Chapter 8

From C To C++

Peter Müller
Globewide Network Academy (GNA)
pmueller@uu-gna.mit.edu

This section presents extensions to the C language which were introduced by C++ [6]. It also deals with object-oriented concepts and their realization.

8.1 Basic Extensions

The following sections present extensions to already introduced concepts of C. Section 8.2 presents object-oriented extensions.

C++ adds a new comment which is introduced by two slashes (//) and which lasts until the end of line. You can use both comment styles, for example to comment out large blocks of code:

```
/* C comment can include // and can span over
   several lines. */
// /* This is the C++ style comment */ until end of line
```

In C you must define variables at the beginning of a block. C++ allows you to define variables and objects at any position in a block. Thus, variables and objects should be defined where they are used.

8.1.1 Data Types

C++ introduces a new data type called *reference*. You can think of them as if they were "aliases" to "real" variables or objects. As an alias cannot exist without its corresponding real part, you cannot define single references. The ampersand (&) is used to define a reference. For example:

```
int ix;          /* ix is "real" variable */
int &rx = ix;    /* rx is "alias" for ix */
```

```
ix = 1;         /* also rx == 1 */
rx = 2;         /* also ix == 2 */
```

References can be used as function arguments and return values. This allows to pass parameters as reference or to return a "handle" to a calculated variable or object.

The table 8.1 is adopted from [1] and provides you with an overview of possible declarations. It is not complete in that it shows not every possible combination and some of them have not been introduced here, because we are not going to use them. However, these are the ones which you will probably use very often.

Declaration	*name* **is ...**	**Example**
type name;	*type*	`int count;`
type name[];	(open) array of *type*	`int count[];`
type name[n];	array with n elements of type *type* (`name[0]`, `name[1]`, ..., `name[`n`-1]`)	`int count[3];`
type *name;	pointer to *type*	`int *count;`
type *name[];	(open) array of pointers to *type*	`int *count[];`
type *(name[]);	(open) array of pointers to *type*	`int *(count[]);`
type (*name)[];	pointer to (open) array of *type*	`int (*count)[];`
type &name;	reference to *type*	`int &count;`
type name();	function returning *type*	`int count();`
type *name();	function returning pointer to *type*	`int *count();`
type *(name());	function returning pointer to *type*	`int *(count());`
type (*name)();	pointer to function returning *type*	`int (*count)();`
type &name();	function returning reference to *type*	`int &count();`

Table 8.1: Declaration expressions.

In C and C++ you can use the modifier `const` to declare particular aspects of a variable (or object) to be constant. The next table 8.2 lists possible combinations and describe their meaning. Subsequently, some examples are presented which demonstrate the use of `const`.

Now let's investigate some examples of contant variables and how to use them. Consider the following declarations (again from [1]):

```
int i;                       // just an ordinary integer
int *ip;                     // uninitialized pointer to
                             // integer
int * const cp = &i;         // constant pointer to integer
const int ci = 7;            // constant integer
const int *cip;              // pointer to constant integer
```

Declaration	*name* **is ...**
const *type* name = *value*;	constant *type*
type * const name = *value*;	constant pointer to *type*
const *type* **name* = *value*;	(variable) pointer to constant *type*
const *type* * const *name* = *value*;	constant pointer to constant *type*

Table 8.2: Constant declaration expresssions.

```
const int * const cicp = &ci; // constant pointer to constant
                              // integer
```

The following assignments are **valid**:

```
i = ci;       // assign constant integer to integer
*cp = ci;     // assign constant integer to variable
              // which is referenced by constant pointer
cip = &ci;    // change pointer to constant integer
cip = cicp;   // set pointer to constant integer to
              // reference variable of constant pointer to
              // constant integer
```

The following assignments are **invalid**:

```
ci = 8;       // cannot change constant integer value
*cip = 7;     // cannot change constant integer referenced
              // by pointer
cp = &ci;     // cannot change value of constant pointer
ip = cip;     // this would allow to change value of
              // constant integer *cip with *ip
```

When used with references some peculiarities must be considered. See the following example program:

```
#include <stdio.h>

int main() {
  const int ci = 1;
  const int &cr = ci;
  int &r = ci;    // create temporary integer for reference
  // cr = 7;      // cannot assign value to constant reference
  r = 3;          // change value of temporary integer
  print("ci == %d, r == %d\n", ci, r);
  return 0;
}
```

When compiled with GNU g++, the compiler issues the following warning:

> conversion from 'const int' to 'int &' discards const

What actually happens is, that the compiler automatically creates a temporay integer variable with value of *ci* to which reference *r* is initialized. Consequently, when changing *r* the value of the temporary integer is changed. This temporary variable lives as long as reference *r*.

