Summary of new features in C++11

Balázs Varga

1 Automatic object type deduction

Syntax: `auto variable=initializer` and `decltype(expression)`

When using the `auto` keyword as a variable type specifier, the actual type of the variable will be deduced from the initializer. In simultaneous declarations, the deduced types must match. It is possible to combine `auto` with modifiers such as `const` or `&`. Furthermore, C++11 provides a way to obtain the type of a variable using the `decltype` keyword. For functions, `auto` can be used to indicate trailing return type (see below).

In standard C, the `auto` keyword is a storage duration specifier. As of C++11, this semantic is removed.

Examples:

```
auto x; //error: no initializer
auto x=5, y=6; //OK, deduced type is int
auto x=5.0; //OK, deduced type is double
auto x=5, y=6.0; //error: conflicting types

std::multimap<std::string, int> mm;
std::multimap<std::string, int>::iterator it=mm.begin(); //this is tedious
auto it=mm.begin(); //much simpler
decltype(mm.begin()) it=mm.begin(); //this also works with typedef
```

2 Trailing return type

Syntax: `auto function(argument list) -> return type`

C++11 allows function templates to be fully generalized even if the return type depends on the types of the arguments.

Example:

```
template<typename T_a, typename T_b>
auto add(T_a a, T_b b)->decltype(a+b){
    return a+b;
}
```

3 Lambda expressions

Syntax: `[capture clause] (parameters) -> return type {body}`

Lambda expressions are unnamed functions, usually defined inline, at the place of the call. They improve readibility and security by eliminating the need to define a multitude of small functions and function objects.

Lambdas can access (or *capture*) variables from the enclosing scope, either by value, or by reference. The capture clause is a comma-separated list of these variables. Alternatively, `[=]` or `[&]` can be used to capture all variables accessed in the body of the lambda by value or by reference, respectively. The capture list may be empty, but the square brackets cannot be omitted.

The parameter list is the same as in ordinary functions. The parentheses can be omitted if the lambda has no parameters.

The return type can also be omitted, in which case it will be automatically deduced from the return statement.

Example:

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(){
    char str[]="the quick brown fox jumps over the lazy dog";
    int oCount=0;
    for_each(str, str+sizeof(str), [&oCount](char c)->void{
        if(c=='o') oCount++;
    });
    cout<<"The letter \"o\" occurs "<<oCount<<" times."<<endl; //4

    auto lambda=[](int x){return x*2;}; //named closure type
    cout<<lambda(5)<<endl; //10

    return 0;
}
```

4 List initialization

Syntax: `type variable {initializer list};`

C++11 introduces a uniform initialization syntax for arrays, structures, classes and container types. List initialization can be used even for arrays dynamically allocated using `new[]`. Furthermore, functions can take and return initialization lists.

Example:

```
#include <iostream>
#include <string>
#include <vector>
#include <map>

using namespace std;

typedef struct{string name; int age;} person_t;

void f(vector<int> v){
    for(auto e: v) cout<<e<<" ";
    cout<<endl;
}

vector<string> initArray(){
    return {"this","is","returned"};
}

int main(){
    int  a1[5] {1,2,3,4,5}; //works for POD arrays
    int* a2=new int[5] {1,2,3,4,5}; //even if allocated with new
    vector<string> v1 {"hello", "world"}; //container types
    map<string,string> m {{"Bambi","fawn"}, {"Garfield","cat"}, {"ET","alien"}};
    person_t p {"Eric Cartman",8}; //structs and classes
    f({10, 9, 8}); //function arguments
    vector<string> v2=initArray(); //function return values

    for(auto e: v1) cout<<e<<" ";
    cout<<endl;
    for(auto e: v2) cout<<e<<" ";
    cout<<endl;
    for(auto e: m) cout<<e.first<<" is "<<e.second<<endl;
    cout<<p.name<<" is "<<p.age<<" years old."<<endl;
    return 0;
}
```

5 Delegating constructors

Syntax: `ClassName(argument list) : ClassName(different argument list)`

In C++11, constructors can call other constructors of the same class, making it possible to concentrate common initializations in one (*target*) constructor, which results in a more robust, readable and maintainable code. Delegating chains are also allowed – i.e. a delegating constructor can become the target of another delegating constructor. Hovewer, recursion is forbidden: a constructor cannot delegate to itself, neither directly, nor indirectly.

Example:

