HPC Practical Course

1: Introduction to Unix Shell and C++

V. Akishina, I. Kisel, G. Kozlov, I. Kulakov, M. Pugach, M. Zyzak
Goethe University of Frankfurt am Main
2015

Basic Unix Shell Commands

A Unix shell is a command-line interpreter or shell that provides a traditional user interface for the Unix operating system and for Unix-like systems (Linux, OS X). There are different shell categories, like Bourne shell (sh), Bourne-Again shell (bash), C shell (csh), TENEX C shell (tcsh). Here we consider bash commands, since bash is often the default shell, or can be called from other shells with *bash* command.

pwd — print working (current) directory
cd [dirname] — change directory. '**cd ..**' - one level up
mkdir [dirname] — create (make) a new directory
ls [dirname] — list of files in the directory
mv [fileOrDirName] [dirname] — move a file or a directory to another directory
mv [fileOrDirNameOld] [fileOrDirNameNew] — renames a file or a directory
cp [filenameOld] [filenameNew] — copy a file
cp -r [dirnameOld] [dirnameNew] — copy a directory and its content
rm [filename] — remove a file
rm -r [dirname] — remove a directory and its content
grep [string] [filenames] — search file(s) for lines that match a given pattern
wget [fileurl] [dirname] — download a file from the web to the directory
ssh [user]@[hostname] — secure shell client (remote login program)
scp [-r] [userOld]@[hostnameOld]:[fileOrDirNameOld] [userNew]@[hostnameNew]:[fileOrDirNameNew] — secure copy (remote file copy)

g++ [filenames].cpp [flags] -o [executablename] — compile a C++ source code with the GNU compiler and create an executable
./[executablename] — run the executable file

Shell commands can be collected into script text files, which have the extension '**.sh**'.
Use '**. [bashScriptName].sh**' to execute a bash script.

Data Types in C++

Type	Values	Size (bytes)
bool	*true* and *false*	1
unsigned char	0 … 255	1
signed char	–128 … 127	1
unsigned short int	0 … 65 535	2
short	–32 768 … 32 767	2
unsigned int	0 … 4 294 967 295	4
int	–2 147 483 648 … 2 147 483 647	4
float	3.4e–38 … 3.4e+38	4
double	1.7e–308 … 1.7e+308	8
long double	3.4e–4932 … 3.4e+4932	10 (12,16)

Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

- **Boolean type:** The boolean type, known in C++ as bool, can only represent one of two states, true or false.
- **Character types:** They can represent a single character, such as 'A' or '$'. The most basic type is char, which is a one-byte character. Other types are also provided for wider characters.
- **Numerical integer types:** They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be *signed* or *unsigned*, depending on whether they support negative values or not.
- **Floating-point types:** They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.

http://www.cplusplus.com/doc/tutorial/variables/

Type Casting

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as *type-casting*.

Implicit conversion **short a=200; float b; b = a;**	Does not require any operator, automatically performed when a value is copied to a compatible type. Exists for standard types. For users classes a constructor should be written.

Explicit conversion
b = (type) a;
b = type (a);
b = static_cast <new_type> (a)
	Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to, therefore is not safe, can lead to code that while being syntactically correct can cause runtime errors.

dynamic_cast <new_type> ()	Can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.
static_cast <new_type> ()	Can perform conversions between pointers to related classes. Ensures that at least the classes are compatible if the proper object is converted. The overhead of the type-safety checks of dynamic_cast is avoided. A programmer should ensure the conversion is safe.
reinterpret_cast <new_type> ()	Converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.
const_cast <new_type> ()	Manipulates the constness of an object, either to be set or to be removed.

http://www.cplusplus.com/doc/tutorial/typecasting/

Conditional Operators

A simple C++ statement is each of the individual instructions of a program. They always end with a semicolon (;), and are executed in the same order in which they appear in a program. But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C++ provides flow control statements that serve to specify what has to be done by the program, when, and under which circumstances.

The *if* keyword is used to execute a statement or block, if, and only if, a condition is fulfilled.

The conditional operator if/else:

```
if (condition) {
    statement1;
}
else {
    statement2;
}
```

If the condition is *true*, then *statement1* is executed,
if *false* – *statement2.*
The structure else is optional.

