

# Introduction to Systems Programming

## מבוא לתוכנות מערכות

234124

Slides by Omer Strulovich , Ron Rubinstein and Liav Adi

*With thanks to: Gil Barequet, Chaim Gotsman, Yechiel Kimchi, Yossi Gil,  
Eliezer Kantorowitz, Ayal Itzkoviz, Dani Kotlar and Itai Sharon.*



# חזקיכם אתם

## מוקד סיוע ותמיכה לסטודנטים



**לפניהם סטודנטים בנושאים הבאים יש לסרוק את הבקרוד**

### נושא אקדמיים והתאמות למשרדי מילואים

כגון מועד מיוחד, קורס קדם, חזרה על קורס, קבוצות התמחות, פטורים, שיפור ציון, בקשת גמר לימודים, פגישה עם סגן הדיקן ללימודי הסמכה ועוד

### נושא אישים

כגון סיוע בלמידה, סיוע רגשי, סיוע כלכלי, ועוד

# Course Objectives

- ▶ Learn to develop **large software**
- ▶ Learn modern programming methodology
- ▶ Learn C++
- ▶ Learn Python



# Program

# Software

- ▶ Typically 10-1,000 lines

- ▶ Typically at least 10,000 lines

## Consequences

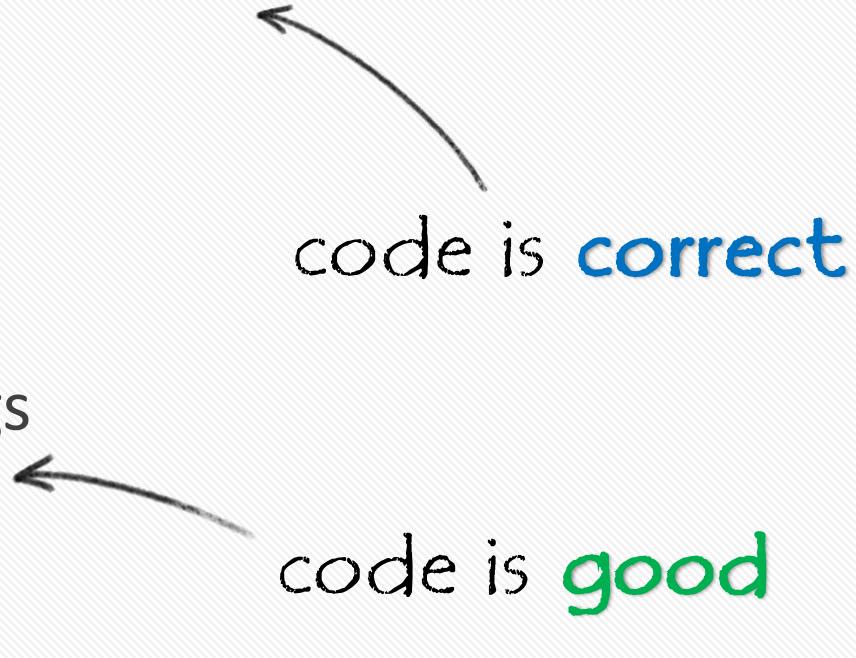
- ▶ Developed by an individual
- ▶ Used by the programmer.
- ▶ Can be coded immediately
- ▶ May have flaws
- ▶ Requires debugging
- ▶ No standards required
- ▶ **(Relatively) easy**

- ▶ Developed by a team
- ▶ Used by a “customer”
- ▶ Requires analysis/design
- ▶ Must be robust
- ▶ Requires long-term maintenance
- ▶ Standard compliant
- ▶ **Difficult !!**

*Example: A sorting program*

*Example: MS-Word*

# What makes software **good**?

- ▶ Fulfilling requirements
    - ◆ Determining software requirements is not the subject of this course
  
  - ▶ Easy to maintain:
    - ◆ Find errors early, fix bugs
    - ◆ Add new features
    - ◆ Portable
- 
- code is **correct**
- code is **good**

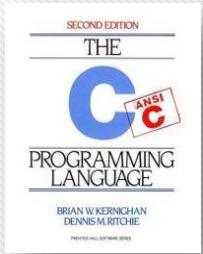
# What makes code **good/bad**?

- ▶ **Simple** to read
- ▶ Follows unified **coding conventions**
- ▶ Avoids **code duplication**
- ▶ Can be **tested** easily
- ▶ Errors are **found early**
- ▶ **Intuitive**
- ▶ Efficient?
- ▶ Commented? - *It is better if the code explains itself*

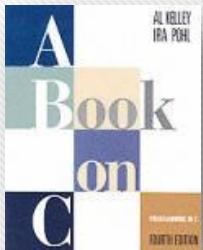
# The course in a nutshell

- ▶ More C functionality
- ▶ The C++ language
- ▶ Object Oriented Programming and design
- ▶ Software development, testing, version control
  
- ▶ Working in a UNIX environment (tutorials)
- ▶ Scripting with Python (tutorials)

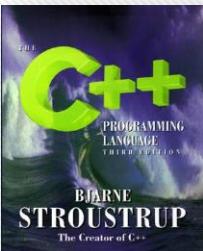
# Bibliography



The C Programming Language (2nd Edition, ANSI-C)  
B. Kernighan and D. Ritchie, Prentice-Hall, 1988

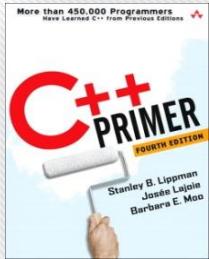


A Book on C: Programming in C (4th Edition)  
A. Kelley and I. Pohl, Addison-Wesley Professional, 1998



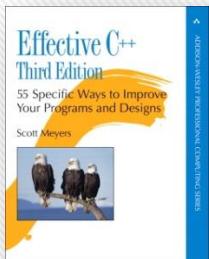
The C++ Programming Language (4th Edition)  
B. Stroustrup, Addison-Wesley Longman, 2013

# Bibliography



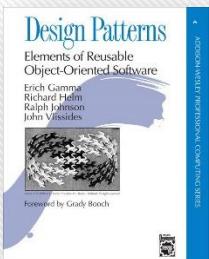
## C++ Primer (5th Edition)

S. Lippman, J. Lajoie, B. Moo, Addison-Wesley Professional, 2012



## Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)

S. Meyers, Addison-Wesley Professional, 2005



## Design patterns: Elements of Reusable Object Oriented Software

Richard Helm, Ralph Johnson, and John Vlissides 2002

# Web Resources



<https://stackoverflow.com/>



<https://www.codeguru.com/>



<https://www.codeproject.com/>



<https://docs.python.org>

And a lot more, just search in Google ...

*"Programs must be written for people to read,  
and only incidentally for machines to execute."*

- Abelson and Sussman

# Advanced C Programming

Reminders and Advanced Features

# Enumerated Types (**enum**)

- ▶ The **enum** keyword defines a new type **and** a set of named values it can take
  - ◆ Each name is associated with an integer value
  - ◆ The **names** are more important than the actual values

```
// definitions
enum Gender { MALE, FEMALE, NON_BINARY};
enum Month { JAN, FEB, MAR, /* ... */, DEC, MONTH_NUM };
enum Season { SUMMER=1, FALL, WINTER=8, SPRING };

// usage
enum Gender gender = MALE;
enum Season seasons[MONTH_NUM];
seasons[JAN] = WINTER;
...
int count = 0;
for (int month = 0; month < MONTH_NUM; month++) {
    count += (seasons[month] == WINTER);
}
printf("%d months are in winter\n", count);

enum Month next_month(enum Month current) {
    if (current == DEC)
        return JAN;
    return (enum Month)(current + 1);
}
```

useful trick!

