke, NIKHEF

64

Scoping rules – hiding

- What happens if two variables declared in different scopes have the same name?
  - Definition in inner scope ***hides*** definition in outer scope
  - It is legal to have two variables with the same name defined in different scopes

```
int a ;
int main() {
   int b=0 ;                  ← 'b' declared in main() visible

   if (b==0) {
LEGAL! → int b = 1;           ← 'b' declared in if() visible
   }                            'b' declared in main() hidden!

}
```

  - NB: It is not legal to have two definitions of the same name in the same scope, e.g.

```
int main() {
  int b ;
  …
  int b ;   ERROR!
}
```

© 2006 Wouter Verkerke, NIKHEF

65

Scoping rules – The :: operator

- Global variables, even if hidden, can always be accessed using ***the scope resolution operator ::***

```
int a=1 ;

int main() {
   int a=0 ;    ← LEGAL, but hides global 'a'

   ::a = 2 ;    ← Changes global 'a'
}
```

- No tools to resolve symbols from intermediate unnamed scope
  - Solution will be to use 'named' scopes: namespaces or classes
  - More on classes later

© 2006 Wouter Verkerke, NIKHEF

66

More on memory use

- By default all objects defined *outside* {} blocks (global objects) are allocated statically
  - Memory allocated before execution of `main()` begins
  - Memory released after `main()` terminates
- By default all defined objects defined *inside* {} blocks are 'automatic' variables
  - Memory allocated when definition occurs
  - Memory released when closing bracket of scope is encountered

```
if (x>0) {
  int i = 0 ;     <- Memory for 'int i' allocated
  // code
}                 <- Memory for 'int i' released
```

  - You can override behavior of variables declared in {} blocks to be statically allocated using the `static` keyword

© 2006 Wouter Verkerke, NIKHEF

67

More on memory allocation

- Example of static declaration

```
void func(int i_new) {
   static int i = 0 ;
   cout << "old value = " << i << endl ;
   i = i_new ;
   cout << "new value = " << i << endl ;
}

int main() {
  func(1) ;
  func(2) ;
}
```

  - Output of example

```
old value = 0 ;
new value = 1 ;
old value = 1 ; Value of static int i preserved between func() calls
new value = 2 ;
```

© 2006 Wouter Verkerke, NIKHEF

68

Dynamic memory allocation

- Allocating memory at run-time
  - When you design programs you cannot always determine how much memory you need
  - You can allocate objects of unknown size at compile time using the 'free store' of the C++ run time environment
- Basic syntax of runtime memory allocation
  - Operator new allocates single object, returns pointer
  - Operator new[] allocates array of objects, returns pointer

```
// Single object
Type* ptr = new Type ;
Type* ptr = new Type(initValue) ;

// Arrays of objects
Type* ptr = new Type[size] ;
Type* ptr = new Type[size1][size2]…[sizeN] ;
```

© 2006 Wouter Verkerke, NIKHEF

69

Releasing dynamic memory allocation

- Operator delete releases dynamic memory previously allocated with new

```
// Single object
delete ptr ;

// Arrays of objects
delete[] ptr ;
```

  - **Be sure to use delete[] for allocated arrays**. A mismatch will result in an incomplete memory release
  - **The delete operator only deletes memory that the pointer points to, not pointer itself**
  - **Every call to new must be matched with a call to a delete**
- How much memory is available in the free store?
  - As much as the operating system lets you have
  - If you ask for more than is available your program will terminate in the new operator
  - It is possible to intercept this condition and continue the program using 'exception handling' (we'll discuss this later)

70

Dynamic memory and leaks

- A common problem in programs are memory leaks
  - Memory is allocated but never released even when it is not used anymore
  - Example of leaking code

```
void leakFunc() {
  int* array = new int[1000] ;
  // do stuff with array
}

int main() {
  int i ;
  for (i=0 ; i<1000 ; i++) {
    leakFunc() ; // we leak 4K at every call
  }
}
```

Leak happens right here (at the closing `}` of `leakFunc()`)
we loose the pointer `array` *here and with that our only possibility to release its memory in future*

© 2006 Wouter Verkerke, NIKHEF

71

Dynamic memory and leaks

- Another scenario to leak memory
  - Misunderstanding between two functions

```
int* allocFunc() {
  int* array = new int[1000] ;
  // do stuff with array
  return array ;
}

int main() {
   int i ;
   for (i=0 ; i<1000 ; i++) {
     allocFunc() ;
   }
}
```

`allocFunc()` allocates memory but pointer as return value memory is not leaked yet (at `return array ;`)

Author of `main()` doesn't know that it is supposed to delete array returned by `allocFunc()` (at `allocFunc() ;`)

Leak occurs here, pointer to dynamically allocated memory is lost before memory is released (at `allocFunc() ;`)

© 2006 Wouter Verkerke, NIKHEF

72

Dynamic memory and ownership

- Avoiding leaks is a matter of good bookkeeping
  - All memory allocated should be released after use
- Memory handling logistics usually described in terms of **ownership**
  - The 'owner' of dynamically allocated memory is responsible for releasing the memory again
  - **Ownership is a 'moral concept'**, not a C++ syntax rule. Code that never releases memory it allocated is legal, but may not work well as program size will increase in an uncontrolled way over time
  - Document your memory management code in terms of ownership

© 2006 Wouter Verkerke, NIKHEF

73

Dynamic memory allocation

- Example of dynamic memory allocation with ownership semantics
  - Less confusion about division of responsabilities

```
int* makearray(int size) {
   // NOTE: caller takes ownership of memory
   int* array = new int[size] ;

   int i ;
   for (i=0 ; i<size ; i++) {
     array[i] = 0 ;
   }
   return array;
}

int main() {
  // Note: We own array
  int* array = makearray(1000) ;

  delete[] array ;
}
```

© 2006 Wouter Verkerke, NIKHEF

74