Examples:

```
if (a>0) {
    b = a;
}
else {
    b = 0;
}
```

```
if (a>0)
    b=a;
```

The ternary conditional operator:

```
(condition) ? expression1 : expression2;
```

If condition is *true*, then the result is *expression1*,
if it is *false* – *expression2*.

Examples:

```
b = (a>0) ? a : 0;

b = (a>0) ? a : b;
```

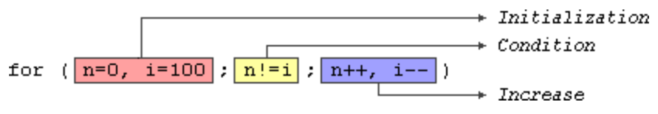
http://www.cplusplus.com/doc/tutorial/control/

The for Loop

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords while, do, and for.

The for loop is designed to iterate a number of times. Its syntax is:

```
for (initialization; condition; increase) statement;
```



- Repeats the statement while condition remains *true*.
- Provides specific locations to contain the initialization statement and the increase statement.
- Specially designed to perform a repetitive action with a counter, which is initialized and increased at each iteration.

Examples:

```
for (int i=0; i<10; i++) {
    a[i] = i;
}
```

```
int i=0;
for (; i<10; i++) {
    a[i] = i;
}
```

```
for ( n=0, i=100 ; n!=i ; n++, i-- ){
    // whatever here...
}
```

http://www.cplusplus.com/doc/tutorial/control/

Pointers

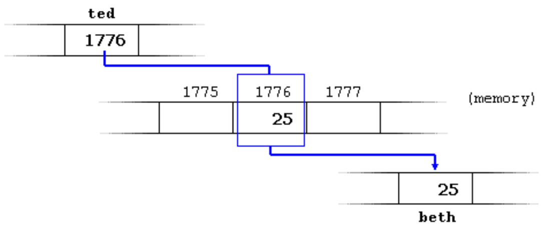
For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address. These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses. This way, each cell can be easily located in the memory by means of its unique address. When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address).

An address can be stored in a pointer variable.

```
*ted = 25;
beth = *ted;
```

Using a pointer we can directly access the value stored in the variable which it points to.

To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to “value pointed by”.



http://www.cplusplus.com/doc/tutorial/pointers/

Pointers: Declaration and Initialization

```
// declaration and initialization
float f;
float fArray[100];
float* a = &f;              // with an address of a float variable
float* a(&f);               // with an address of a float variable
float* b = a;               // with a value of another pointer
float* a = fArray;          // with a name of an array. fArray is also a pointer
float* a = (float*)0xB8000000;  // with an explicit address of a memory
float* a = 0;               // with a zero value
float* a = new float;       // with a memory allocation
```

```
// pointers with constness
float* a;               // a pointer to float
const float* ca;        // a pointer to const float
float* const ac;        // a constant pointer to float
const float* const cac; // a constant pointer to const float
```

http://www.cplusplus.com/doc/tutorial/pointers/

Pointers: Manipulation with Memory

```
// allocation of the memory
float*          myFloatPointer = new float;                        //a single float
float*          myArrayPointer = new float[100];                   //an array of floats
MyClassType*    myClassPointer = new MyClassType(10,20);           //a single object of a class
float*          myFloatAlignedPointer = (float*) _mm_malloc(4,16);//a single float aligned to 16 bytes

// free the memory
delete      myClassPointer;
delete[]    myArrayPointer;
delete      myFloatPointer;
_mm_free(myFloatAlignedPointer);        // only if allocated with _mm_malloc()

// manipulation with pointers
*myFloatPointer = 2.f;                  // manipulation with the value pointed by
*(myArrayPointer+1) *= 2.f;             // manipulation with the second element of the array
myArrayPointer[1] *= 2.f;               // equivalent manipulation with the second element
(*myClassPointer).MyPrintFunction();    // call of the class member function
myClassPointer->MyPrintFunction();      // equivalent call of the function

float myFloat = 10;
*myFloatPointer = myFloat ;             // set the value to the cell pointed by the pointer
myFloatPointer = &myFloat;              // store the address of the float to the pointer
```

http://www.cplusplus.com/doc/tutorial/pointers/

Functions: Call by Value, by Reference or by Pointer

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

```
type name ( parameter1, parameter2, ...) { statements }
```

