The C++ Programming Language

Pointers to Member Functions

Outline

Pointers to Functions

- Pointers to functions are a surprisingly useful and frequently underutilized feature of C and C++.
- Pointers to functions provide an efficient and effective form of subprogram generality
  - *e.g.*, the qsort standard C library function:

```
qsort (void *, int, int, int (*)(void *,void *));
static int asc_cmp (void *i, void *j) {
      return *(int *)i - *(int *)j;
}
static int dsc_cmp (void *i, void *j) {
      return *(int *)j - *(int *)i;
}
void print (int a[], int size) {
      for (int i = 0; i < size; i++)
            printf ("%d", a[i]);
      putchar ('\n');
}
void main (void) {
      int a[] = { 9, 1, 7, 4, 5, 8, 3, 1, 2, 0};
      int size = sizeof a / sizeof *a;
      print (a, size);
      qsort (a, size, sizeof *a, asc_cmp);
      print (a, size);
      qsort (a, size, sizeof *a, dsc_cmp);
      print (a, size);
}
```


Pointers to Member Functions

- Pointers to member functions provide an implementation-independent way of declaring and using pointers to class member functions.
  - Note, this works with **virtual** and non-**virtual** functions!
- Earlier C++ versions required tricking the C++ type system into utilizing the internal non-member function representation to achieve pointer to member function semantics, *e.g.*,

```
struct X { void f (int); int i, j; };
typedef void (*PTF) (...); // Bad style.

void f (void) {
      PTF fake = (PTF) &X::f; // Assume a lot!
      X a;
      (*fake)(&a, 2); // Fake the call...
}
```

- This approach is clearly inelegant and error-prone.
  - and doesn't work at all if **f** is a virtual function!

The Type of a Class Member

- A pointer to a function cannot be assigned the address of a member function even when the return type and signature of the two match exactly:

```
class Screen {
private:
      short height, width;
      char *screen, *cur_pos;
public:
      Screen (int = 8, int = 40, char = ' ');
      ~Screen (void);
      int get_height (void) { return height; }
      int get_width (void) { return width; }
      Screen &forward (void);
      Screen &up (void);
      Screen &down (void);
      Screen &home (void);
      Screen &bottom (void);
      Screen &display (void);
      Screen &copy (Screen &);
      // ...
};

int height_is (void) { /* ...*/ }
int width_is (void) { /* ...*/ }
int (*ptfi)(void);
ptfi = &height_is; // OK
ptfi = &width_is; // OK
ptfi = &Screen::get_height; // Error
ptfi = &Screen::get_width; // Error
```

Declaring a Pointer to Member Function

- A member function has an additional type attribute absent from a non-member function, namely: "its class." A pointer to a member function must match exactly in three areas:
  - The data types and number of its formal arguments.
    - *i.e.*, the function's signature.
  - The function's return data type.
  - The class type of which the function is a member.
- The declaration of a pointer to a class member function is similar to a regular pointer to a function.
  - However, it also requires an expanded syntax that takes the class type into account.

Pointer to Class Member Function

- As mentioned above, a pointer to member function is defined by specifying its return type, its signature, and its class.
- Therefore,
  - A pointer to the `Screen` member functions are defined for `Screen::get_height ()` and `Screen::get_width ()` as:

    ```
    int (Screen::*)(void);
    ```
  - That is, a pointer to a member function of `class Screen` taking no arguments and returning a value of type **int**, *e.g.*,

    ```
    int (Screen::*pmf1)(void) = 0;
    int (Screen::*pmf2)(void) = &Screen::get_height;

    pmf1 = pmf2;
    pmf2 = &Screen::get_width;
    ```


Pointers to static Class Member Functions

- Note that **static** class member functions behave differently that non-**static** member functions *wrt* pointers-to-member functions.
  - *i.e.*, **static** class member functions behave like regularnon-member functions.
  - *e.g.*,