Reference *cr* is defined as *read-only* (constant reference). This disables its use on the left side of assignments. You may want to remove the comment in front of the particular line to check out the resulting error message of your compiler.

8.1.2 Functions

C++ allows function overloading as defined in section 6.3. For example, we can define two different functions *max()*, one which returns the maximum of two integers and one which returns the maximum of two strings:

```
#include <stdio.h>

int max(int a, int b) {
  if (a > b) return a;
  return b;
}

char *max(char *a, char * b) {
  if (strcmp(a, b) > 0) return a;
  return b;
}

int main() {
  printf("max(19, 69) = %d\n", max(19, 69));
  printf("max(abc, def) = %s\n", max("abc", "def"));
  return 0;
}
```

The above example program defines these two functions which differ in their parameter list, hence, they define two different functions. The first *printf()* call in function *main()* issues a call to the first version of *max()*, because it takes two integers as its argument. Similarly, the second *printf()* call leads to a call of the second version of *max()*.

References can be used to provide a function with an alias of an actual function call argument. This enables to change the value of the function call argument as it is known from other languages with call-by-reference parameters:

```
void foo(int byValue, int &byReference) {
```

```
    byValue = 42;
    byReference = 42;
  }

  void bar() {
    int ix, jx;

    ix = jx = 1;
    foo(ix, jx);
    /* ix == 1, jx == 42 */
  }
```

8.2 First Object-oriented Extensions

In this section we present how the object-oriented concepts of section 4 are used in C++.

8.2.1 Classes and Objects

C++ allows the declaration and definition of classes. Instances of classes are called *objects*. Recall the drawing program example of section 5 again. There we have developed a class *Point*. In C++ this would look like this:

```
  class Point {
    int _x, _y;       // point coordinates

  public:             // begin interface section
    void setX(const int val);
    void setY(const int val);
    int getX() { return _x; }
    int getY() { return _y; }
  };

  Point apoint;
```

This declares a class *Point* and defines an object *apoint*. You can think of a class definition as a structure definition with functions (or "methods"). Additionally, you can specify the *access rights* in more detail. For example, *_x* and *_y* are **private**, because elements of classes are private as default. Consequently, we explicitly must "switch" the access rights to declare the following to be **public**. We do that by using the keyword `public` followed by a colon: Every element following this keyword are now accessible from outside of the class.

We can switch back to private access rights by starting a private section with `private:`. This is possible as often as needed:

```
  class Foo {
```

```
    // private as default ...

  public:
    // what follows is public until ...

  private:
    // ... here, where we switch back to private ...

  public:
    // ... and back to public.
  };
```

Recall that a structure `struct` is a combination of various data elements which are accessible from the outside. We are now able to express a structure with help of a class, where all elements are declared to be public:

```
class Struct {
public:       // Structure elements are public by default
  // elements, methods
};
```

This is exactly what C++ does with `struct`. Structures are handled like classes. Whereas elements of classes (defined with `class`) are private by default, elements of structures (defined with `struct`) are public. However, we can also use `private:` to switch to a private section in structures.

Let's come back to our class *Point*. Its interface starts with the public section where we define four methods. Two for each coordinate to set and get its value. The set methods are only declared. Their actual functionality is still to be defined. The get methods have a function body: They are defined *within* the class or, in other words, they are *inlined methods*.

This type of method definition is useful for small and simple bodies. It also improve performance, because bodies of inlined methods are "copied" into the code wherever a call to such a method takes place.

