c Programming

Dr.Weerakaset Suanpaga
(D.Eng)

Department of Civil Engineering
Faculty of Engineering , Kasetsart University
Bangkok, Thailand

https://pirun.ku.ac.th/~fengwks/ram/ge749/lect/lect1.pdf

1.00 Lecture 37

A Brief Look at C++:
A Guide to Reading C++ Programs

C(++) programs without classes

```
// File main_temp.C: Main program
#include <iostream.h>                  // Standard header file(C)
#include “celsius.h”         // User header file

int main( ) {                          // Can ignore main args
   for (double temp= 10.0; temp <= 30.1; temp+=10.0)
        cout << celsius(temp) << endl;    // System.out.println
   cout << “ End of list” << endl;        // C uses printf()
   return 0;}                             // Java System.exit(0)

// File celsius.C: File with function code
double celsius(double fahrenheit)         // Call by value
{return (5.0/9.0*(fahrenheit - 32.0));}

// File celsius.h: Header file with function prototype
double celsius(double fahrenheit);

// No relationship between file, method or class names in C++
// Distribute .h and .o files but not .C files to your users
```

Call by reference example

```
#include <iostream.h>
void triple( int& base);             // & means send reference
                                     // Above usually in .h file
int main() {
   int value;
   cout << “Enter an integer: “;
   cin >> value;
   triple( value);
   cout << “ Tripled value is: “ << value << endl;
   return 0;}

void triple(int& base) {             // Must match prototype
   base *= 3;}

// In C++ you can choose call by reference or value for most
// primitives or objects, unlike Java where primitives are
// “by value” and objects are “reference by value”
// Call by value occurs when no &s are used, same as Java
```

Call with pointer example

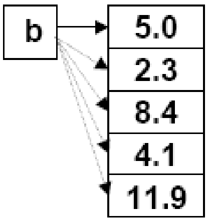
```
#include <iostream.h>
void triple( int* base);          // * means pointer (address)
                                  // Reference is constant ptr
int main() {                      // Ptr is variable ref, can
   int value;                     // point to anything of type
   cout << “Enter an integer: “;
   cin >> value;
   triple( &value);               // Must send address (ptr)
   cout << “ Tripled value is: “ << value << endl;
   return 0;}

void triple(int* base) {          // Must match prototype
   *base *= 3;}                   // * means dereference, or
                                  // use the value, not the
                                  // address

// This is old-fashioned (C doesn’t support call by reference)
// Technically, the pointer is passed by value (“fake ref”)
```

Pointer arithmetic

```
#include <iostream.h>
double average1(double b[], int n);     // Function prototypes
double average2(double b[], int n);

int main() {
  double values[]= {5.0, 2.3, 8.4, 4.1, 11.9};
  int size= 5;             // C++ arrays don’t know their size
  double avg1= average1(values, size);
  double avg2= average2(values, size);
  cout << "Averages are: " << avg1 << " and " << avg2 << endl;
  return 0;}
```

Pointer arithmetic, p.2

```
double average1(double b[], int n) {
   double sum= 0;
   for (int i= 0; i < n; i++)
        sum += b[i];
   return sum/n; }            // Should check n> 0

double average2(double b[], int n) {  // double* b ok too
   double sum= 0;
   for (int i= 0; i < n; i++)
        sum += *(b + i);
   return sum/n; }
```



