CS450/550 — C++ Tutorial — Spring 1995

1 Non-Object-Oriented Features of C++

1.1 Parameter Passing

In C, the only parameter passing mode is "pass by value". A copy of each argument is passed to the called function, and the resulting value is copied when it is returned to the caller.

In C++, there are two other modes: "pass by reference" and "pass by constant reference".

```
void swap(int & i, int & j)
{
  int temp=i;
  i = j;
  j = temp;
}
```

This function will interchange two integer values. For example, we might invoke it as

```
int A[10];          // 10-element array of integers
swap(A[2],A[5]);  // interchange elements 2 and 5.
```

This is implemented by passing addresses, but the code is much cleaner looking. To accomplish the same thing in C, you would have to write:

```
void swap(int * i, int * j)
{
  int temp=*i;
  *i = *j;
  *j = temp;
}

int A[10];            // 10-element array of integers
swap(&A[2],&A[5]);  // interchange elements 2 and 5.
```

One can also pass by "constant reference". This is used to avoid copying large data types:

```
int arraysum (const int & A[], const int len)
{
  int sum=0;

  for (int i=0; i<len; i++) sum := A[i];
  return sum;
}
```

The `const` keyword indicates that we will not be changing `A` or `len`. The compiler will enforce this by flagging any attempts to assign a value to elements of `A` or to change the value of `len`. (However, you can circumvent this protection by using the `&` operator or doing other nasty things.)

1.2 Variables may be declared anywhere that a statement is permitted.

Note, for example, the `for` loop above. We declared the variable `i` there. The lexical scope of `i` will continue from that point until the end of the block (i.e., the closing } of `arraysum`). (Actually, in the pending ANSI standard for C++, the lexical scope of `i` will be just inside the `for` loop. Hence, you should not write code that assumes the `i` will be available outside the loop.)

Here is another example:

```
{
  int i;
  scanf("%d", &i);
  int j;
  scanf("%d", &j);
}
```

1.3 Reference Types

In addition to passing arguments by reference, one can also create variables that are references to other variables. For example:

```
int a=0;
int & b = a;

b = 4;  // the value of a is now 4
```

A reference is an alias—an alternative name—for a storage location. In this case, `b` is an alias for the integer `a`. They are the same memory location, so changing `b` changes `a`. References must be initialized—once created, they cannot be changed to name a different memory location.

1.4 Functions may have optional arguments

Optional arguments must appear after all required arguments. They are indicated by supplying a default value. In this increment function, `inc`, the argument `amount` is optional. If it is not specified, it defaults to the value 1.

```
int inc(int & v, const int amount=1)
{
  v += amount;
  return v;
}


void main()
{
   int a=10;

   inc(a);   // a is now 11
   inc(a,5); // a is now 16
}
```

1.5 Heap Memory Allocation

There is no longer any need to use `malloc` and `free`. Built-in operators, `new` and `delete` are used instead.

```
int *A = new int[20];   // allocate an array of 20 integers on the heap
point_t * pp = new point_t;  // allocate a point on the heap

// when we are done with them, we can deallocate them:

delete [] A;   // the [] must be used when deallocating an array
delete pp;     // otherwise, we just use the delete keyword.
```

2 Object-Oriented Aspects of C++

2.1 Classes

A class is a set of data values and associated functions. The data values are called *data members*, the functions are called *member functions*. The class declaration is usually placed in a "header file" with the suffix `.h`, while the definition of each of the member functions is given in a file with the suffix `.c`. The header file can be viewed as an interface specification. It provides all of the information about the argument types and return types of the member functions. Using it, the compiler can successful compile all calls to these member functions. Any function using the class defined in the header file should include the header file using `#include`.

For example, consider the two files `point.h` and `point.c`.

```
// ************* point.h **************
class point_t
{
 public:
   double givx();
   double givy();
   void setx(double);
   void sety(double);

 protected:
   double x;
   double y;
} ;

// ************* point.c **************

double point_t::givx()
{
  return x;
}
```

```
void point_t::setx(double vx)
{
  x = vx;
}
```

This declares a class named `point_t`. This class has two data fields, `x` and `y`. It has four member functions. `givx` and `givy` can be called to return the values of `x` and `y`. `setx` and `sety` can be called to set the values of `x` and `y`. The data fields are `protected`, which means that functions defined outside the class may not access those fields directly, but only by using the member functions. (Functions in classes that are subclasses of this class are permitted direct access. See below.) The member functions are `public`, which means that any function can invoke these member functions.