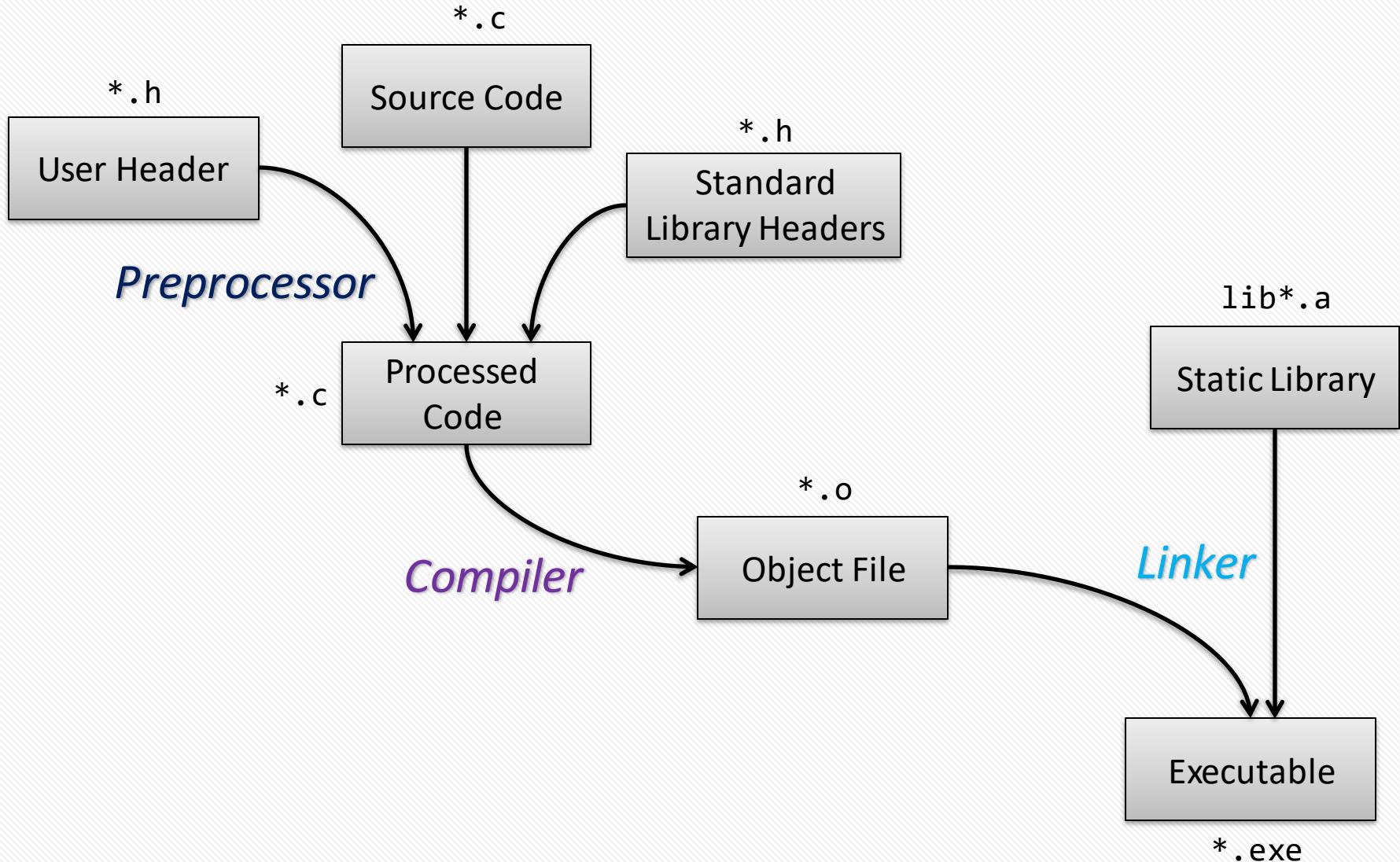
# Structures (`struct`)

- ▶ Programming with basic C types can become **complicated and tedious** when more complex objects are involved
- ▶ **Structures** are the basis for defining new types in C that can represent complicated objects
- ▶ The **struct** keyword defines a new type which is a composition of several fields
  - ◆ Each field has a type and a name
  - ◆ A field can itself be a struct
  - ◆ Fields are accessed with the '.' or '->' operators
- ▶ **Syntax:** `struct name { fields-list };`

# Example: Complex Numbers

```
struct complex {  
    double re, im; // real and imaginary parts  
};  
  
the name of our new type  
  
struct complex c1, c2;  
  
c1.re = 1.0;  
c1.im = 3.0;  
c2.re = 2.0;  
c2.im = 5.0;  
  
printf("c1 = %f + i * %f", c1.re, c1.im);
```

# The Compilation Process



# Function declaration

- ▶ Reminder: If a function **f2** calls **f1** in the **same file**
  - ◆ **f1** needs to be defined before **f2**  
**or**
  - ◆ a **declaration** of **f1** should be provided before **f2** (then the body can appear later)

```
int array_max(int a[], int n);

void test_array_max() {
    int a[] = { 3, 4, 1, 5, 6 };
    int maxval = array_max(a, sizeof(a) / sizeof(int));
    print_array(a);
    ...
}

int array_max(int a[], int n) {
    int result = a[0];
    for (int i = 1; i < n; ++i) {
        result = max(a[i], result);
    }
    return result;
}
```

utility.c

# Multi-File Projects

- ▶ A typical project has **hundreds (or more)** of functions, types, and constants
- ▶ We typically want to **separate our code into multiple files**
  - ◆ Makes it easier to **find code** which is relevant to a specific task
  - ◆ Makes it easier to **cooperate** on a project

```
int max(int a, int b) {  
    return a > b ? a : b;  
}  
  
int array_max(int a[], int n) {  
    int result = a[0];  
    for (int i = 1; i < n; ++i) {  
        result = max(a[i], result);  
    }  
    return result;  
}  
  
void print_array(int a[], int n) {  
    ...  
}
```

utility.c

```
void test_array_max() {  
    int a[] = { 3, 4, 1, 5, 6 };  
    int maxval = array_max(a, sizeof(a) / sizeof(int));  
    print_array(a);  
    ...  
}  
  
void test_array_sum() {  
    ...  
}
```

testing.c

```
int main() {  
    test_array_max();  
    test_array_sum();  
    return 0;  
}
```

main.c

# Multi-File Projects

- ▶ Several files can be easily compiled into one executable file:

```
> gcc main.c utility.c testing.c
```

← don't forget to **add flags**

- ▶ In order to call the function **f** defined in file **a.c**

from a different file **b.c**, **f must be declared** in **b.c**

→ we'll soon replace  
this with #include

utility.c

```
int array_max(int a[], int n) {  
    int result = a[0];  
    for (int i = 1; i < n; ++i) {  
        result = max(a[i], result);  
    }  
    return result;  
}
```