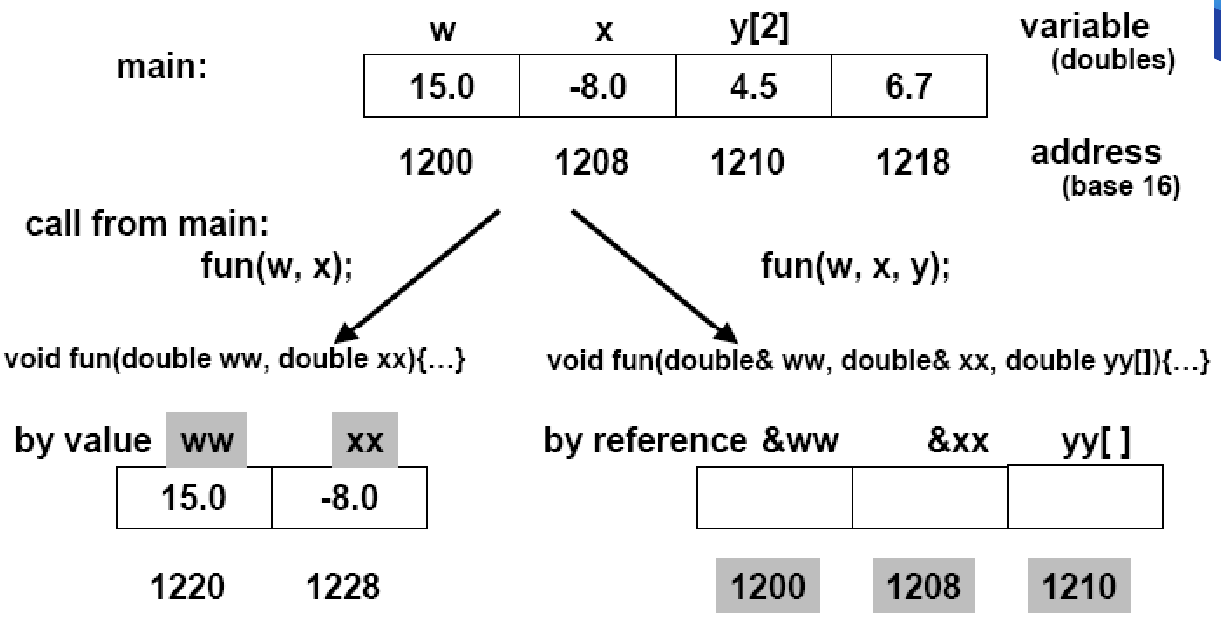
Bracket Program (C, C++)

```
#include <math.h>    // Obsolete, use #include <cmath>
#define FACTOR 1.6   // Obsolete, use const double FACTOR=1.6;
#define NTRY 50      // Use const int NTRY= 50;

int zbrac(float (*func)(float), float *x1, float *x2) {
   void nrerror(char error_text[]);        // Function prototype
   int j;               // Can define vars at top or on the fly
   float f1, f2;
   if (*x1 == *x2) nrerror(“Bad initial range”);
   f1= (*func)(*x1);
   f2= (*func)(*x2);
   for (j=1; j <=NTRY; j++) {
        if (f1*f2 < 0.0) return 1;      // false/true are 0, non-0
        if (fabs(f1) < fabs(f2))
                f1= (*func)(*x1 += FACTOR*(*x1-*x2));
        else
                f2= (*func)(*x2 += FACTOR*(*x2-*x1));
   }
   return 0;
}
```

Main() for bracket

```
#include <math.h>          // Obsolete, use #include <cmath>
#include <iostream.h>      // Obsolete, use #include <iostream>
using namespace std;

// Either place here or in .h file which is #included here:
int zbrac(float (*func)(float), float *x1, float *x2);
float f(float x);

int main() {
   float n1= 7.0;
   float n2= 8.0;
   int bracketFound= zbrac(f, &n1, &n2);
   cout << "Lower " << n1 << " upper " << n2 << endl;
   return 0;    }

void nrerror(char error_test[]) { cout << error_test << endl; }

float f(float x) {  return x*x -2.0;}
```

Passing arguments: value, reference



	w	x	y[2]		
main:	15.0	-8.0	4.5	6.7	variable (doubles)
	1200	1208	1210	1218	address (base 16)

call from main:

fun(w, x);

fun(w, x, y);

void fun(double ww, double xx){…}

void fun(double& ww, double& xx, double yy[]){…}

by value	ww	xx
	15.0	-8.0
	1220	1228

by reference	&ww	&xx	yy[ ]
	1200	1208	1210

Point Class

```
// File Point.h
#include <iostream>
using namespace std;

class Point{
   public:
       Point (double a=0, double b=0);   // Default constructor
       ~Point(){};                        // Destructor
       double getX();                     // Prototype only
       double getY();
       void print();
       Point addPoint (Point& g);         // Choose value or ref
   private:
       double x, y;
} ;
```

Point Class, cont.