Where:

- type is the type of the value returned by the function.
- name is the identifier by which the function can be called.
- parameters (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.
- statements is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

Call by value:

the value of a variable is copied to a local variable.

```
int increment(int i) { return i++; }
int increment(int i, const int a) { return i+a; }
```

Call by reference:

faster, since the memory for a local variable is not allocated, the function can modify the value of an input variable.

```
void increment(int &i) { i++; }
void increment(int &i, const int &a) { i+=a; }
```

Call by pointer:

the memory for variable is not allocated, the pointer to variable is copied to a local pointer variable, value of a variable can be modified, the array can be given as an input.

```
void increment(int *i) { (*i)++; }
void increment(int *i, const int *a) { *i += *a; }
void increment(int *array, const int N) {
    for(int i=0; i<N; i++)
        array[i]++;
}
```

http://www.cplusplus.com/doc/tutorial/functions/
http://www.cplusplus.com/doc/tutorial/functions2/

Classes

Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are defined using either keyword class or keyword struct, with the following syntax:

```
class class_name {
access_specifier_1:
    member1;
    function1();
access_specifier_2:
    member2;
    function2();
    ...
} object_names;

class_name::function1() {
    // function1 definition
}
...
```

http://www.cplusplus.com/doc/tutorial/classes/

Example of a Class

```
// class example
#include <iostream>
    using namespace std;

    class CRectangle {
    public:                // the following members can be accessed from outside of the class
        CRectangle(int a, int b): x(a), y(b) {}; // class constructor, equivalent to SetNewValues
        void SetNewValues (int, int);
        int Area () { return (x*y); }
    private:               // the following members can be accessed only by class member functions
        int x, y;
    };

    void CRectangle::SetNewValues (int a, int b) {
        x = a;
        y = b;
    }

    int main () {
        CRectangle rect (1,1);
        cout << "area1: " << rect.Area();
        rect.SetNewValues (3,4);
        cout << "area2: " << rect.Area();
        return 0;
    }
```

Templates

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.
The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

To use this function template we use the following format for the function call:

```
function_name <type> (parameters);
```

We also have the possibility to write class templates, so that a class can have members that use template parameters as types.

http://www.cplusplus.com/doc/tutorial/templates/

Templates: Function Example

```
// function template
#include <iostream>
    using namespace std;

    template <typename T>
    T GetMax (T a, T b) {       // define
        T result;
        if (a>b)    result = a;
        else         result = b;
        return result;
    }

    int main () {
        int i=5, j=6, k;
        double l=10.2, m=5e10, n;
        k=GetMax<int>(i, j);    // use
        n=GetMax<double>(l, m);
        cout << k << endl;
        cout << n << endl;
        return 0;
    }
```

Output:

```
6
5e10
```

Templates: Class Example

```
// class templates
#include <iostream>
    using namespace std;

    template <class T>
    class mypair {          // define the class
        T a, b;
    public:
        mypair (T first, T second)
        {a=first; b=second;}
        T getmax ();
    };

    template <class T>
    T mypair<T>::getmax ()  // define the class member function
    {
        T result;
        if (a>b)  result = a;
        else        result = b;
        return result;
    }

    int main () {
        mypair <int> myobject (100, 75);    // use
        cout << myobject.getmax();
        return 0;
    }
```

Output:

```
100
```

Templates: Example with Several Parameters

Several template parameters can be used.

```
#include <iostream>
    using namespace std;

    template <class T, class T2, int N>
    struct myData {                      // define struct
        T a;
        T2 b[N];
    };

    int main () {
        myData <int,int,10> myStruct1;  // use
        myData <int,float,20> myStruct2;
        myStruct1.b[5] = 5.2;
        myStruct2.b[9] = 9.1;
        cout << myStruct1.b[5] << endl;
        cout << myStruct2.b[9] << endl;
        return 0;
    }
```

Output:

```
5
9.1
```