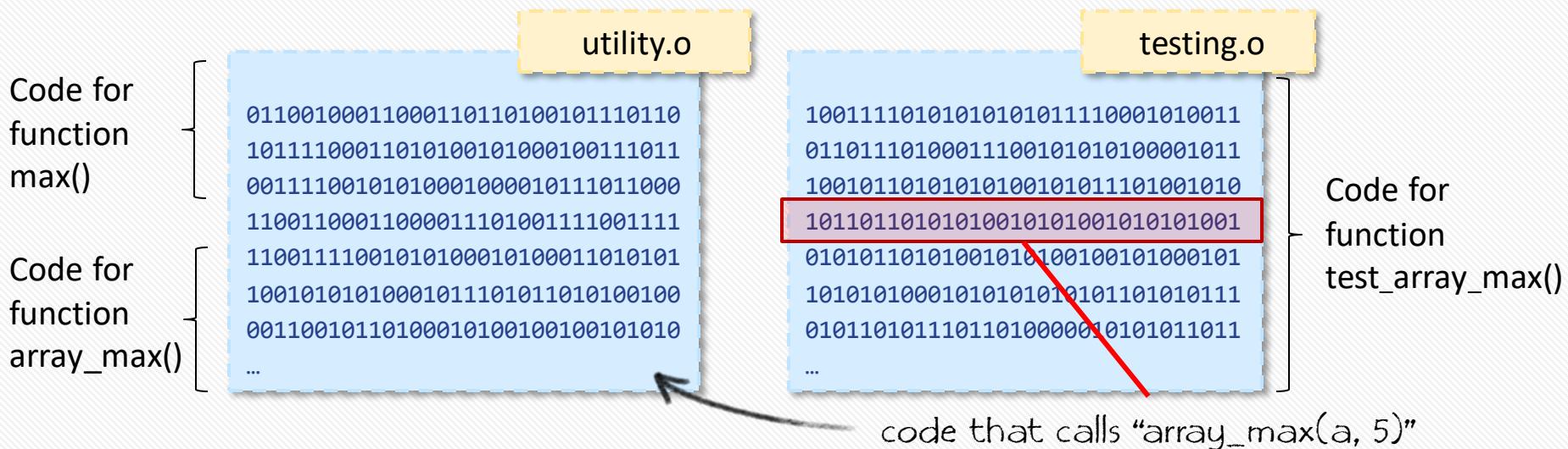
```
int array_max(int a[], int n);  
  
void test_array_max() {  
    int a[] = { 3, 4, 1, 5, 6 };  
    int maxval = array_max(a, sizeof(a) / sizeof(int));  
    print_array(a);  
}
```

testing.c

- ▶ In order to use a **type**, it must be **defined**

# Multi-File Projects

- An **object file** contains the compiled (machine) code of a single source file
  - An object file **cannot be executed** since it may contain calls to functions that are defined in other files



- The “-c” flag makes GCC stop after creating object files

```
> gcc -c utility.c testing.c
```

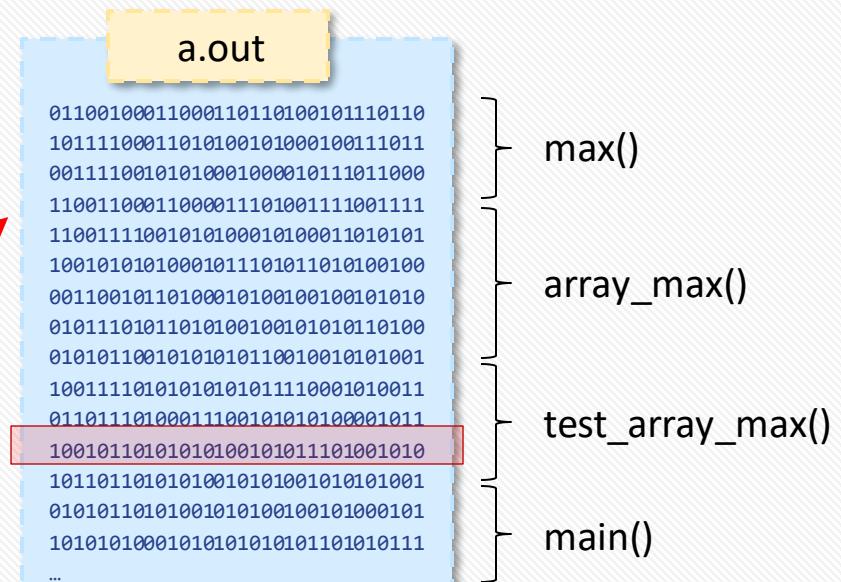
\* Binary code is for illustration purposes and is not real machine code

# Multi-File Projects

- ▶ The **linking** phase takes several object files as input, and merges them into a single executable file
  - ◆ It associates each function call with the code of that function
- ▶ Each function must be defined in exactly one object file
- ▶ A `main()` function must be defined
  - ◆ This function is called when the program starts

```
> gcc utility.o testing.o main.o
```

This machine code now means “call the function located at address XXXX, with parameters `a` and `5`”



# Multi-File Projects

- ▶ Compilation and linking can be combined. In this case, every **.c** file is first compiled to a **.o**, and then all the **.o** files are linked:

```
> gcc main.c utility.c testing.c
```

- ▶ Different combinations are possible...

```
> gcc main.c utility.o testing.o
```

- ▶ GCC (like other compilers) has flags to perform only specific stages:
  - ◆ **-E**: Stop after preprocessing
  - ◆ **-c**: Stop after compilation (only create object files)
  - ◆ **-o <name>**: Specifies the output filename (default is **a.out**)

# Multi-File Projects

- ▶ Recompiling an entire program can take **many hours**
- ▶ By using **selective compilation**, we can **avoid most of the time** needed for recompiling
  - ◆ Only the object files that have changed need to be recompiled
  - ◆ Linking still needs to be done, but this is faster
- ▶ As a project gets bigger, tools are needed to manage its modular compilation
  - ◆ **GNU Make** is an example of such a tool
  - ◆ Nowadays IDEs handle this automatically

# Multi-File Projects and structs

```
struct complex {  
    double re, im; // real and imaginary parts  
};  
typedef struct complex Complex;  
  
Complex complexAdd(Complex x, Complex y);
```

complex.h

Allows using Complex  
in lieu of **struct complex**

```
#include "complex.h"  
  
Complex complexAdd(Complex x, Complex y) {  
    Complex z;  
    z.re = x.re + y.re;  
    z.im = x.im + y.im;  
    return z;  
}
```

complex.c

# Type Definitions (**typedef**)

- ▶ The **typedef** keyword declares a new name for a type
- ▶ Usage: just write the **typedef** keyword, followed by code that looks like a variable declaration of the desired type
  - ◆ The "variable" declared in this way will actually be a nickname to the type it was defined as
- ▶ A **typedef** will commonly appear in a **header (.h) file** so it can be shared by many code files

```
typedef int Length; // Length is a nickname for int
```

types.h

```
#include "types.h"  
...  
Length len;           // equivalent to int Len;  
Length lengths[SIZE]; // equivalent to int Lengths[SIZE];
```

geometry.c

# Uses of **typedef**

- ▶ Defining a type that may need to be changed in the future

```
typedef double Real; // represents real numbers in a physics simulation  
                     // if we run into memory problems, change to float
```

- ▶ Improve code readability

```
Length l1, l2; // we understand l1 and l2 represent lengths
```

- ▶ Hide implementation details

```
// stdio.h defines a FILE type for reading and writing files  
#include <stdio.h>  
  
FILE* in_file = fopen("params.dat"); // implementation of the FILE type  
                                   // is irrelevant - it is only used  
                                   // through dedicated functions
```

# Uses of **typedef**

- ▶ Abbreviate long type names

```
typedef struct { double re, im; } Complex; // now we can use Complex
```

- ▶ Example: define a 3-D point type for a physics simulation

```
typedef Real Point3D[3]; // Point3D represents an array of 3 doubles
```

```
Point3D p, q; // p and q are each an array of 3 doubles.  
// equivalent to: double p[3], q[3]
```

```
Point3D points[N]; // define an array of N points  
// points[0] ... points[N-1] are each a double[3]  
// equivalent to: double points[N][3]
```