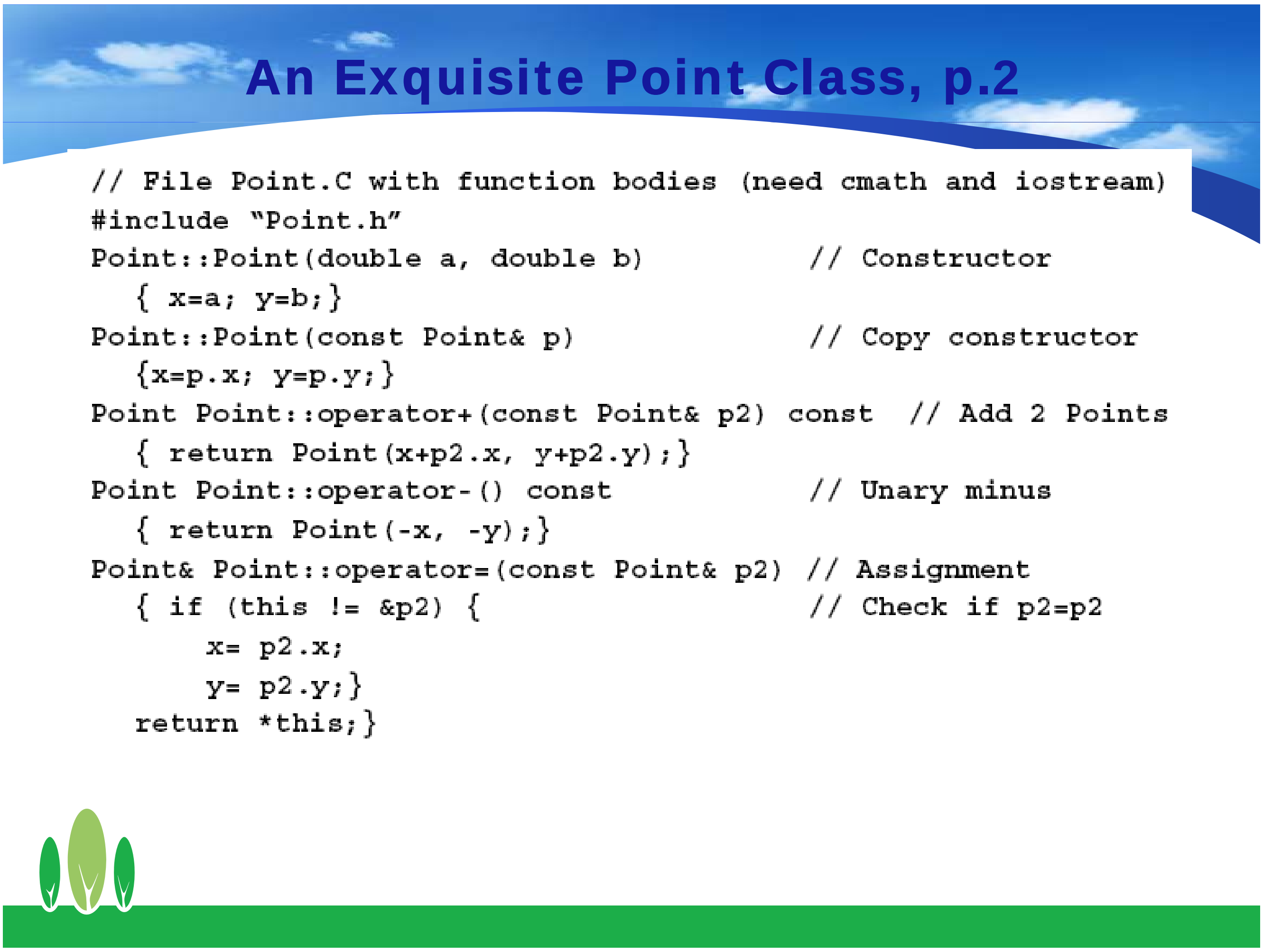
```
// File Point.C
#include “Point.h”
Point::Point(double a, double b) {x=a; y=b;}
double Point::getX() {return x;}
double Point::getY() {return y;}
void Point::print(){cout << ‘(‘ << x << ‘,’ << y <<‘)’<< endl;}
Point Point::addPoint(Point& g)          // Pass by ref
{
       Point h;                        // Don’t use new unless dynamic
       h.x= x + g.x;                   // x,y are this point’s x
       h.y= y + g.y;                   // g’s x,y are added to us
       return h;
}
// Scope resolution operator :: indicates function is member
// of Point class. Without it, it has no access to private data
```