    ```
    class Foo {
    public:
        static int si (void);
        int nsi (void);
    };
    int (*ptsfi) (void);
    int (Foo::*ptnsfi) (void);

    ptsfi = &Foo::si; // ok
    ptsfi = &Foo::nsi; // Error
    ptnsfi = &Foo::si; // Error
    ptnsfi = &Foo::nsi; // ok
    ```


Using typedef to Enhance Readability

- Use of a typedef can make the pointer to member function syntax easier to read.
- For example, the following **typedef** defines ACTION to be an alternative name for:

  ```
  Screen &(Screen::*)(void);
  ```
- That is, a pointer to a member function of class Screen taking no arguments and returning a reference to a class Screen object, *e.g.*,

  ```
  typedef Screen &(Screen::*ACTION)(void);
  ACTION default = &Screen::home;
  ACTION next = &Screen::forward;
  ```

Function Arguments

- Pointers to members may be declared as arguments to functions, in addition, a default initializer may also be specified:

```
typedef Screen &(Screen::*ACTION)(void);

Screen my_screen;
ACTION default = &Screen::home;

Screen& foo (Screen&, ACTION = &Screen::display);

void ff (void)
{
     foo (my_screen); // pass &Screen::display
     foo (my_screen, default);
     foo (my_screen, &Screen::bottom);
}
```


Using a Pointer to Class Member Function

- Pointers to class members must always be accessed through a specific class objects.

- This is accomplished by using `.*` and `->*`, the two pointer-to-member selection operators, *e.g.*,

```
Screen my_screen, *buf_screen = &my_screen;
int (Screen::*pmfi)(void) = &Screen::get_height;
Screen &(Screen::*pmfs)(Screen &) = &Screen::copy;

/* ...*/

// Direct invocation of member functions
if (my_screen.get_height () == buf_screen->get_height ())
     buf_screen->copy (my_screen);

// Pointer to member equivalent
if ((my_screen.*pmfi) () == (buf_screen->*pmfi)())
     (buf_screen->*pmfs)(my_screen);
```


Using a Pointer to Class Member Function (cont'd)

- A declaration wishing to provide default arguments for member function `repeat ()` might look as follows:

```
class Screen
{
public:
     Screen &repeat (ACTION = &Screen::forward,
                     int = 1);
     /* ...*/
};
```

- An invocation of repeat might look as follows:

```
Screen my_screen;

/* ...*/

my_screen.repeat (); // repeat (&Screen::forward, 1);
my_screen.repeat (&Screen::down, 20);
```


Using a Pointer to Class Member Function (cont'd)

- A non-general implementation of a repeat function, that performs some user-specified operation *n* times could be done the following way:

```
enum Operation { UP, DOWN, /* ...*/ };
Screen &Screen::repeat (Operation op, int times)
{
     switch (op)
     {
     case DOWN: /* code to iterate n times */;
          break;
     case UP: /* code to iterate n times */;
          break;
     }
     return *this;
}
```

- Pointers to member functions allow a more general implementation:

```
typedef Screen &(Screen::*ACTION)(void);

Screen &Screen::repeat (ACTION op, int times)
{
     for (int i = 0; i < times; i++)
          (this->*op) ();
     return *this;
}
```

Example Usage (cont'd)

- A table of pointers to class members can also be defined. In the following example, `menu` is a table of pointers to `class Screen` member functions that provide for cursor movement:

```
ACTION menu[] =
{
    &Screen::home;
    &Screen::forward;
    &Screen::back;
    &Screen::up;
    &Screen::down;
    &Screen::bottom;
};
enum Cursor_Movements
{
    HOME, FORWARD, BACK, UP, DOWN, BOTTOM
};

Screen &Screen::move (Cursor_Movements cm)
{
    (this->*menu[cm])();
    return *this;
}
```


Difference between PTMF and PTF

- *e.g.*,

```
#include <stream.h>

class Base_1 {
public:
    void a1 (int);
    static void a2 (int); // Note static...
};

// Pointer to function type
typedef void (*F_PTR)(int);

// Pointer to Base_1 member function type
typedef void (Base_1::*MF_PTR)(int);

void a3 (int i); // Forward decl.

class Base_2 {
public:
    void b1 (MF_PTR);
    void b2 (F_PTR);
};
```