The items in the header file merely declare the argument types and return types of each member function. In the `.c` file, the actually functions are declared.

In C++, the full name of a function has the format $\langle classname \rangle$`::`$\langle functionname \rangle$. So the function `point_t::givx` is the full name of the `givx` member function. Because these are member functions, they are permitted to use the names of the protected fields, `x` and `y`. Hence, the *scope* of these field names is restricted to the bodies of all member functions of the class.

Here is an example of how member functions are invoked:

```
point_t p;   // create an instance of the class point_t.

p.setx(1.0); // set its x field to 1.0, p is called the receiver
p.sety(2.0); // set its y field to 2.0
if (p.givx()+p.givy() > 0.0) { ... }
```

A member function is invoked by the following syntax:

$\langle expression\ that\ evaluates\ to\ an\ instance\ of\ the\ class \rangle$`.`$\langle member\ function\ name \rangle$
`(` $\langle argument \rangle$ `,` $\langle argument \rangle$ `...).`

The member function name may be just the "last name" (e.g., `setx`, or it may be a fully-qualified name `point_t::setx`.

2.2 Constructors

C++ provides a very powerful way of initializing instances of each class. Whenever an instance of a class is created, memory is allocated and then a special *constructor* function is called to initialize this memory. Constructor functions always have the name $\langle classname \rangle$`::`$\langle classname \rangle$.

```
// in point.h

class point_t
{
  point_t(double,double);
  point_t (& point_t);
};
```

```
// in point.c

point_t::point_t(double xv=0.0, double yv=0.0)
{
  x=xv;
  y=yv;
}
```

Notice that a constructor does not return a value. You can not even declare them to be `void`.

2.3 Invoking Constructors

The syntax for creating a variable is changed to permit arguments to be passed to the constructor. For example, the following lines all create new `point_t` instances:

```
point_t p1;          // calls point_t::point_t(0.0,0.0)
point_t p2(5.0);     // calls point_t::point_t(5.0,0.0)
point_t p3(5.0,2.0); // calls point_t::point_t(5.0,2.0)
```

Constructors can also be invoked within ordinary expressions. For example, suppose a function `distance()` takes two `point_t`'s as arguments. We could write:

```
d = distance(point_t(1.0,1.0), point_t(5.0,5.0));
```

This would create two temporary points, pass them to the `distance` function, and then delete those temporary points.

2.4 Multiple Constructors and the Copy Constructor

Many constructors can be supplied with each class as long as their arguments differ in number or type. In this case, a second constructor is defined that initializes one point from another point:

```
point_t::point_t(point_t & old)
{
  x = old.x;
  y = old.y;
}
```

This constructor is called the "copy constructor", and it is invoked by the compiler whenever it is necessary to make a copy of an instance of the `point_t` class. For example, if a `point_t` is passed by value as an argument in a function call, then this constructor will be called to initialize the copy that is created. Similarly, if a function returns a `point_t`, this constructor will be called to initialize the returned copy.

Constructors are permitted to use a special syntax called an initialization form. Here are the previous definitions re-expressed using initialization forms:

```
// in point.c

point_t::point_t(double xv=0.0, double yv=0.0): x(xv), y(yv)
{}
```

```
point_t::point_t(point_t & old): x(old.x), y(old.y)
{}
```

Initialization forms are the *only* way to initialize data members that are references to other items.

2.5 Subclassing and Inheritance

New classes can be defined by taking an existing class and adding new member functions and data members (or by overriding existing members).

```
class point3_t: public point_t
{
  public:
    point3_t(double,double,double);

    double givz();
    void setz(double);

  protected:
    double z;
}
```

This defines a new class. It has all of the same members as the class `point_t`. In addition, it has a new data member, `z`, and three new member functions (a constructor, `givz`, and `setz`). The following all works:

```
point3_t  p(1.0,2.0,3.0);

p3.givx();     // evaluates to 1.0
p3.setx(4.0);  // p3 is now (4.0,2.0,3.0)
p3.setz(0.0);  // p3 is now (4.0,2.0,0.0)
```

The code for the constructor is written as follows:

```
point3_t::point3_t(double a, double b, double c): point_t(a,b), z(c)
{}
```

This shows that in addition to using the names of data members as initialization forms, you can also use the names of ancestor classes. This invokes the `point_t::point_t` constructor to initialize the `x` and `y` fields. Then the `z` field is initialized. The actual order in which initialization forms appear is irrelevant. All calls to ancestor classes are performed first (with the most distant ancestors initialized before their children). Then the data member initializations are performed.