Point Program Example

```
// File pointDemo.C
#include <iostream>          // New style header (could use old)
using namespace std;
#include “Point.h”

int main(){
   double a;
   Point p1(3,4), p2(2), p3;        // Constructor(no ‘new’ reqd)
   p3= p1.addPoint(p2);             // Add point1 and point2
   p3.print();
   a= p3.getX();
   cout << “x: “ << a << endl;
   return 0;}
```

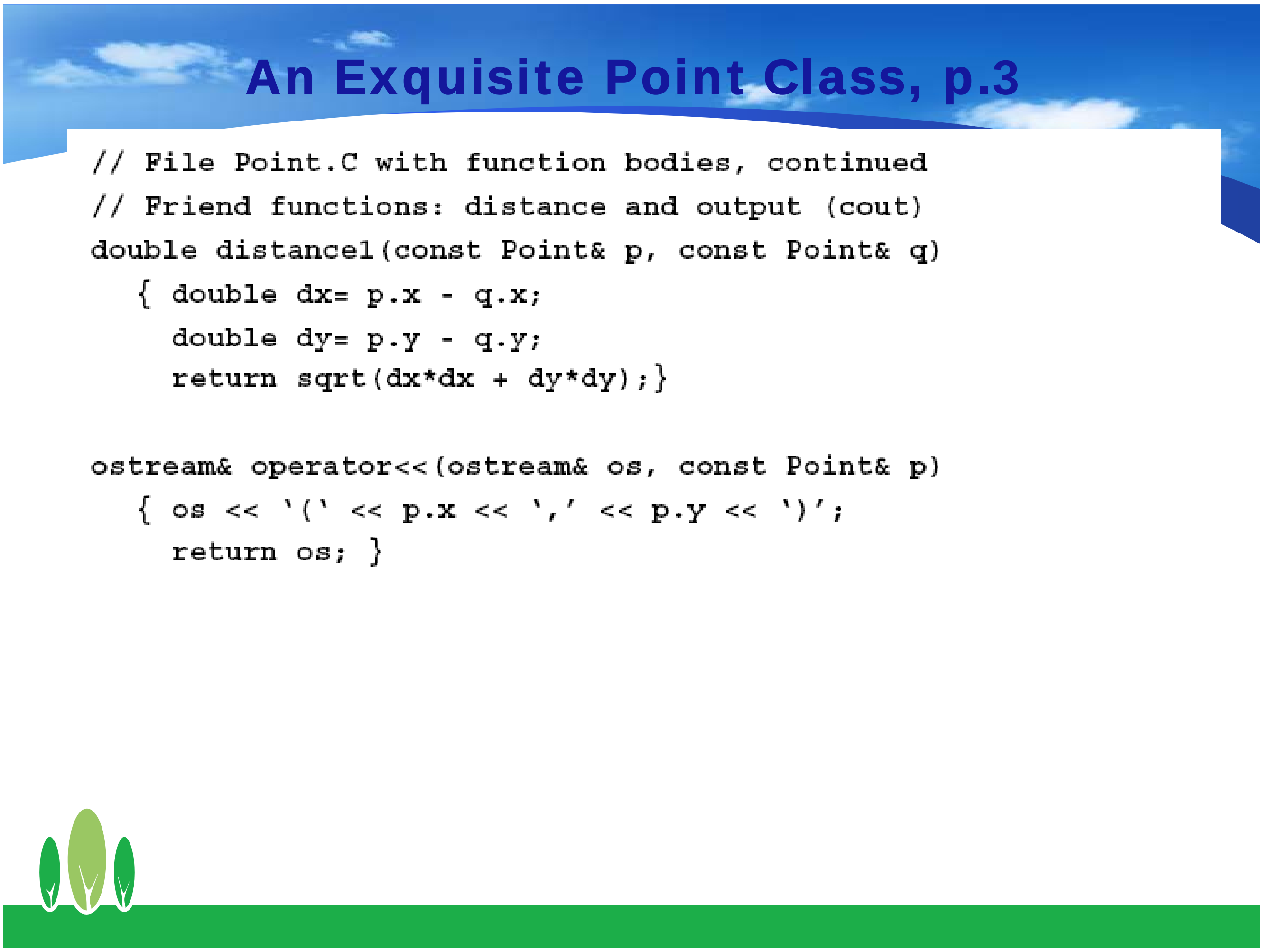
An Exquisite Point Class

```
// File Point.h
#include <iostream>
#include <cmath>
using namespace std;

class Point{
   friend double distance1(const Point& p, const Point& q);
   friend ostream& operator<<(ostream& os, const Point& p);
public:
   Point(double a=0.0, double b=0.0);    // Constructor
   Point(const Point& p);                // Copy constructor
   Point operator+(const Point& p) const;  // Add 2 Points
   Point operator-() const;        // Unary minus
   Point& operator=(const Point& p);     // Assignment
   ~Point() {};                          // Destructor
private:
   double x, y;
};
```