# Example: **struct** Person

person.h

```
typedef enum Gender {  
    MALE, FEMALE, NON_BINARY  
} Gender;  
  
typedef struct address {  
    char* street_address; // street & number  
    char* city;  
    unsigned int zip;  
} Address;  
  
typedef struct {  
    unsigned long id;  
    char* name;  
    Gender gender;  
    Address address;  
} Person;
```

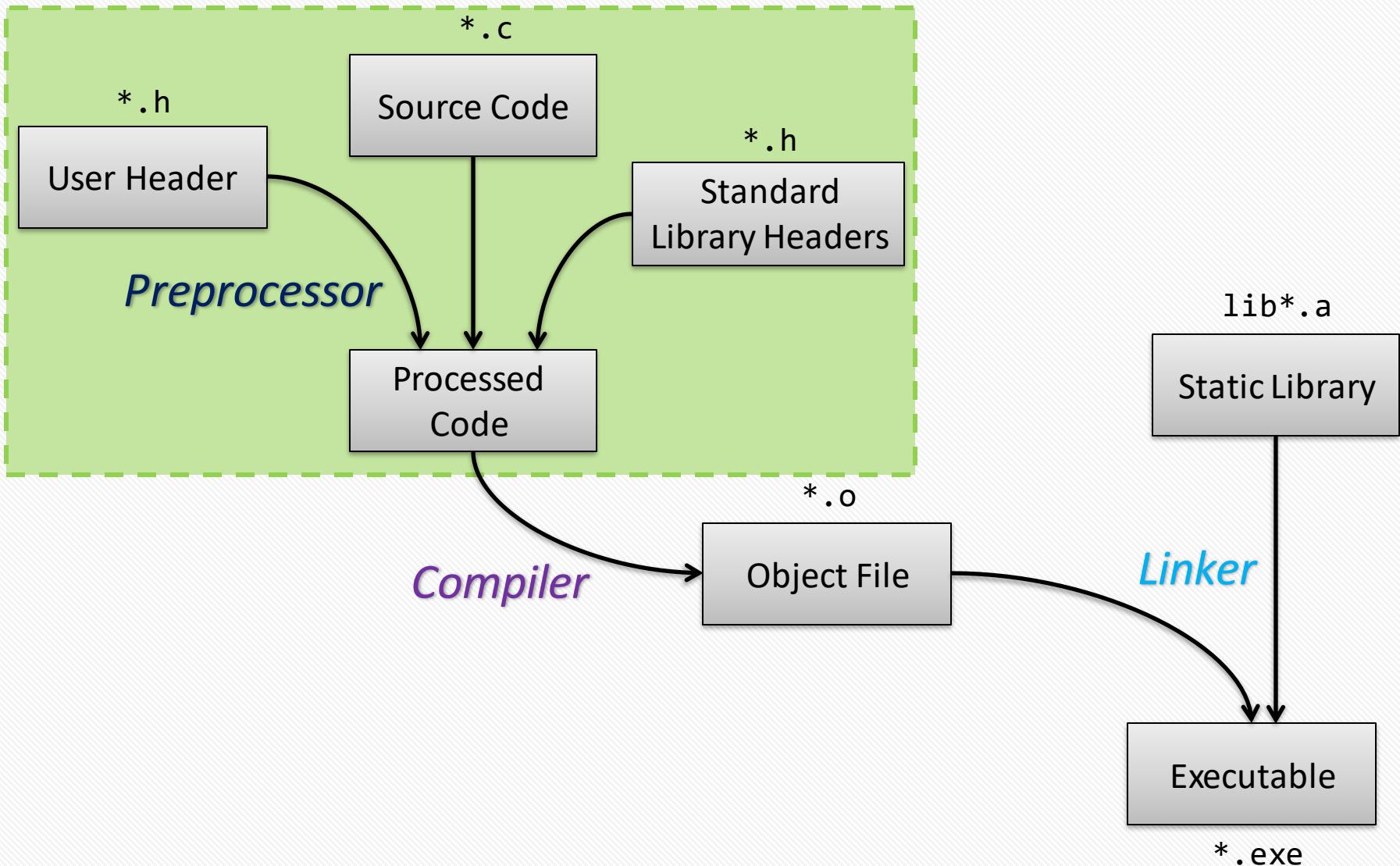
can be omitted



```
#include "person.h"  
  
unsigned long id, zip;  
char *name1, *name2;  
Person person, *person_ptr;  
  
...  
  
person.id = id;  
person_ptr->id = id+1; // or: (*person_ptr).id  
person_ptr->name = name1;  
person.address.zip = person_ptr->address.zip;
```

structs should generally come with  
**functions** for handling them

# The C Preprocessor



# The C Preprocessor

## ▶ File Inclusion

```
#include <file-name>
#include "file-name"
```

What is the difference?



## ▶ Macro definitions

```
#define identifier value
#define identifier(variables) value
#undef identifier
```

## ▶ Conditional compilation

```
#if condition
#ifndef identifier
#endif identifier

#else
#endif
```

# File Inclusions

- The **#include** directive causes the preprocessor to paste the content of the included file

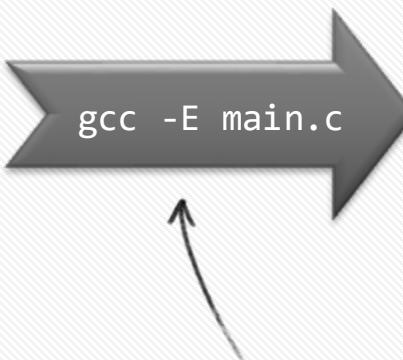
functions.h

```
int function1(int n);
int function2(int n);
```

main.c

```
#include "functions.h"

int main() {
    printf("%d\n", function1(5));
    printf("%d\n", function2(8));
    return 0;
}
```



```
int function1(int n);
int function2(int n);

int main() {
    printf("%d\n", function1(5));
    printf("%d\n", function2(8));
    return 0;
}
```

only preprocess  
the code

# C Preprocessor: Macros

```
#define identifier token-sequence  
#define identifier(parameters) token-sequence
```

- ▶ The preprocessor replaces each appearance of *identifier* in the code by the *token-sequence*
  - ◆ If **parameters** are involved, they are replaced with their **arguments**
  - ◆ No spaces are allowed between the identifier and the parentheses
- ▶ A macro is active from the **line after its declaration** and until the end of the file



why?