2.6 Destructors

C++ also permits user code to be executed when an instance of a class is about to be destroyed. This allows the code to clean up. For example, suppose we had a class to represent a linked list of integers:

```
class link
{
 public:
   link(int, link *);
   ~link();

   int a;
   link * next;   // pointer to the next element in the list, or 0
}

link::link(int i, link * ptr): a(i), next(ptr)
{}

link::~link()
{
  if (next) delete next;
}

void main()
{
  // create a linked list of three links 1, 2, 3, allocated on the heap
  link * head = new link(1, new link(2, new link(3,0)));
  // real stuff skipped
  delete head;
}
```

The member function `link::~link` is called the *destructor*.

When the head of the linked list is to be deleted, the destructor is called. It (recursively) deletes the `link` that it points to. This recursion is terminated with the last link in the linked list. The effect is to free up all of the links in the linked list, even when only the head of the linked list is explicitly deleted.

2.7 Overloading and Operator Overloading

In C++, many functions may have the same name as long as their signatures (i.e., the number and data types of their arguments) differ. We have already seen this with constructors.

In addition, the builtin operators (such as `+ * / = [] () ->`) all have function names, so that they can be overloaded as well. Here is an example of permitting vector addition of two points:

```
class point_t
{
  public:
    point_t operator + (const point_t &) const;
    point_t & operator += (const point_t &);
}

point_t point_t::operator + (const point_t & p2) const
{
```

```
  return (point_t(x+p2.x,y+p2.y));
}

point_t & point_t::operator += (const point_t &p2)
{
  x += p2.x;
  y += p2.y;
  return *this;
}
```

The name of the + function is `operator +`. Hence, if we write

```
point_t a(1,1), b(2,2), c;

c = a+b;
```

then the `point_t::operator +` function will be called, and `c.x` will be 3, and so will `c.y`.

In the class definition and function header, the symbol `const` at the end of the line indicates that this member function does not cause any change to the `point_t` instance.

The function `operator +=` *does* change the current `point_t` instance. It also returns a reference to the current instance. To do this, it uses the reserved symbol `this`, which is always a pointer to the current instance. The expression `* this` is therefore the instance itself, which is returned (without being copied).

2.8 Virtual Function Inheritance

Consider the following code fragments:

```
class prim_t
{
  public:
    prim_t * givnext();
    virtual void draw( scene_t &);
  protected:
    prim_t * next;  // pointer to next instance, to support linked list
};

class line_t: public prim_t
{
  public:
    void draw(scene_t &);
};

class circle_t: public prim_t
{
  public:
    void draw(scene_t &);
};
```

```
class scene_t
{
  public:
    void draw();
  private:
    prim_t * head;   // head of linked list
};

void scene_t::draw()
{
  for(prim_t * ptr=head; ptr; ptr = ptr→givnext())
  {
    ptr→draw(*this);
  }
}
```

The class `prim_t` defines a so-called `virtual` function named `draw`. Each actual primitive class (such as `line_t` and `circle_t`) redefines this member function so that it knows how to draw instances of each class. Suppose we create a scene named `sc` with a linked list of primitives that contains some circles and some lines. When we invoke `sc.draw()`, it loops through the primitives and invokes the `draw` function for each of them. Because `draw` is a virtual function, each instance contains a pointer (called the `vptr`) that indicates which class it belongs to. So when `ptr` points to a `circle_t`, then the `circle_t::draw()` function is invoked. But when `ptr` points to a `line_t`, the `line_t::draw()` function is invoked.

This is a kind of run-time type disambiguation. It permits very nice polymorphism. The `scene_t::draw` routine can draw each object without knowing what each one is.