An Exquisite Point Class, p.2

```
// File Point.C with function bodies (need cmath and iostream)
#include “Point.h”
Point::Point(double a, double b)         // Constructor
   { x=a; y=b;}
Point::Point(const Point& p)             // Copy constructor
   {x=p.x; y=p.y;}
Point Point::operator+(const Point& p2) const  // Add 2 Points
   { return Point(x+p2.x, y+p2.y);}
Point Point::operator-() const           // Unary minus
   { return Point(-x, -y);}
Point& Point::operator=(const Point& p2) // Assignment
   { if (this != &p2) {                  // Check if p2=p2
       x= p2.x;
       y= p2.y;}
   return *this;}
```



An Exquisite Point Class, p.3

```
// File Point.C with function bodies, continued
// Friend functions: distance and output (cout)
double distance1(const Point& p, const Point& q)
   { double dx= p.x - q.x;
     double dy= p.y - q.y;
     return sqrt(dx*dx + dy*dy);}

ostream& operator<<(ostream& os, const Point& p)
   { os << ‘(‘ << p.x << ‘,’ << p.y << ‘)’;
     return os; }
```

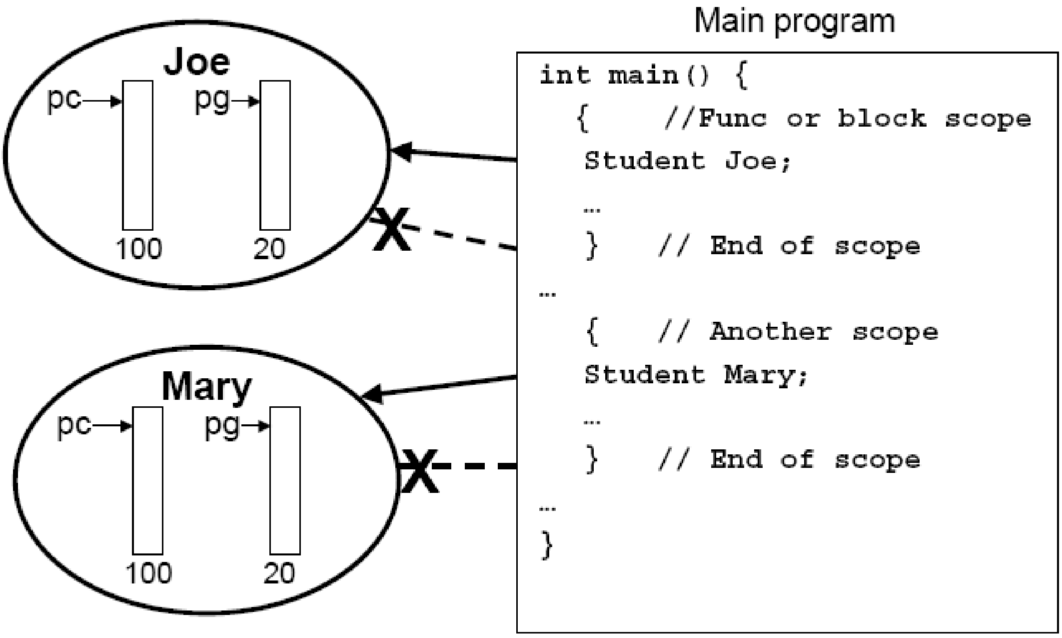
Using the Point Class

```
#include <iostream>
using namespace std;
#include “Point.h”

int main(){
   Point p1(3,4), p2(2);    // Use constructor, default args
   Point p3(p2);            // Use copy constructor
   Point p4= Point(1,2);   // Assignment operator(member copy)
   p3= p1 + p2;             // Same as p3= p1.operator+(p2)
   p4= p1 + p2 + p3;        // Chaining
   p3= -p1;                 // Unary minus
   p2= p4 + p1;             // We could implement subtraction!
   double a;
   a= distance1(p1, p2);
   cout << “The distance from p1 to p2 is: “ << a << endl;
   cout << “p1= “ << p1 << “ and p2= “ << p2 << endl;
   return 0; }
   // Our Java matrix methods would be nicer with op overload!
```