# Macro Examples

```
#define MAX_STR_LEN 20
```

Use capital letters for macros

```
#define IS_UPPER(c) ((c) >= 'A' && (c) <= 'Z')
```

```
#define TO_LOWER(c) (IS_UPPER(c)? (c) - 'A' + 'a' : (c))
```

```
#define SQR(x) ((x) * (x))
```

Parentheses prevent  
unintended behavior

- ▶ IS\_UPPER, TO\_LOWER and SQR are all better off implemented as **functions**.

```
double sqr(double x) {  
    return x*x;  
}
```

```
X = SQR(++i); // bad (result is undefined)  
X = sqr(++i); // no problem (ordinary function call)
```

- ▶ Prefer using **functions** whenever possible

# Conditionals

- ▶ We can check if a macro has been defined using

```
#ifdef MY_MACRO
```

- ▶ Similarly, we can check if a macro has **not** been defined using

```
#ifndef MY_MACRO
```

- ▶ We can use **#else** with conditional statements to specify alternative code to include if the condition fails

```
#ifdef DEBUG_ON
#define DEBUG_PRINT(MSG) printf("%s", MSG)
#else
#define DEBUG_PRINT(MSG) ((void)0)
#endif
```

a statement that does nothing, but still requires a ';' after it to compile

# Conditionals

- ▶ Conditionals can help us add **extra checks when debugging**, without affecting the performance of the **final product**

```
int safe_get(int *arr, int size, int i) {  
#ifndef NDEBUG  
    if (arr == NULL) {  
        fprintf(stderr, "Null array\n");  
        exit(1);  
    }  
    if (i < 0 || i >= size) {  
        fprintf(stderr, "Out of bounds\n");  
        exit(1);  
    }  
#endif  
    return arr[i];  
}
```

This is the **common convention** – we are **debugging** unless specified otherwise (using `NDEBUG`)

```
int sum = 0;  
for (int i = 0; i < n; i++) {  
    sum += safe_get(arr, i);  
}
```

Adding `#define NDEBUG` removes all checks from compiled code

Better yet, use `gcc -DNDEBUG`

# The `assert` macro

- ▶ A good example for using the preprocessor
  - ◆ Used to help find errors as early as possible
  - ◆ Defined in `<assert.h>`
- ▶ Usage: `assert(condition);`
  - ◆ When debugging, `assert` evaluates the condition, and if it fails, immediately **terminates the program** with an error message
  - ◆ For release (when `NDEBUG` is defined), **nothing is done** (the condition is never evaluated)

```
#include <assert.h>
...
int safe_get(int *arr, int size, int i) {
    assert(arr != NULL);
    assert(i >= 0 && i < size);
    return arr.a[i];
}
```



```
#include <assert.h>
...
int sum(int* array, int size) {
    assert(array != NULL && size >= 0);
    ...
}
```

Assertion failed: arr != NULL, file main.c, line 23

# File Inclusions (revisited)

- ▶ Most declarations must not appear more than once
  - ◆ We must therefore protect against including a file multiple times

utility.h

```
// conditional inclusion of this file
#ifndef UTILITY_H
#define UTILITY_H

// the contents of utility.h comes here
...
#endif /* UTILITY_H */
```

utility.c

```
#include "utility.h"
...
```

main.c

```
#include "utility.h"
#include "testing.h" // will NOT include utility.h again
...
```

testing.h

```
#ifndef TESTING_H
#define TESTING_H
#include "utility.h"

// the contents of testing.h comes here
...
#endif /* TESTING_H */
```

testing.c

```
#include "utility.h"
#include "testing.h" // will not include utility.h again
...
```

# The **const** keyword

```
const type identifier = expression; // cannot be changed after initialization
```

```
const double PI = 3.141592654;
const double EPSILON = 1e-6;
double rad;

if (PI*rad*rad < EPSILON) {
    printf("circle is too small!\n");
}
```

## ▶ **const** versus **enum**

- ◆ Can represent non-integer values
- ◆ Use const when the **value** of the constant has a meaning

## ▶ **const** versus **#define**

- ◆ A const variable can get its value at runtime
- ◆ A const variable is scoped (like all C variables)

# Const and Pointers

- ▶ A **constant pointer** cannot change **where** it points to

```
int x = 5;
int* const ptr = &x;      // ptr will always point to x
*ptr = 7;                // fine, changes x to 7
ptr++;                  // will not compile (ptr is const!)
```

- ▶ A **pointer to a const** cannot change **what** it points to

```
int x = 5;
const int* ptr = &x;        // ptr will never change the value of x
*ptr = 7;                  // will not compile (ptr points to a const)
ptr++;                     // fine, ptr points to another location
*ptr = 10;                 // still won't compile
const int* const ptr2 = &x; // can combine both const types
```

# Const and Pointers

```
Person p;  
printPerson(p)  
...  
  
printPerson(Person p)  
{  
    printf("%d", p.id);  
}
```

**Passing by value** ensures a **function cannot change** a passed **struct**, but this wastes memory and can be slow

```
Person p;  
printPerson(&p)  
...  
  
printPerson(const Person* p)  
{  
    printf("%d", p->id);  
}
```

**Solution:**  
Use **const pointers** instead

# Pointers to Functions

- ▶ A **function pointer** contains the address of a function's machine code in memory
- ▶ Similar to other pointers, a function pointer knows **where** the function is in memory, and also what the **function type** is.
- ▶ The **type of a function** is the function's signature – the number and type of its arguments, and its return type.

From <math.h>

```
double sin(double);
double cos(double);
double fabs(double);
```

All these functions have  
the **same type**

```
double (*fptr)(double); // fptr can point to any function that
                        // takes one double and returns a double
fptr = cos;           // point fptr to the code of function cos()
fptr = printf;        // will not compile (printf has wrong type)
```

# Pointers to Functions

- ▶ Using function pointers we can pass a function as a parameter to another function
  - ◆ Useful when we want to choose which function to call **only during runtime**
- ▶ A function pointer can be used to call the function it points to by simply using the **() operator** – just like a normal function!

```
void apply_to_array(double a[], int n, double (*operation)(double)) {  
    for(int i=0; i<n; i++) {  
        a[i] = operation(a[i]);  
    }  
}
```

```
double a[100];  
apply_to_array(a, 100, sin); // replaces each a[i] with sin(a[i])
```

# Example: Computing a Derivative

```
double derivative(double (*func)(double), double x) {  
    const double h = 1e-6;  
    double diff = func(x+h) - func(x-h);  
    return diff / (2*h);  
}
```

```
#include <math.h>  
  
double square(double x) {  
    return x*x;  
}  
  
void compute() {  
    printf("%lf\n", derivative(square, 1));      // 2.000  
    printf("%lf\n", derivative(sin, 0));           // 1.000  
    printf("%lf\n", derivative(cos, PI/2));        // -1.000  
}
```

*"C is quirky, flawed, and an enormous success."*

- Dennis M. Ritchie (1941-2011).

# Introduction to C++

A Tour of the C++  
Programming Language

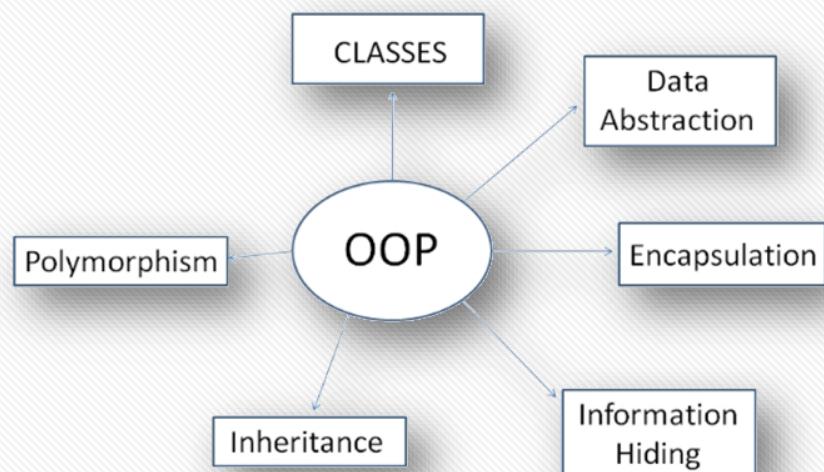
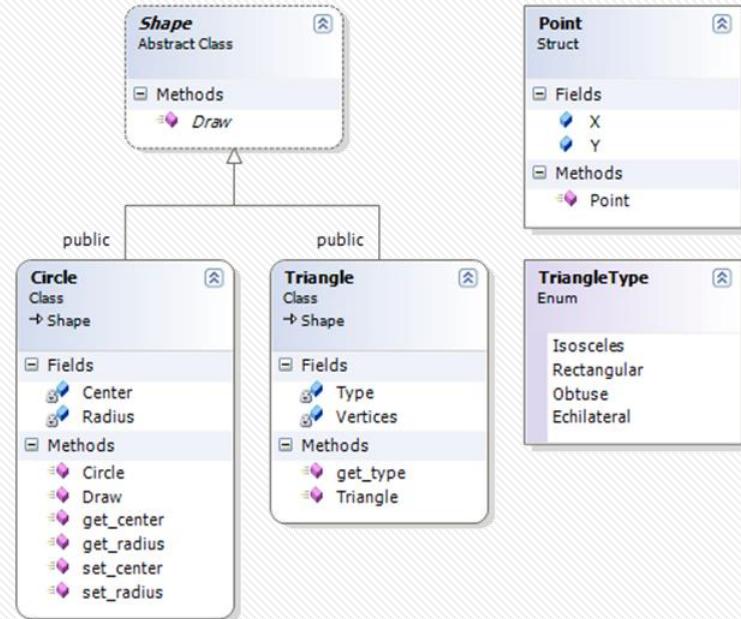
# Why C++?