```
#include <iostream>

using namespace std;

class C{
    public:
        int i;
        C(int a) : i(a) {}
        C(int a, int b): C(a+b) {} //delegates to C(int)
        C(int a, int b, int c): C(a,b*c) {} //delegates to C(int,int)
};

int main(){
    C c1(3), c2(3,4), c3(3,4,5);
    cout<<c1.i<<endl; //3
    cout<<c2.i<<endl; //7
    cout<<c3.i<<endl; //23
    return 0;
}
```

6 Explicitly defaulted and deleted functions

Syntax: `function(argument list)=default;` and `function(argument list)=delete;`

Appending the `=default;` specifier to the end of a function declaration instructs the compiler to generate the default implementation for that function. Naturally, this cannot be done for any function: it must be a special member function, and it cannot have any default arguments. Typical use-cases include the copy constructor and the copy operator. Note that unless the programmer provides their own implementations for these functions, the compiler will always implicitly generate them even without explicit defaulting. However, a defaulted function has a declaration, therefore the programmer has control over its access rules and other special aspects (e.g. it is possible to make a copy constructor protected, or to generate a default virtual destructor).

Similarly, the `=delete;` specifier can be used to disable the usage of a function. This is useful for disabling certain implicit type conversions, and for creating uncopyable classes.

Example:

```
#include <iostream>

using namespace std;

class C{
    public:
        int i;
        C() : i(3) {}
        C(int a) : i(a) {}
        C(double)=delete;
        C(const C&)=delete;
        C& operator=(const C&)=delete;
};

class D{
    public:
        int i;
        D(int a) : i(a) {}
        D()=default;
        D(const D&)=default;
        D& operator=(const D&)=default;
        ~D()=default;
};

int main(){
    C c1;       //OK: default constructor, i=3
    C c2(5);    //OK: constructor with int parameter, i=5
    C c3(5.0); //error: using deleted C(double) constructor
    C c4(c1);  //error: using deleted copy constructor
    c2=c1;      //error: using deleted operator=

    D d1;       //default constructor, i=?
    D d2(8);    //constructor with int parameter i=8
    D d3(d2);   //default copy constructor, i=8
    d3=d2;      //default operator=, i=8

    return 0;
}
```

7 Rvalue references

Syntax: `type&& name`

In C++11, references can bind to rvalues, which is most useful for the purpose of implementing movable classes. It is now possible to overload the copy constructor and the assignment operator so that they can be called on rvalues. The function `std::move()` can be used to cast a variable into an rvalue reference.

An important use-case is in-place sorting algorithms: swapping elements without making deep copies becomes very simple for any type that implements move semantics (i.e. it overloads its copy constructor and assignment operator to take rvalue arguments).

Example:

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

using namespace std;

/* this class implements move semantics */
class Movable{
    Movable(const Movable&); //copy constructor
    Movable(Movable&&); //move constructor
    Movable& operator=(const Movable&); //copy operator
    Movable& operator=(Movable&&); //move operator
};

/* temp is a deep copy of a, but will be discarded */
template<class T>
void swapNaive(T& a, T& b){
    T temp=a;
    a=b;
    b=temp;
}

/* if T is moveable, then this is more efficient */
template<class T>
void swapSmart(T& a, T& b){
    T temp=move(a);
    a=move(b);
    b=move(temp);
}

int main(){
    string a="hello", b="world";
    swapNaive<string>(a,b);
    cout<<a<<" "<<b<<endl; //world hello
    swapSmart<string>(a,b);
    cout<<a<<" "<<b<<endl; //hello world
    return 0;
}
```

8 Multithreading

The C++11 Standard Library provides programmers with multithreading support. Threads can be created from functions and callable objects (instances of classes that overload the `()` operator). The multithreading API implements the usual thread management functions (join, yield, sleep), synchronization tools (mutexes, condition variables, futures and promises), atomic types and thread local variables.

