CS 225 Data Structures

Lab 1: Introduction to C++ / Imperative Core

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

1 The Imperative Core of C++

1.1 Data Types

C++ is a strongly typed programming language, i.e. every object must have a well-specified type (or class)

Data types for **integers**:

- **short** (8-16 bits: -32768 . . . 32767)
- **int** (16 bits: -32768 . . . 32767)
- **long** (16-32 bits: -2147483648 . . . 2147483647)

Operators on integer data types: addition (+), subtraction (-), multiplication (*), division (/), modulo (%), increment (++), decrement (−)

Comparison operators are equality (==), inequality (!=) less than (<), less than or equal (<=), greater than (>), greater or equal (>=)

Note that **++i** differs from **i++** in the order of evaluation: **i++ / j** may give a different result than **++i / j**

Natural Numbers, Booleans, Characters, Enumerations

All integer data types provide an **unsigned** version, e.g. in **unsigned int x**

These unsigned types are used to represent non-negative integers, i.e. natural numbers—thus, the range of **unsigned int** is 0 . . . 65535

Constants of type **long** are marked by suffix **L**, unsigned constants by suffix **U**

The data type for truth values is **bool** with only two values **true** and **false**

Comparison operators yield values of type **bool**

Operators on Booleans are negation (**!**), conjunction (**&&**) and disjunction (**| |**), as well as the comparison operators **==** and **!=**

The data type for characters is **char** with values in single quotes, e.g. **'A'**

void is data type with no values

An **enumeration type** is defined by a finite set of named constants, e.g.
enum traffic_light **{** red, yellow, green **}**

Floating Point Numbers

Data types for **floating point numbers** are:

- **float** using 4 bytes, e.g. **3.142f** or **2.9979e8f**
- **double** using 8 bytes, e.g. **3.1415926535897932** or **2.997925e8**
- **long double** using 10 or 12 bytes, e.g. **0.5778143296e3L**

Operators and comparison operators are the same as for integers except %

There are automatic conversions between integers and floating point numbers

Variable declarations take the form **int x**, **double y = 2.997925e8**

For **assignments** use the operator = and the shortcuts +=, −=, *=, /=, %=

Constant declarations use the keyword **const** as in **const double pi = 3.141592653589793**

1.2 Control Structure

Every imperative language uses sequences of statements, so does C++; statements usually need a **;** at the end

Compound statements (aka **blocks**) are sequences of statements within braces: **{ . . . }**

Most importantly, the **scope** of a variable or constant declared inside a block is within the block

```
#include ⟨iostream⟩
using namespace std;
void main()
{ int x = 10;
        {   double x;
            x = 1.772;
            cout << x << "\n" ;        (prints double value 1.772)
        }
  cout << x << "\n" ;                  (prints int value 10)
}
```

Branching Statements

Branching statements come in different forms:

- simple **if**-statements: **if (**⟨condition⟩**)** ⟨statement⟩, where the ⟨statement⟩ may be compound
- nested **if-else** statements:
  if (⟨condition$_1$⟩**)** ⟨statement$_1$⟩
  else if (⟨condition$_2$⟩**)** ⟨statement$_2$⟩
  ...
  else ⟨statement$_n$⟩
- case-by-case **switch**-statements:
  switch (⟨expression⟩**) {**
  case ⟨constant$_1$⟩**:** ⟨statement$_1$⟩**;**
  ...
  case ⟨constant$_n$⟩**:** ⟨statement$_n$⟩**;**
  default: ⟨last statement⟩**; }**

The statements ⟨statement$_i$⟩ in a **switch**-statement usually terminate with **break;**

If not, control drops through, which may prevent code being duplicated, but is disastrous, if the **break** is forgotten

Loop Statements

Loop statements come in different forms:

- **while**-loops: **while (**⟨condition⟩**)** ⟨statement⟩, where the ⟨statement⟩ may be compound
- **for**-loops: **for (**⟨initialise⟩**;** ⟨condition⟩**;** ⟨change⟩**)** ⟨statement⟩

  Here ⟨initialise⟩ can be any statement that is executed first and only once

  The ⟨condition⟩ is checked in every iteration; if it becomes false, the loop will be terminated

  The ⟨change⟩ statement is executed after each iteration

  Any of ⟨initialise⟩, ⟨condition⟩ and ⟨change⟩ may be omitted, but not the **;**
- **do**-loops: **do** ⟨statement⟩ **while (**⟨condition⟩**);**

The **break** statement can also be used inside loops with the effect that the loop will be terminated

The **continue** statement can be used inside loops with the effect that the current iteration step will be terminated and the next iteration step will be started—note that in a **for**-loop the ⟨change⟩ statement is still executed in case of a **continue**

Miscellaneous

The comma **,** can be used to separate a sequence of expressions, e.g. in initialisations **int** x = 10 **,** y = 4

This may be used also in **for**-loops for the ⟨initialise⟩ statement or the ⟨change⟩ statement, e.g. **for (int** $i = 0$ **,** $j = 0$ **;** $i < 10$ **&&** $j < 0$ **;** $++i$ **,** $++j$ **)** ...