- ▶ We already know C
  - ◆ **Everything** that can be done with a computer, **can be done in C**, so ...
- ▶ **Why C++??**
  - ◆ **Object-oriented language**
  - ◆ Industry standard
  - ◆ Extension of C → efficient code, cross-platform
  - ◆ Larger toolbox, more features → code can be made **simpler**

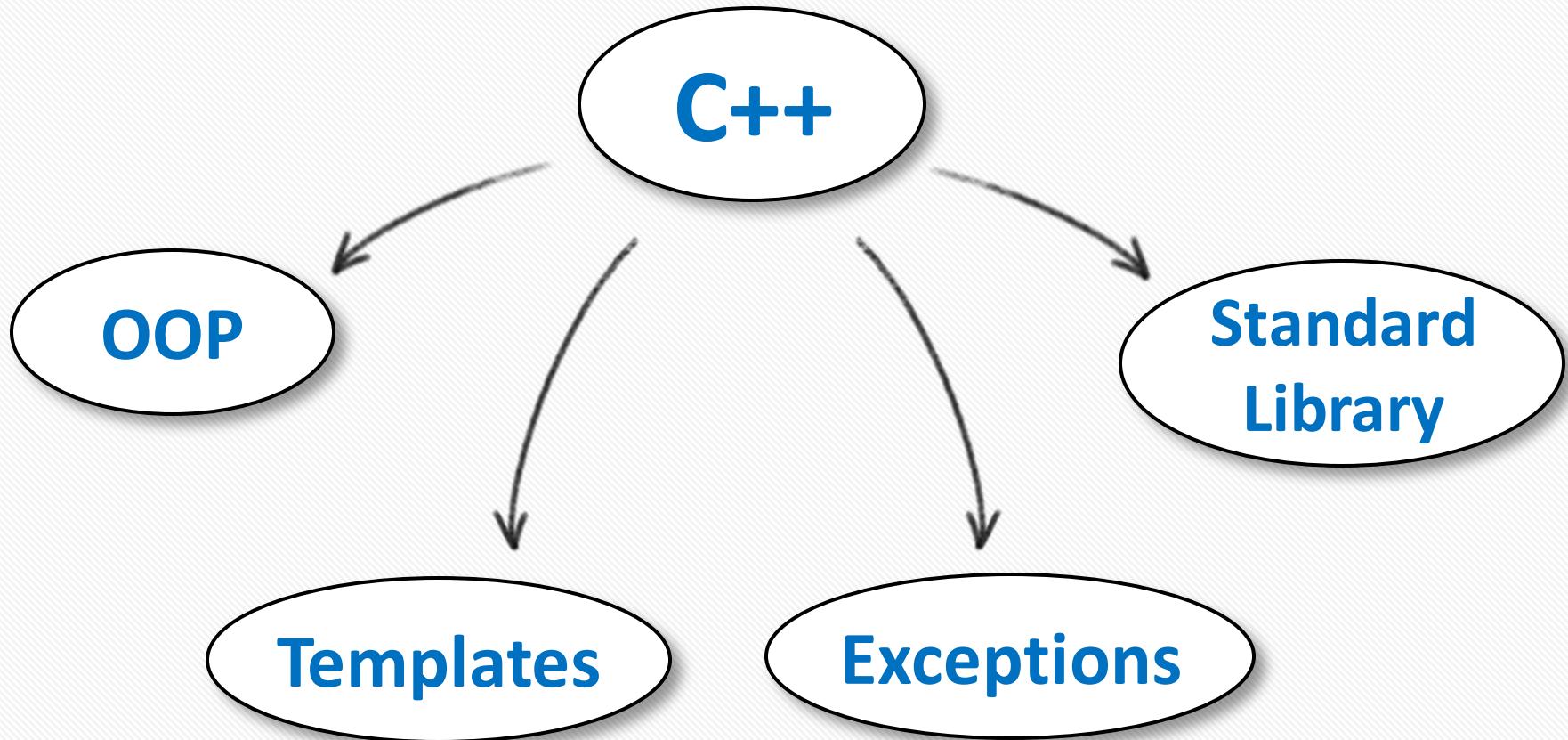


# Why OOP?

- ▶ An **intuitive** concept
- ▶ Enables **modularity** and **information hiding**
- ▶ Makes testing and maintenance **easier**



# Main Features of C++



# An Overview of C++

- ▶ C++ is based on C
  - ◆ However, there are some fundamental **differences**
- ▶ We will first cover a few of the **enhancements** needed before we move on
- ▶ C files end with .c, C++ files end with .cpp

# C++ Is Based On C

- ▶ Most C code will also work in C++

```
#include <stdio.h>
```

C

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

```
int main() {
    printf("Hello World!\n");
    printf("5! = %d\n", factorial(5));
    return 0;
}
```

```
#include <iostream>
using namespace std;
```

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

```
int main() {
    cout << "Hello world!" << endl;
    cout << "5! = " << factorial(5) << endl;
    return 0;
}
```

C++

# Input / Output in C++

- ▶ General standard streams :

- ◆ **cin** – standard input
- ◆ **cout** – standard output
- ◆ **cerr** – standard error



*global variables  
defined in  
<iostream>*

- ▶ Printing to cout is done using **<<**
- ▶ Receiving input is done using **>>**
- ▶ Can concatenate multiple inputs / outputs

```
int main() {  
    int n;  
    cin >> n; // read an int from stdin  
    cout << "You entered " << n << endl;  
    return 0;  
}
```

*end line  
(print a new line)*

# Input / Output in C++

- ▶ No need to **remember** format specifiers, a.k.a flags (%d, %s ...)
  - ▶ Impossible to use the wrong flag (=**undefined behavior** in C!)
- ▶ Can support input/output of **new types**

next lecture!

```
#include <stdio.h>
```

```
int main() {  
    int n;  
    const char* str = "Hello";    equivalent to  
    int* ptr = &n;  
    fscanf(stdin, "%d", &n);  
    fprintf(stdout, "%s\n", str);  
    fprintf(stderr, "%d at %p\n", n, ptr);  
    return 0;  
}
```

C

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int n;
```

```
    const char* str = "Hello";
```

```
    int* ptr = &n;
```

```
    cin >> n;
```

```
    cout << str << endl;
```

```
    cerr << n << " at " << ptr << endl;
```

```
    return 0;  
}
```

C++

ignore this for now

# Const Correctness

- ▶ C++ has the **const keyword** like in C
  - ◆ However, it is used **much more widely** in C++
- ▶ Why is **const** not that useful in C?
  - ◆ It's very easy to mistakenly “lose” constness

yup, this  
compiles in C!

