CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 10 – References & Copy-Constructor Overloading & Default Arguments

Passing objects as arguments

- Whenever possible, always pass an object to a function as a **const** reference
- If the object has to be modified, then pass a simple (non-const) reference
- Passing an object by value will include the overhead of constructor call and copy of contents
- Pointers are only helpful if you want the possibility of an optional object argument (since the pointer can be null)

```
void process_event ( const Event& e, Event* newevent=0 );
```

Don’t do this!

```
//: C11:PassingBigStructures.cpp
struct Big {
  char buf[1000];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) // Passing Big by value will copy 1000 chars to
{                 // the new local Big b object!! Use a reference!
  b.i = 100;      // Do something to the argument
  return b;
}

int main() {
  B2 = bigfun(B);
}
```

Copy Constructor

- When you pass an object by value it calls the copy constructor
- If you don't make a copy-constructor, the compiler will create one for you

```
class Point
{
  private:
    int x, y;
  public:
    Point(int x1, int y1) { x = x1; y = y1; }
    Point(const Point &p1) {x = p1.x; y = p1.y; } // Copy constructor
    int getX() {  return x; }
    int getY() {  return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here
    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
    return 0;
}
```

Preventing pass-by-value

- **“How do you know that an object will never be passed by value?”**
  - You can declare a private copy-constructor. You don’t even need to create a definition (unless one of your member functions or a friend function needs to perform a pass-by-value.)

Preventing pass-by-value

```
//: C11:NoCopyConstruction.cpp - Preventing copy-construction

class NoCC {
  int i;
  NoCC(const NoCC&); // No definition ok (just declaration)
public:
  NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
  NoCC n;
  f(n);          // 1) Error: copy-constructor called
  NoCC n2 = n;   // 2) Error: c-c called
  NoCC n3(n);    // 3) Error: c-c called
}
```

Overloading functions

- Overloading allows the definition of functions with the same name, but with different arguments.
  - Example:

```
void print(char);
void print(float);
```

  - OpenGL is a C interface; therefore, cannot use overloading.
    - API 1.1 of OpenGL:

```
glVertex2f ( float, float );
glVertex2d ( double, double );
glVertex3d ( double, double, double );
```

Overloading on return values

- Not possible!
  - C++ allows ignoring a return value, making it difficult to overload on a returned value.
- So, you cannot do this:

```
void f();
int f();    → will generate error (f already exists)
```

Function Overloading: Updating Stash

- Very useful in constructors!

```
//: C07:Stash3.h - Function overloading
class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage
spaces
  int next;      // Next empty space
  // Dynamically allocated array of
bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size); // Zero quantity
  Stash(int size, int initQuantity);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
```

Function Overloading: Updating Stash

```
//: C07:Stash3.h - Function overloading

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}
```

Wake-up!

- https://youtu.be/-hVCk4GSJOQ

Default Arguments

- Sometimes default arguments can be used to reduce overloading:

```
// The 2 constructors in the Stash class:

     Stash(int size); // Zero quantity is used here
     Stash(int size, int initQuantity);

// Can be reduced to one:

     Stash(int size, int initQuantity = 0);

// These definitions now use the same constructor:

     Stash A(100), B(100, 0);
```

Default Arguments

- Rules:
  - only the last arguments can have default values
  - when an argument has a default value, all the next ones will also need to have default values
  - default arguments only appear in the declaration (not in the definition/implementation of the method/function)

```
// f.h:
void f (int x, int y=0, float f=1.1);
//or even:
void f (int x, int = 0, float = 1.1);

// f.cpp (no default values):
void f (int x, int y, float f) {  ...  }
```

Overloading vs. default arguments

- It is a design choice
  - Efficiency considerations:
    - if you start using if statements to test contents of default values, it may be a better design to split into several overloaded functions
  - Code maintenance considerations:
    - avoid implementing the same initialization code twice

Overloading vs. default arguments

```
//: C07:Mem.h
typedef unsigned char byte;
class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz); // could become a default value in
               // the previous constructor
 ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
};
```

Example of techniques

```
class GsVec2
 { public :
    union { struct{float x, y;};
            float e[2];
          };
   public :
    static const GsVec2 null;     //!< (0,0) null vector
    static const GsVec2 one;      //!< (1,1) vector
    static const GsVec2 minusone; //!< (-1,-1) vector
    static const GsVec2 i;        //!< (1,0) vector
    static const GsVec2 j;        //!< (0,1) vector
   public :

    /*! Initializes GsVec2 as a null vector. Implemented inline. */
    GsVec2 () : x(0), y(0) {}

    /*! Copy constructor. Implemented inline. */
    GsVec2 ( const GsVec2& v ) : x(v.x), y(v.y) {}

    /*! Initializes with the two given float coordinates. Implemented inline. */
    GsVec2 ( float a, float b ) : x(a), y(b) {}

    /*! Initializes with the two given int coordinates converted to floats. Implemented inline. */
    GsVec2 ( int a, int b ) : x(float(a)), y(float(b)) {}

    /*! Initializes with one int and one float. Implemented inline. */
    GsVec2 ( int a, float b ) : x(float(a)), y(b) {}
```

allow access to coordinates
by member or by float array syntax

multiple overloaded constructors
to reduce type-casting
when instantiating GsVec2

Example of techniques

```
/*! Initializes with one float and one int. Implemented inline. */
GsVec2 ( float a, int b ) : x(a), y(float(b)) {}

/*! Initializes with the two given double coordinates converted to floats. Implemented inline. */
GsVec2 ( double a, double b ) : x(float(a)), y(float(b)) {}

/*! Initializes from a float pointer. Implemented inline. */
GsVec2 ( const float* p ) : x(p[0]), y(p[1]) {}

/*! Set coordinates from the given vector. Implemented inline. */
void set ( const GsVec2& v ) { x=v.x; y=v.y; }

/*! Set coordinates from the two given float values. Implemented inline. */
void set ( float a, float b ) { x=a; y=b; }

/*! Set coordinates from the two given int values. Implemented inline. */
void set ( int a, int b ) { x=float(a); y=float(b); }

/*! Set coordinates from the two given double values. Implemented inline. */
void set ( double a, double b ) { x=float(a); y=float(b); }

…etc
```

multiple overloaded set methods to reduce type-casting