# Classes

* Built-ins used to make complex types
* Passed by copy … the entire structure
* Syntax of “.” And “->”.

// Use the "primitives" or "built-ins" to make your own types:

struct Point { // This is just a template ... not an actual instance

int x; // No initialization here ... not an instance

int y;

};

// 2: Passed a copy just like "int". Does not change the caller.

void setToOrigin(Point p) {

p.x = 0;

p.y = 0;

}

// 3: The function does something to a given "point".

void setToOrigin2(Point\* p) {

p->x = 0; // Like the "." but for pointers

p->y = 0;

}

void main() {

int a;

a=2;

Point p; // Make one ... like "int a".

Point q = p; // BIG copy. Memory copy.

p.x = 4; // The "." is the membership operator

p.y = 5;

setToOrigin(p);

cout << p.x << "," << p.y << endl;

system("pause");

}

* Building types from types
* Drilling in through the “.”

// Use the "primitives" or "built-ins" to make your own types:

struct Point { // This is just a template ... not an actual instance

int x; // No initialization here ... not an instance

int y;

};

struct Line {

Point a; // How big is a Line structure?

Point b;

};

void main() {

Line lineOne;

Line\* p = &lineOne;

lineOne.a.x = 4;

p->b.y = 2;

Point\* pa = &lineOne.a;

}

* Set of functions to go with a structure, each with a pointer

struct Point {

int x;

int y;

int accessCount;

};

void print(Point\* p) {

++p->accessCount;

cout << "Point("<<p->x<<","<<p->y<<") accessed "<<p->accessCount<<" times."<<endl;

}

void initialize(Point\* p, int nx, int ny) {

p->x = nx;

p->y = ny;

p->accessCount = 1;

}

int getX(Point\* p) {

++p->accessCount;

return p->x;

}

void setX(Point\* p, int nx) {

++p->accessCount;

p->x = nx;

}

void main() {

Point p;

initialize(&p,2,3);

int a = getX(&p);

print(&p);

system("pause");

}

These functions are in code space. There is just one copy. There are multiple structures running around. The caller tells which structure to work on.

* This pattern came out of C. Very common. Compiler allows you to specify which methods go with the structure.
* Move the function inside. The “this->” is always first argument. Can use it … easier to drop it.
* These functions inside a structure with a default (hidden) this are called “methods”.

struct Point {

int x;

int y;

int accessCount;

void print(/\*Point\* this\*/) { // And drop the "this->"

++accessCount;

cout << "Point(" << this->x << "," << this->y << ") accessed "

<< accessCount << " times." << endl;

}

};

p.print();

Remember. The code is in code space. It is NOT in the object. When you create 20 Points you only have one copy of code used for all. Just a compiler trick.

* Special “initialization” method name

struct Point {

int x;

int y;

int accessCount;

Point(int nx, int ny) { // Special method here ... no return and same name

x = nx;

y = ny;

accessCount = 1;

}

Point() { // Can have multiple with different arguments. This is the "no-arg" constructor

x=0;y=0;accessCount=1;

}

Point q(); // Looks like a function

Point p(2,3);

int a=p.getX();

q.print();

p.print();

* Code users can still access my “x” and “y”. I want to hide the details from them so I can change the details later if needed. Maybe I am going to a database. Maybe I am going to cache a length so it doesn’t have to be calculated everytime.
* Want people to use my methods … my “interface”.

struct Point {

private: // Only methods in this structure definition can get to these

int x;

int y;

int accessCount;

public: // Backwards compatible with C ... public by default

Point(int nx, int ny)

* Discourage public things. Need a replacement word for “struct” that is private by default.

class Point {

// Private by default

int x;

int y;

int accessCount;

public:

Point(int nx, int ny)

* When you say “int a = p.x” you are letting the compiler write the code. When you say “int a=p.getX()” then YOU get control over what happens.
* Your implementation can change without affecting callers. You can prevent things like “p.x=4”.
* You can do range checking and other complex multi-step processes that the user doesn’t need to know about. A long descriptive function name tells more about what is going on.

**Program to the interface … not the implementation.**

* Inline (default behavior when you code in header)
* Just a suggestion.
* Destructors and the default constructor/destructor

~Point() {

cout << "I am done now" << endl;

}

void someFunction() {

Point q(1,2);

}

void main() {

Point p(2,3);

someFunction();

Point\* q = new Point(2,3);

delete q;

system("pause");

// At the end of this function p goes away. Compiler puts in code to call

// the destructor "~Point".