A single semicolon **;** defines the **null statement**

The operator **?** permits to write a **conditional expression** in the form ⟨condition⟩ **?** ⟨result$_1$⟩ **:** ⟨result$_2$⟩

Command lines starting with #⟨keyword⟩ are not control statements, but rather instructions for the C++ preprocessor, e.g. for conditional compilation

The most important keywords for preprocessing are **include** and **define**

1.3 Functions

Function declarations take the form
return_type function_name (type$_1$ identifier$_1$, ..., type$_n$ identifier$_n$)

Function definitions take the form
return_type function_name (type$_1$ identifier$_1$, ..., type$_n$ identifier$_n$)
{ function_body }

Function declarations are usually placed in a separate header file

Functions with short body can be declared to be **inline**; then the function definition should also be in the header file; inline functions are merely an indication (for the compiler) to use the function body for code replacement rather than for implementation of function calls

Functions may be recursive, i.e. contains calls of themselves in the function body

Function bodies may contain declarations, but no declaration or definition of other functions (except in **main()**)

Unless the return type is **void**, the function body must contain a **return** statement, and the returned value must have (or be converted to) the return type

Example

```
int factorial(int n) // calculates the factorial n!
{
  int result = 1 ;
  if (n > 0) {
  do {
         result *= n ;
         --n ;
     } while (n > 1);
  }
  else if (n < 0) { cout << "Error : negative argument = "
                  << n << "in factorial function\n" ; }
  return result;
}
```

We may prefer a recursive definition of a function computing factorials (exercise)

Function Calls

A **function call** simply takes the form

function_name(argument$_1$, . . . , argument$_n$)

Such a function call in C++ uses **call-by-value**, i.e. the arguments are copied and bound to the identifiers in the function declaration

Such a function call can be used like any other exression of the return type

A function declaration may contain arguments that are not used in the function body; then no argument identifier is required, but a function call must still provide an argument

A function declaration may provide **default values** for the identifiers (syntax is the same as for variable declarations with initialisation)

If default values are provided by the function definition, tailing arguments may be omitted in function calls (so the default values are used)

Example 1: Euclidean Algorithm

See the files `gcd.h` and `gcd.cpp`

Function declaration in `gcd.h`:

```
#ifndef gcd_h
#define gcd_h
int euklid(int n, int m);
#endif /* gcd_h */
```

Compile and link instructions in `makefile`:

```
gcd: gcd.cpp gcd.h
     g++ gcd.cpp -o gcd
```

Example 1 / cont.

Function definition in `gcd.cpp`:

```
#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include "gcd.h"
using std::cout;
using std::cin;
int euklid(int n, int m)
{
    if (n == m)
        return n;
    else
        if (n > m)
            return euklid(n - m, m);
    else
        return euklid(m - n, n);
}
```

Example 1 / cont.

Main program in `gcd.cpp`:

```
int main()
{
    int n, m, gcd;
    cout << "Enter two positive integers.\n";
    cin >> n >> m;
    if (n > 0 && m > 0)
    {
        gcd = euklid(n, m);
        cout << "The gcd of " << n << " and " << m << " is " << gcd << ".\n";
        return(EXIT_SUCCESS);
    }
    else
    {
        cout << "Error: Input " << n << " or " << m << " is 0 or negative.\n";
        exit(EXIT_FAILURE);
    }
}
```

Example 2: Russian Multiplication

See the files `russe.h` and `russe.cpp`

Function declaration in `russe.h`:

```
#ifndef russe_h
#define russe_h
int russianmult(int n, int m);
#endif /* russe_h */
```

Compile and link instructions in `makefile`:

```
russe: russe.cpp russe.h
     g++ russe.cpp -o russe
```

Main program in `russe.cpp`

Example 2 / cont.

Function definition in `russe.cpp`:

```
#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include "russe.h"
using std::cout;
using std::cin;
int russianmult(int n, int m)
{   int s = 0;
    do
    {
        if (m % 2 != 0) s += n;
        n *= 2;
        m /= 2;
    }
    while (m != 0);
    return s;
}
```

Miscellaneous

The scope of any variable (including the identifiers in the head) declared inside a function is bound to the function

As a consequence, memory for such local variables is allocated when the function is invoked, and deallocated when control leaves the function

A function may also access variables outside the scope of the function by using the **scope resolution operator ::**, e.g. as in **::x = ...**

Conversely, variables declared inside a function may be made persistent by using the keyword **static**, e.g. in **static double total_sum ;**

Static variables are only initialised once–by 0, if nothing else is specified—and retain their value after control leaves the function

Function names may be overloaded—the number of arguments or the argument types must be different

Function overloading may be useful, if "essentially the same" function is implemented (like a scalar product on vectors of different dimension), but should be avoided for other purposes

1.4 Pointers and Arrays

The address of an identifier is obtained by the **address-of** operator **&**, e.g. **&**x retrieves the address of variable x

The **dereferencing operator** ***** retrieves the value at a given address (as in ***pt**—clearly, we have $^*\&x = x$

A **pointer** holds an address, usually the start address of some stored value—the special **null pointer** (**pt = 0**) does not point to any valid address

Pointers are declared by giving the type of the value stored at the address pointed to, e.g.