```
const int n = 10;
const int* p = &n;
int* q = p;
*q += 5; // modifies n!!!
```

- ▶ This error might seem obvious, but in **real code** with thousands of functions, it can be very easy to lose track

```
Complex complexSubtract(const Complex* c1, const Complex* c2) {
    complexNegate(c2);
    return complexAdd(c1, c2);    yes, this compiles
}                                too! (and will change
                                    c2...)
```

```
void complexNegate(Complex* c) {
    c->re *= -1;    c->im *= -1;
}
```

# Const Correctness

- In C++, you **cannot remove const** without an explicit cast

compilation errors in  
C++!

```
Complex complexSubtract(const Complex* c1, const Complex* c2) {  
    complexNegate(c2);  
    return complexAdd(c1, c2);  
}
```

```
const int n = 10;  
const int* p = &n;  
int* q = p;  
int* q2 = (int*)p;
```

compiles, though  
**almost never** a  
good idea

- Const correctness** means that once an object is marked as **const**, this constness propagates through all involved code, with the **compiler verifying this**
  - Having the **compiler find errors for us** is an **essential technique** of experienced programmers

## References

- ▶ C++ allows us to define a **reference to a variable**
    - ◆ The notation **T&** means a reference to type **T**
    - ◆ A reference acts as an **alternative name** for the variable

```
int i = 1;
```

```
int& r = i;
```

```
int x = r;
```

$$r = 2;$$

j = 2

$$x = 1$$

r and i refer to  
the same int

# References

- ▶ References **must be initialized**

```
int& r;
```

**error:** initializer missing

- ▶ Once initialized, a reference target cannot be changed

```
int i = 0, j = 5;  
int& r = i;  
r = j;  
int* p = &r;
```

*i becomes 5*

*p points to i*

- ▶ A reference is **similar** to a constant pointer

- ◆ However, a reference **isn't an object** that can be manipulated like a pointer
- ◆ A reference always **refers to a legal object**

no “NULL reference”

# References

- ▶ References can be used as function parameters
  - ◆ Can be used when a function has to modify its arguments

```
void swap(int& a, int& b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
int x = 5;  
int y = 3;  
  
swap(x, y);
```

- ▶ Like the **address-of** operator &, a reference cannot refer to a temporary value or a constant

```
swap(a, 3);  
swap(a, x + y);
```

**error**, can't initialize an int& with a temporary ( $x+y$ ) or a constant (3)

# References

- ▶ References can also be used as a return value

```
int& max(int array[], int size) {  
    int maxIndex = 0;  
    for (int i = 1; i < size; i++) {  
        if (array[i] > array[maxIndex]) {  
            maxIndex = i;  
        }  
    }  
    return array[maxIndex];  
}
```

never return a reference  
to a local variable!!

(only to a variable that  
will exist after the  
function ends)

- ▶ References can serve as a *left-hand-side* (LHS) in assignment
  - ◆ This may seem weird for now, but we'll see many natural uses

```
int array[] = { 1, 3, 4, 2, 5 };  
max(array, 5) = 7;
```

array[4] = 7

# Const References

- ▶ A reference can be declared to refer to a const
  - ◆ In this case the referenced object cannot be changed **through the reference**
  - ◆ A **const reference** can refer to a temporary value

indicates that *print*  
does not change *r*

```
void print(const int& r) {  
    cout << r << endl;  
}
```

```
int a = 5;  
print(a)  
print(3);  
print(3 + a);
```

o.k., because *r* is  
a *const int&*

avoid duplicating large  
structs in memory

```
void print(const Complex& c) {  
    cout << c.re << " " << c.im << endl;  
}
```

```
Complex c = { 2.0, 3.2 };  
print(c);
```

# Function Overloading

- ▶ C++ allows to define **several functions** with the **same name**

- ◆ We say that the functions are **overloaded** on the name
- ◆ These functions must have different enough parameters
  - The compiler chooses the function which best fits the arguments

```
void swap(int&, int&);  
void swap(double&, double&);  
  
int a, b;  
double c, d;  
  
swap(a,b);    // calls first swap  
swap(c,d);    // calls second swap  
swap(a,c);    // syntax error
```

more on this in  
the tutorials



# Default Arguments

- ▶ C++ allows to define a **default value** for function arguments
  - ◆ The function may be called without this parameter, in which case the default value is passed
  - ◆ The default values are only specified in the function **declaration**

num\_utils.h

```
void printNumbers(int n = 10);
```

num\_utils.cpp

```
void printNumbers(int n) {  
    for (int i = 0; i < n; ++i) {  
        cout << i << " ";  
    }  
    cout << endl;  
}
```

do not redefine default  
arguments in the .cpp

```
printNumbers(5);  
printNumbers(); // printNumbers(10);
```

more examples in the  
tutorials

# Casting

- ▶ There is no **implicit cast** from void\* to other pointers in C++
  - ◆ C++ has mechanisms to avoid the need for such casts
- ▶ **Pointer casting** should be avoided in C++
  - ◆ There are better solutions!

*only legal in C*

```
int* ptr = malloc(sizeof(*ptr));
```

*required in C++*

```
int* ptr = (int*)malloc(sizeof(*ptr));
```

# New & Delete

- ▶ Dynamic memory allocations are done in C++ using the **new** and **delete** operators

- ◆ **new** & **delete** are **typed** (i.e., return a pointer of a **specific type**)
    - Unlike **malloc** and **free**, which **return void\***
  - ◆ **new** initializes the object
    - So there is no “**uninitialized memory**” containing garbage
    - Built-in types (such as int) can be allocated with or without initialization
- (for compatibility  
with C)

```
int* ptr = malloc(sizeof(*ptr));
*ptr = 5;
int* array = malloc(sizeof(*array) * 10);
// ... need to verify allocations ...

free(ptr);
free(array);
```

C

```
int* ptr = new int(5);
int* array = new int[10]();
// ...
delete ptr;
delete[] array;
```

C++

the () are **optional**,  
with them the  
array is initialized  
to zeros

we'll learn later what happens when **new**  
fails

# Simpler type names in C++

- ▶ In C++, we do not need to use **typedef** for enums or structs anymore
  - ◆ The type name is just the name itself

```
enum Color {RED, GREEN, BLUE, ...};
```

```
Color color = GREEN;
```

```
struct Complex {  
    double re, im;  
};
```

```
Complex c = { 2.7, 4.0 };
```

*“The C language combines all the power of assembly language with all the ease-of-use of assembly language.”*

- Mark Pearce

# Classes

Defining new data types and  
ADTs in C++

# Motivation

- ▶ Assume we need **complex numbers** in our software
  - ◆ We create the following struct:

```
struct Complex {  
    double re, im;  
};
```

- ▶ Is this enough?
  - ◆ **No!**

# Motivation

- ▶ What do we want from our Complex type?
  - ◆ **Represent** complex numbers in our code
  - ◆ Provide ways to **use** and **operate** on these complex numbers (add, subtract, ...)
- ▶ While also:
  - ◆ Preventing **code duplication**
  - ◆ Minimizing **dependency** of other code on implementation details

# Data Types

- ▶ We want a complete **data type** to represent complex numbers
    - ◆ Provide operations allowing the user to handle this type:
      - add
      - multiply
      - absolute value
      - print
      - more...
- 
- Who is the user?*

**Data Type = Representation + Operations**

# Data Types

- ▶ We want our complex data type to be **reusable**
- ▶ Can be used in different files
  - Therefore, we must define **complex.h** containing the **interface** of a complex number
- ▶ Can be passed from one project to the next
  - The **implementation** will be in its own source file **complex.cpp**

# Structs with Functions

- In C++, a struct can include **functions** that operate on it
  - Called **member functions** or **methods**