}

* If you do not provide a constructor you get the default no-args which does nothing.
* If you DO provide ANY constructor the compiler will not make a no-args.
* If you do not provide a destructor you get a do-nothing.

# Inheritance

* Extending a class to ADD TO it
* Derived and base.

I need to add a Z coordinate to my Point. Duplicate code? Need a way to say this new type is all of an existing type plus some new things.

class Point3D

{

// Need some magic to say start with Point instead of start with nothing

int z;

public:

Point3D(int nx, int ny, int nz) {

}

};

class Point3D : public Point // Show first without public

{

// Need some magic to say start with Point instead of start with nothing

int z;

public:

Point3D(int nx, int ny, int nz) : Point(nx,ny), z(nz) // alternate for z

{

z = nz;

}

int getZ() {

++accessCount; // PROTECTED

return z;

}

// Override. Method has the same name. Compiler knows which one based on

// the variable type.

void print() {

Point::print(); // Tells which

cout << "Oh. And Z=" << z << endl;

}

};

# IS-A and HAS-A

Base is drawn at top in UML diagrams. Upcast/downcast

void doPrinting(Point p) {

cout << "I have a point here: " << endl;

p.print();

}

void doPrinting3D(Point3D\* p) {

cout << "I have a 3D point here: " << endl;

p->print();

}

Point one(1,2);

doPrinting(one);

Point3D two(1,2,3);

doPrinting(two); // Up cast. Derived to base. Works every time

// Point3D is-a Point ... plus more. It can always be treated lesserly.

Point a = two; // Copy only the data for x and y. Are methods copied?

//doPrinting3D(&one);

doPrinting3D((Point3D\*)&one); // Force the compiler to downcast

// Now change the doPrinting to take a reference.

// No help. The compiler still uses the variable

// type to pick the functions. Need a way to let

// the compiler look at the object in memory and

// decide what it is is.

// Next time

Point.h

#ifndef POINT\_H\_ // "POINT\_H\_" is up to you. Pick something unique.

#define POINT\_H\_

class Point {

// Private by default

int x;

int y;

public: // Public interface for people to use

Point(); // "No-args" constructor

Point(int nx, int ny); // Constructor with arguments

~Point(); // Destructor

// Could (should) make these "inline".

int getX();

int getY();

void print();

}; // Don't forget the semicolon

#endif

Point.cpp

#include "Point.h"

#include <iostream>

using namespace std;

Point::Point() : x(0), y(0) {

//x=0; // Initializer list is mandatory for initing base class.

//y=0; // Shown here for demonstration.

}

Point::Point(int nx, int ny) {

x = nx;

y = ny;

}

Point::~Point() {

}

int Point::getX() {

return x;

}

int Point::getY() {

return y;

}

void Point::print() {

cout << "Point " << x << "," << y << endl;

}

Point3D.h

#ifndef POINT3D\_H\_

#define POINT3D\_H\_

#include "Point.h"

class Point3D : public Point { // Public inheritance

int z;

public:

Point3D(int nx, int ny, int nz);

inline int getZ(); // "inline" added here for demonstration

void print();

};

#endif

Point3D.cpp

#include "Point3D.h"

#include <iostream>

using namespace std;

Point3D::Point3D(int nx, int ny, int nz) : Point(nx,ny), z(nz)

{

//z = nz; // Another option for z

}

int Point3D::getZ() {

return z;

}

void Point3D::print() {

// x and y are private in the base class or we could have

// used them directly here. Better to program to the

// interface anyway.

cout << "Point " << getX() << "," << getY() << "," << z << endl;

}

Main.cpp

#include <iostream>

using namespace std;

#include "Point.h"

#include "Point3D.h"

void main() {

Point a; // For no-args constructor, leave off ()

Point b(1,2);

Point\* c = new Point(); // Here you need the () for no args constructor

Point\* d = new Point(1,2);

delete c; // Don't forget to later delete what you "new"

delete d;

Point3D two(5,6,7);

a.print();

two.print();

a = two; // Down-cast is no problem.

a.print(); // Forgotten the fact it is 3D.

system("pause");

}