On the contrary, calls to the set methods would result in a "real" function call. We define these methods outside of the class declaration. This makes it necessary, to indicate to which class a method definition belongs to. For example, another class might just define a method *setX()* which is quite different from that of *Point*. We must be able to define the *scope* of the definition; we therefore use the scope operator "::":

```
void Point::setX(const int val) {
  _x = val;
}

void Point::setY(const int val) {
  _y = val;
}
```

Here we define method *setX()* (*setY()*) within the scope of class *Point*. The object *apoint* can use these methods to set and get information about itself:

```
Point apoint;

apoint.setX(1);     // Initialization
apoint.setY(1);

//
// x is needed from here, hence, we define it here and
// initialize it to the x-coordinate of apoint
//

int x = apoint.getX();
```

The question arises about how the methods "know" from which object they are invoked. This is done by implicitly passing a pointer to the invoking object to the method. We can access this pointer within the methods as `this`. The definitions of methods *setX()* and *setY()* make use of class members *_x* and *_y*, respectively. If invoked by an object, these members are "automatically" mapped to the correct object. We could use `this` to illustrate what actually happens:

```
void Point::setX(const int val) {
  this->_x = val;   // Use this to reference invoking
                    // object
}

void Point::setY(const int val) {
  this->_y = val;
}
```

Here we explicitly use the pointer `this` to explicitly dereference the invoking object. Fortunately, the compiler automatically "inserts" these dereferences for class members, hence, we really can use the first definitions of *setX()* and *setY()*. However, it sometimes make sense to know that there is a pointer `this` available which indicates the invoking object.

Currently, we need to call the set methods to initialize a point object[1]. However, we would like to initialize the point when we define it. We therefore use special methods called *constructors*.

8.2.2 Constructors

Constructors are methods which are used to initialize an object at its definition time. We extend our class *Point* such that it initializes a point to coordinates (0, 0):

[1]In the following we will drop the word "object" and will speak of "the point".

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

Constructors have the same name of the class (thus they are identified to be constructors). They have no return value. As other methods, they can take arguments. For example, we may want to initialize a point to other coordinates than (0, 0). We therefore define a second constructor taking two integer arguments within the class:

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = x;
    _y = y;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

Constructors are implicitly called when we define objects of their classes:

```
Point apoint;           // Point::Point()
Point bpoint(12, 34);   // Point::Point(const int, const int)
```

With constructors we are able to initialize our objects at definition time as we have requested it in section 2 for our singly linked list. We are now able to define a class *List* where the constructors take care of correctly initializing its objects.

If we want to create a point from another point, hence, copying the properties of one object to a newly created one, we sometimes have to take care of the copy process. For example, consider the class *List* which allocates dynamically memory for its elements. If we want to create a second list which is a copy of the first, we must allocate memory and copy the individual elements. In our class *Point* we therefore add a third constructor which takes care of correctly copying values from one object to the newly created one:

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = x;
    _y = y;
  }
  Point(const Point &from) {
    _x = from._x;
    _y = from._y;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

The third constructor takes a constant reference to an object of class *Point* as an argument and assigns *_x* and *_y* the corresponding values of the provided object.

This type of constructor is so important that it has its own name: *copy constructor*. It is highly recommended that you provide for each of your classes such a constructor, even if it is as simple as in our example. The copy constructor is called in the following cases:

```
Point apoint;            // Point::Point()
Point bpoint(apoint);    // Point::Point(const Point &)
Point cpoint = apoint;   // Point::Point(const Point &)
```

With help of constructors we have fulfilled one of our requirements of implementation of abstract data types: Initialization at definition time. We still need a mechanism which automatically "destroys" an object when it gets invalid (for example, because of leaving its scope). Therefore, classes can define *destructors*.

8.2.3 Destructors

Consider a class *List*. Elements of the list are dynamically appended and removed. The constructor helps us in creating an initial empty list. However, when we leave the scope of the definition of a list object, we must ensure that the allocated memory is released. We therefore define a special method called *destructor* which is called once for each object at its destruction time:

```
void foo() {
  List alist;     // List::List() initializes to
                  // empty list.
  ...             // add/remove elements
}                 // Destructor call!
```

Destruction of objects take place when the object leaves its scope of definition or is explicitly destroyed. The latter happens, when we dynamically allocate an object and release it when it is no longer needed.

Destructors are declared similar to constructors. Thus, they also use the name prefixed by a tilde (˜) of the defining class:

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = xval;
    _y = yval;
  }
  Point(const Point &from) {
    _x = from._x;
    _y = from._y;
  }

  ~Point() { /* Nothing to do! */ }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

Destructors take no arguments. It is even invalid to define one, because destructors are implicitly called at destruction time: You have no chance to specify actual arguments.