- **double *pt_x** declares a pointer with name pt_x, which directs to a location storing a double-precision floating-point number
- **int *pt_y** declares a pointer with name pt_y, which directs to a location storing an integer

As pointers contain addresses, they can be incremented, added and subtracted, e.g. in **++pt_x** or **pt_x += 4**

Pointers may point to pointers, e.g. in the declaration **int **pt** we have a pointer pt that points to a pointer to an integer

Arrays

An **array** is a contiguous sequence of memory locations defined by a base address and a length

A declaration **type identifier[length]** defines an array with name identifier that stores as many values of the given type as indicated by the non-negative integer length, e.g.

- **double x[100]** defines an array of 100 double-precision floating point numbers
- **int *pt[10]** defines an array of 10 pointers to integers, whereas **int (*pt_x)[10]** defines a pointer pt_x to an array of 10 integers

For an array named x the expression x**[i]** retrieves the i'th element of the array ($0 \leq i \leq \text{length} - 1$)—instead of i we can use any expression that evaluates to an integer in the permissible range (we have $x[\text{i}] = *(\&x[0] + i)$)

The name of an array can be validly used as its base address, so an assignment **pt = x** can be used, where pt is a pointer, in particular for a generic pointer defined by **void *pt**

Multi-Dimensional Arrays / Strings

Multi-dimensional arrays are defined and accessed analogously

For instance, **double x[4][7][2]** defines a $4 \times 7 \times 2$ array of doubles, and **x[2][3][1]** will retrieve the double at the address &x $+2 \cdot 7 \cdot 2 + 3 \cdot 2 + 1$

Character strings such as `"this is a string"` are treated as one-dimensional arrays of type **char**

This could be defined by **char message[]** = `"this is a string"`

However, in C++ such character strings always end with a special character '\0', so the array has always at least the size 1—this becomes relevant for string operations

Use **#include ⟨cstring⟩** to make string operations such as **strcpy()** and **strcat()** available

Call by Reference

C++ also supports call-by-reference, which can be done in two ways:

- Using the **reference declarator &** the address of a memory location outside the scope of the function will be passed, as in **swap(int &x, int &y)**

  A function call looks the same as for call-by-value, e.j. **swap(i,j)** with variables i, j of type **int**

- Using a pointer as function argument, as in **void swap(int *pt_x, int *pt_y)**

  Then in a function call addresses must be passed, as in **swap(&i, &j)** with variables i, j of type **int**

As arrays de facto provide pointers, the second option can be used to handle arrays as funvction arguments, e.g. **double sum(double x[], int n)**—the argument **n** need not be the length of the array, and a length argument for the array argument is not permitted

A call can hand over any (one-dimensional) array, as in **sum(height,100)** or in **sum(height[10],38)** assuming a definition **double height[100]** or the like

Examples

```
void swap(int *pt_x, int pt_y)
{  int temp;
   temp = *pt_x;
   *pt_x = *pt_y;
   *pt_y = temp;
}
```

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

```
int main()
{   int i = 10, j = 20;
    swap(&i, &j);
    cout << "i = " << i ;
    cout << "j = " << j ;
    cout << "\n";
    return(EXIT_SUCCESS);
}
```

```
int main()
{   int i = 10, j = 20;
    swap(i, j);
    cout << "i = " << i ;
    cout << "j = " << j ;
    cout << "\n";
    return(EXIT_SUCCESS);
}
```

Multi-Dimensional Arrays and Functions as Arguments

Multi-dimensional arrays as function arguments are handled in the same way

However, all dimensions (except the first one) must be specified, as e.g. in **double trace(double x[] [5])**

We can declare pointer to a function, as e.g. in **double (*g)(int m)** (NOT **double *g(int m)**, which declares a function returning a pointer to a double)

Noter that in this case **g** contains the address of the function, where **g(...)** contains the value produced by the function on the given argument(s) **...**

In the same way pointer arguments to functions can be used in function declarations, e.g. **double sum(double (*g)(int m), int n)** declares a function **sum** that that two arguments, a pointer to a function on integers returning doubles plus an integer, the output of the function **sum** is a double

Dynamic Memory Management

A C++ program has access to **free storage** (aka **heap**), so it can allocate and deallocate memory when needed

For allocation the operators **new** and **new[]** are used, e.g. **double *pt = new double;** allocates space for a new double, and
double *pt_x = new double[100]; allocates space for an array of 100 doubles

In the former case a simultaneous initialisation is possible, e.g.
double *pt = new double(0.459e4);

The **new[]** operator can also be used to allocate space for an array of pointers, as e.g. in **double **pt_y = new double *[100];**

For deallocation the operators **delete** and **delete[]** are used, e.g. **delete pt;** or **delete[] pt_x;**

Dynamic memory management mostly makes sense for large data objects such as arrays or class objects, in particular for those data structures we will study in this course—for the standard types dynamic memory allocation and deallocation can usually be dispensed with