Example:

```
#include <iostream>
#include <thread>
#include <chrono>

using namespace std;

void f1(){
    for(int i=0;i<5;i++){
        cout<<"Hello from f1!"<<endl;
        this_thread::sleep_for(chrono::milliseconds(10));
    }
}

void f2(){
    for(int i=0;i<5;i++){
        cout<<"Hello from f2!"<<endl;
        this_thread::sleep_for(chrono::milliseconds(10));
    }
}

class C{
    public:
        int i;

        C(int x) : i(x) {}

        void operator()(){
            cout<<"Hello from an instance of C! ";
            cout<<"Parameter is "<<i<<"."<<endl;
            this_thread::sleep_for(chrono::milliseconds(10));
        }
};

int main(){
    C c1(3), c2(5);
    thread t1(f1), t2(f2);
    t1.join();
    t2.join();
    thread t3(c1), t4(c2);
    t3.join();
    t4.join();
    cout<<thread::hardware_concurrency()<<endl; //max concurrent threads
    return 0;
}
```

9 Nullptr

C++11 introduces a strongly typed keyword for the null pointer, which is no longer just a macro for the integer literal 0.

Example:

```
#include <iostream>

using namespace std;

void f(int  a, int  b){ cout<<"1 is called."<<endl; }
void f(int* a, int  b){ cout<<"2 is called."<<endl; }

int main(){
  f(0,0); //calls 1, but ambiguous
  f(nullptr,0); //calls 2
  f(nullptr,nullptr); //error: no matching declaration
  return 0;
}
```

10 New smart pointers

C++98 introduced a smart pointer template called `auto_ptr`, in order to make the management of dynamically allocated objects safer. A smart pointer automatically deletes the managed object when the pointer itself is destroyed, thus preventing memory leak. Furthermore, `auto_ptr` takes exclusive ownership of the object: no more than one `auto_ptr` can point to an object at the same time.

In C++11, `auto_ptr` is now deprecated, and is replaced by `unique_ptr`, which has very similar functionality, but its copy constructor is deleted, therefore a simple assignment cannot leave the pointer dangling.

There are two more smart pointers in C++11: `shared_ptr` and `weak_ptr`. As its name suggests, `shared_ptr` does not take exclusive ownership of the managed object – multiple `shared_ptr`s can point to the same destination. A `weak_ptr` holds a non-owning reference to an object that is managed by a `shared_ptr`. The destination of a `weak_ptr` may be deallocated by someone else at any time, and the `weak_ptr` can detect this.

Examples:

```
/* auto_ptr (now deprecated) has exclusive ownership */
auto_ptr<string> ap1(new string("hello"));
cout<<*ap1<<endl; //hello
auto_ptr<string> ap2=ap1; //ownership is transferred
cout<<*ap2<<endl; //hello
cout<<*ap1<<endl; //runtime error (segmentation fault)

/* unique_ptr is like auto_ptr, but more secure */
unique_ptr<string> up1(new string("hello"));
cout<<*up1<<endl; //hello
unique_ptr<string> up2=up1; //compile error (copy operator is deleted)
unique_ptr<string> up3=move(up1); //OK, behaves like auto_ptr
cout<<*up3<<endl; //hello
cout<<*up1<<endl; //runtime error (segmentation fault)

/* shared_ptr does not take exclusive ownership of the object */
shared_ptr<string> sp1(new string("hello"));
shared_ptr<string> sp2=sp1;
cout<<*sp1<<endl; //hello
cout<<*sp2<<endl; //hello

/* weak_ptr does not take ownership */
shared_ptr<string> sp3(new string("hello"));
weak_ptr<string> wp1=sp3;
cout<<wp1.expired()<<endl; //0
cout<<*(wp1.lock())<<endl; //hello
sp3.reset(new string("hello2")); //old target is deleted
cout<<wp1.expired()<<endl; //1
cout<<*(wp1.lock())<<endl; //runtime error (segmentation fault)
weak_ptr<string> wp2=sp3; //points to new data
cout<<*(wp2.lock())<<endl; //hello2
```