Constructors and Destructors
Dynamic memory allocation

```
Class Student
  public:
    Student(…)
      { pc= new courselist[ncourses];
        pg= new gpalist[nterm];}
    ~Student()
      { delete[] pc;
        delete[] pg; }
```

Constructors and Destructors, p.2



Main program

```
int main() {
  {     //Func or block scope
   Student Joe;
   …
   }   // End of scope
…
   {   // Another scope
   Student Mary;
   …
   }   // End of scope
…
}
```

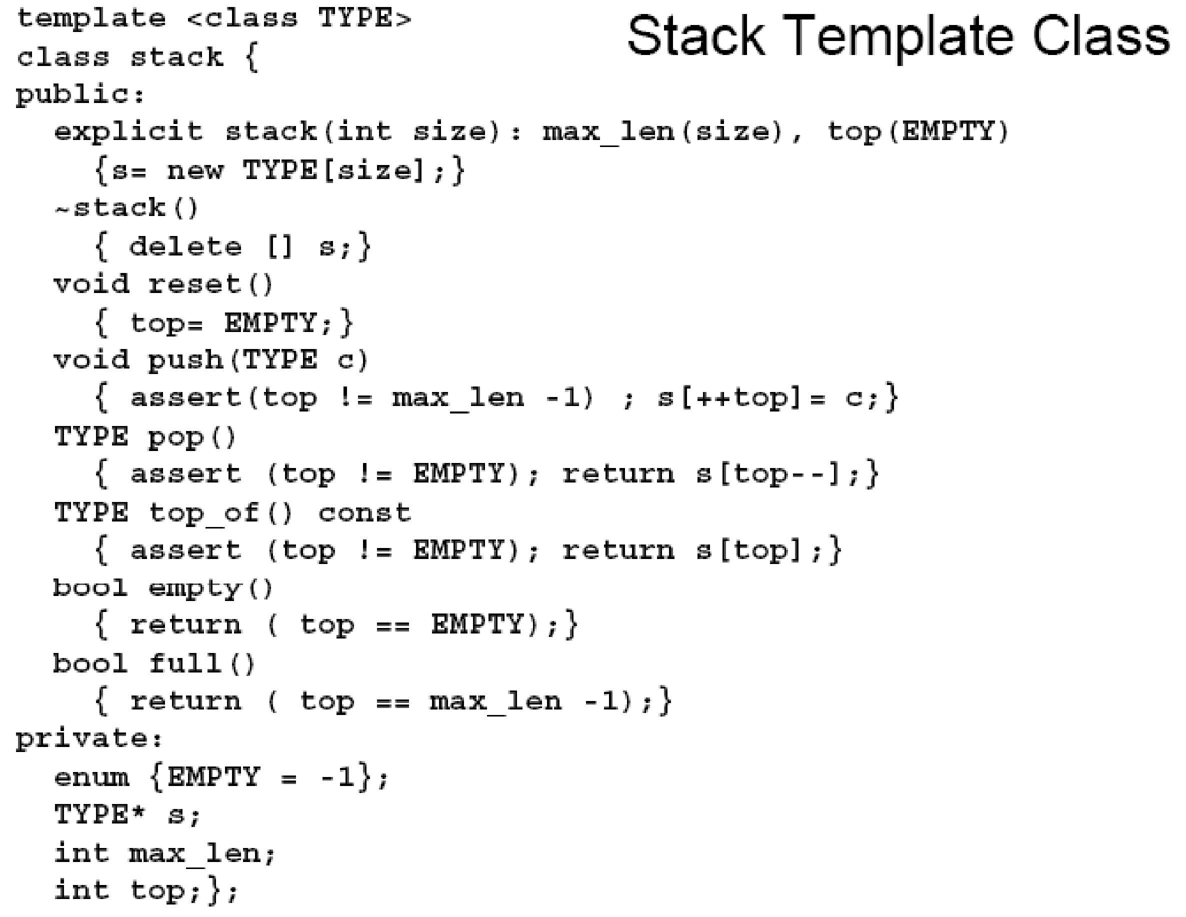
Constructors and Destructors, p.3

- No memory management in main program or functions, as a goal in C++
  - In C, memory was managed for each variable
    - You had to remember to allocate it and free it when done, no matter where these events occurred.
    - Dynamic memory errors are over 50% of C program errors
  - In C++, we build memory management into classes
    - ‘New’ only in constructors; ‘delete’ only in destructors
    - Application developer sees nearly automatic garbage collection. She “never” uses new; creates new objects just by defining them: Student Joe, same as int i
    - Class developer has control of garbage collection when needed
  - C++ garbage collection is tough on lists, trees, etc.

Memory Management Errors

- Deleting same memory twice is a bug
  - Hard to find!
  - Trick: Comment out all deletes if you have a weird bug in a program with dynamic memory
    - If bug goes away, you're deleting some memory twice
- Not deleting memory when done is a leak
  - Leaking 100 bytes per call on a Web server with a million calls a day means buying GBs of memory and crashing regularly anyway

---