Difference between PTMF and PTF (cont'd)

- *e.g.*,

```
void Base_1::a1 (int i) {
    cout << "Base_1::a1 got " << i << "\n";
}

void Base_1::a2 (int i) {
    cout << "Base_1::a2 got " << i << "\n";
}

void a3 (int i) {
    cout << "a3 got " << i << "\n";
}

// Define tw objects.
Base_1 base_1;
Base_2 base_2;

void Base_2::b1 (MF_PTR fp) {
    /* Note object...*/
    (base_1.*fp)(3);
}

void Base_2::b2 (F_PTR fp) { (*fp)(5); }
```


Difference between PTMF and PTF (cont'd)

- main program

```
int main (void) {
    cout << "base_2.b1 (base_1.a1);\n";
    base_2.b1 (base_1.a1);
    // Base_1::a1 got 3

    cout << "\nbase_2.b2 (a3);\n";
    base_2.b2 (a3);
    // a3 got 5

    cout << "\nbase_2.b2 (base_1.a2);\n";
    base_2.b2 (base_1.a2);
    // Base_1::a2 got 5

    cout << "\nbase_2.b2 (Base_1::a2);\n";
    base_2.b2 (Base_1::a2);
    // Base_1::a2 got 5

    return 0;
}
```

Pointer to Class Data Member

- In addition to pointers to member functions, C++ also allows pointers to data members.
  - Pointers to class data members serve a similar purpose to the use of the ANSI C `offsetof` macro for accessing structure fields.
- The syntax is as follows:
  - The complete type of Screen::height is "short member of class Screen."
  - Consequently, the complete type of a pointer to Screen::height is "pointer to short member of class Screen." This is written as:

    short Screen::*
- A definition of a pointer to a member of class Screen of type short looks like this:

```
short Screen::*ps_Screen;
short Screen::*ps_Screen = &Screen::height;

ps_Screen = &Screen::width;
```


Using a Pointer to Data Member

- Pointers to data members are accessed in a manner similar to that use for pointer to class member functions, using the operators `.*` and `->*`, *e.g.*,

```
typedef short Screen::*PS_SCREEN;

Screen my_screen;
Screen *tmp_screen = new Screen (10, 10);

void ff (void)
{
    PS_SCREEN ph = &Screen::height;
    PS_SCREEN pw = &Screen::width;
    tmp_screen->*ph = my_screen.*ph;
    tmp_screen->*pw = my_screen.*pw;
}
```

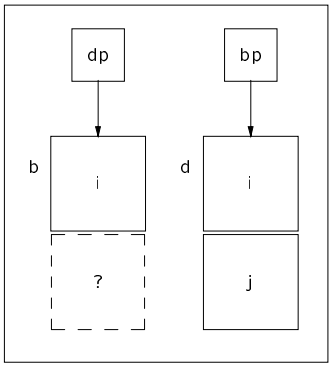
- Note: since height and width are *private* members of Screen, the initialization of `ph` and `pw` within `ff ()` is legal only if `ff ()` is declared a friend to Screen!

Contravariance

- Just as with data members, we must be careful about *contravariance* with pointers to member functions as well.
- *e.g.*,

```
struct Base {
    int i;
    virtual int foo (void) { return i; }
};
struct Derived : public Base {
    int j;
    virtual int foo (void) { return j; }
};
void foo (void) {
    Base b;
    Derived d;
    int (Base::*ptmfb) (void) = &Base::foo; // "ok"
    int i = (b.*ptmfb) ();
    // trouble!
    ptmfb = (int (Base::*) (void)) &derived::foo;
    int j = (b.*ptmfb) ();
    // Tries to access non-existant j part of b!
}
```


Contravariance (cont'd)



- Problem: what happens (b.*ptmfg) () is called?