Stack Template Class

```
template <class TYPE>
class stack {
public:
  explicit stack(int size): max_len(size), top(EMPTY)
    {s= new TYPE[size];}
  ~stack()
    { delete [] s;}
  void reset()
    { top= EMPTY;}
  void push(TYPE c)
    { assert(top != max_len -1) ; s[++top]= c;}
  TYPE pop()
    { assert (top != EMPTY); return s[top--];}
  TYPE top_of() const
    { assert (top != EMPTY); return s[top];}
  bool empty()
    { return ( top == EMPTY);}
  bool full()
    { return ( top == max_len -1);}
private:
  enum {EMPTY = -1};
  TYPE* s;
  int max_len;
  int top;};
```

Main Program, Stack Template

```
int main(){                       // Must #include iostream
  int size;
  char book;
  cout << "Enter size of stack: ";
  cin >> size;
  stack<char> library(size);

  if (!library.full())
    library.push(’a’);
  if (!library.full())
    library.push(’x’);
  if (!library.empty()){
      book= library.pop();
      cout << ”First shelved: " << book << endl;}
  if (!library.empty()){
      book= library.pop();
      cout << "Earlier book, later shelved: " << book << endl;}
  if (!library.full())
    library.push(’g’);
  book= library.top_of();
  cout << "Next book returned: " << book << endl;}
```

Inheritance: Access Specifiers

```
class Base : {
public: …
protected: …
private: …};

class Derived : AccessSpecifier Base {
public:…
protected:…
private:… };
```

Base Member Type	AccessSpecifier		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Inaccessible	Inaccessible	Inaccessible

Most restrictive specifier holds. Almost always use public inheritance

Friendship is not inherited and is not transitive.

Research project revisited

```
class Student {
public:
   // Array of Students in RProject needs default constructor
   Student() {firstName= " "; lastName= " ";}
   Student( const string fName, const string lName):
        firstName(fName), lastName(lName) {}

   // GetData function is virtual: mandatory interface,
   // default implementation for derived classes
   virtual void GetData() const
   { cout << firstName << " " << lastName << " ";}
   virtual ~Student {}              // Destructor
private:
   string firstName, lastName;
};

// If we add:   virtual double GetPay() = 0;
// it makes Student abstract; requires each derived class to
// implement GetPay()
// We can also define const methods, like final Java methods
```

Undergrad

```
class Undergrad : public Student {
public:
   Undergrad(string fName, string lName, double hours,
   double rate);            // Body in different file
   double GetPay() const
        { return UnderWage * UnderHours;}
   virtual void GetData() const
   {
        Student::GetData(); // Instead of super.GetData()
        cout << "weekly pay: $" << GetPay() << endl;
   }
private:
   double UnderWage;
   double UnderHours;
};
// Same model for grad, special grad classes
```

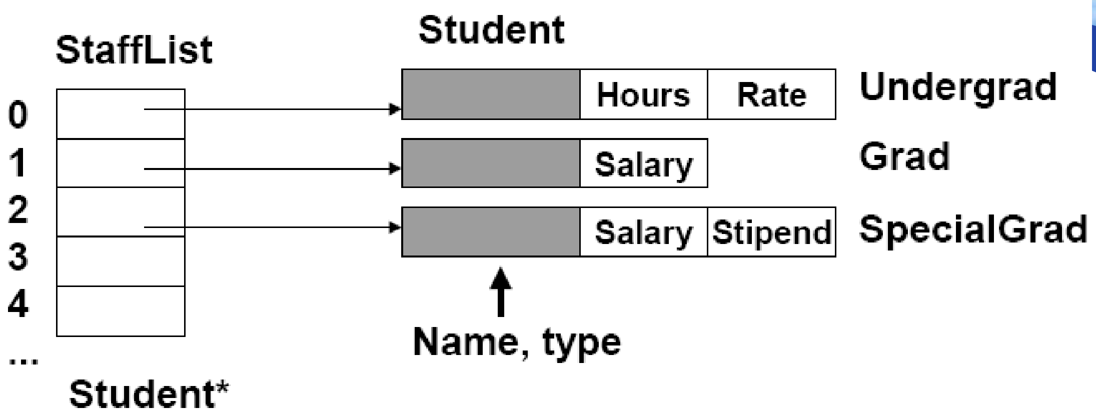
Research project class

```
class RProject {
public:
   explicit RProject(int Size);            // Constructor
   void addStudent(Student* RMember);      // Adds pointer
   void listPay();           // List students, pay on project
private:
   Student** StaffList;      // Array of pointers to Student
   int StaffSize;            // Maximum staff size
   int count;                // Actual staff size
};

// Cannot store the Student object directly; it would not have
// the additional data of the derived classes!

// Should have destructor, etc. if this were for actual use
// Rproject ‘has a’ Student array
```

Research project class, p.2

```
RProject::RProject(int Size) {
   StaffSize= Size;
   StaffList= new Student*[StaffSize];
   count= 0;}

void RProject::addStudent(Student* RMember){
   StaffList[count++]= RMember;
   return; }

void RProject::listPay() {
   cout << "Project staff " << endl;
   for (int i= 0; i < count ; i++)
     StaffList[i]->GetData();     // (*StaffList[i]).GetData()
   return; }

// Output is same as before:
// List of students with their specific pay on the project
```

Research project class



StaffList

0
1
2
3
4
...

Student*

Student

Hours	Rate	Undergrad
Salary		Grad
Salary	Stipend	SpecialGrad

Name, type

C++ will dynamically set Student pointers to Undergrad, Grad, SpecGrad to get the pay for each

If we stored Student objects in StaffList instead of pointers, we would only have the base Student class data!

Main program

```
int main( ) {
// Define 3 students (unchanged from last time)
  Undergrad Ferd("Ferd", "Smith", 8.0, 12.00);
  Ferd.GetData();
  Grad Ann(”Ann", "Brown", 1500.00);
  Ann.GetData();
  SpecGrad Mary("Mary", "Barrett", 2000.00);
  Mary.GetData();
  cout << endl;

// Add 3 students to project and list their pay
  RProject CEE1(5);
  CEE1.addStudent(&Ferd);
  CEE1.addStudent(&Ann);
  CEE1.addStudent(&Mary);
  CEE1.listPay();
  return 0;}        // Output is list of students and pay
```