```
struct Complex {  
    double re, im;  
  
    void init(double x, double y);  
    void multiply(double a);  
};
```

this function sets  
re and im to x and  
y  
*this function*  
multiplies re and im  
by a

- Member functions **do not take any space** in the struct variables
- They are normal functions that work **specifically on our struct**

```
int main () {  
    Complex c1;  
    cout << sizeof(c1) << endl;  
    return 0;  
}
```

prints **16** (= two doubles)

# Structs with Functions

- ▶ The variable that the member function is called through is called **object**.
- ▶ The function operates on **that object's members**

```
struct Complex {  
    double re, im;  
  
    void init(double x, double y);  
    void multiply(double a);  
};
```

no need for a `typedef` here, the type name is already **Complex** (not "struct complex") in C++

sets **re** and **im** of **c1**  
to 4.0 and 1.0

```
int main () {  
    Complex c1;  
    c1.init(4.0, 1.0);  
    c1.multiply(5.0);  
    return 0;  
}
```

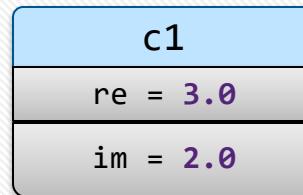
**object**

# Structs with Functions

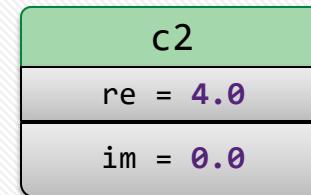
- ▶ A member function has direct access to the **member variables** of the **object that invoked it**

```
int main () {  
  
    Complex c1, c2;  
    c1.init(3.0, 2.0);  
    c2.init(4.0, 0.0);  
  
    return 0;  
}
```

set **re** and **im** of **c2**



set **re** and **im** of **c1**



member function implementation

```
void Complex::init(double x, double y) {  
    re = x;  
    im = y;  
}
```

**re** and **im** of the **invoking object**

# Structs with Functions

- ▶ A member function can access the member variables of **another object** by passing it as a parameter (like any C++ function)

```
struct Complex {  
    ...  
    void add(Complex other);  
    /* increment re and im  
       by other.re and other.im */  
};
```

```
int main () {  
    Complex c1, c2;  
    c1.init(3.0, 2.0);  
    c2.init(4.0, 0.0);  
    c1.add(c2);  
    return 0;  
}
```

re and im of  
**c1**

```
void Complex::add(Complex other) {  
    re += other.re;  
    im += other.im;  
}
```

**c2** passed by value

re and im of **c2**

# Implementing Member Functions

- ▶ Member functions can be implemented in one of two ways:
  - ◆ **Inside** the struct definition
  - ◆ **Outside** (after) the struct definition
- ▶ For readability, we will usually implement functions **outside the struct**
  - ◆ Separates the interface from the implementation details
  - ◆ Very short functions can be implemented inside

```
struct Complex {  
    double re, im;  
  
    void init(double x, double y) {  
        re = x;  
        im = y;  
    }  
  
    void add(Complex other);  
};
```

this comes after  
the struct

```
void Complex::add(Complex other) {  
    re += other.re;  
    im += other.im;  
}
```

# Dividing into Files

- ▶ The **struct definition** will usually appear in a **header file**
- ▶ **Member function** will be implemented in a **source file**
  - ◆ There are various acceptable extensions: **.cpp**, .cc or .C

```
#ifndef COMPLEX_H_
#define COMPLEX_H_

struct Complex {
    double re, im;

    void init(double x, double y) {
        re = x;
        im = y;
    }
    void multiply(double a);
    void add(Complex other);
};

#endif
```

complex.h

note the **capital C**.  
this option is **not portable**

```
#include "complex.h"    complex.cpp

void Complex::multiply(double a) {
    re *= a;
    im *= a;
}

...
```

complex.cpp

# complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H

struct Complex {
    double re, im;

    void init(double x, double y);
    void initFromPolar(double abs, double arg);
    void add(Complex c);
    void subtract(Complex c);
    void multiply(Complex c);
    void divide(Complex c);
    Complex conjugate();
    Complex inverse();
    double abs();
    double arg();
};

#endif /* COMPLEX_H */
```



Header files **should** be thoroughly documented  
**(no room on slide...)**

Users should **not** have to look in the .cpp file to know how to use these functions



They may not have access to the .cpp file

# complex.cpp

```
#include "complex.h"

void Complex::init(double x, double y) {
    re = x;
    im = y;
}

void Complex::add(Complex c1) {
    re += c1.re;
    im += c1.im;
}

void Complex::multiply(Complex c1) {
    double a = re;
    double b = im;
    double c = c1.re;
    double d = c1.im;
    init(a*c-b*d, b*c+a*d);
}
...

```

this calls init() on “myself”  
(a.k.a. the **invoking object**)

Remember, it is safer to use **init()**  
than to initialize a variable directly  
(via Complex c = { 3.2, 5.0 };)

If the fields in the struct are changed,  
the person who updates the struct is  
also **responsible** for updating  
Complex::init().

# Benefits of Data Types

- ▶ **Reusable**
  - ◆ The data type is a separate module – can be used in future projects conveniently
  - ◆ Possible to find existing code that implements this data type, and reuse it
- ▶ Keeps code **simple**
  - ◆ The user of a complex number is spared of the technical details
  - ◆ Prevents code duplication
- ▶ What about minimizing the **dependency of other code** on the implementation of Complex?

# A Dependency Problem

- ▶ Complex numbers can also be represented in **polar form**:

$$a + bi \leftrightarrow r \cdot \text{cis}(\theta)$$

- ✓ Fast multiplication
- ✓ Used for calculating the power of a number
- ✓ etc.

- ▶ Can we switch our implementation to use the polar form?

```
struct Complex {  
    double abs, arg;  
    ...  
};
```

```
void Complex::multiply(Complex c) {  
    initFromPolar(abs * c.abs, arg + c.arg);  
}
```

*So far so good...*

# A Dependency Problem

- ▶ User code of Complex **may break** if it assumes a specific implementation of the struct

```
void printComplex(Complex c) {  
    cout << c.re << " + i*" << c.im;  
}
```

Uh  
oh...

- ▶ How can we prevent the user from writing such code in the first place?
  - ◆ Solution: **block access** to the struct member variables for users of the struct

# Encapsulation

- ▶ We wish to prevent users of type Complex from **accessing the struct fields directly**
- ▶ We will **encapsulate**\* the implementation of the data type
  - ◆ The implementation will be inaccessible to the user
  - ◆ The type will be usable **only** through a set of functions that we define (its **interface**)

\* *Encapsulate = enclose in a capsule, seal off*

# Access Control

- ▶ C++ allows us to **control the user's access** to specific parts of a struct
- ▶ Every member of the struct is either **public** or **private**
  - ◆ Members that are private can **only** be accessed by functions defined within that struct

```
struct Complex {  
  
    private:  
        double re, im;  
  
    public:  
        void init(double x, double y);  
        void add(Complex c);  
};
```

```
Complex c1, c2;  
c1.init(3.0, 2.0);  
c2.init(4.0, 0.0);  
c1.add(c2);  
double d = c1.re;
```

**compilation error**  
**re** is private!

```
void Complex::add(Complex c) {  
    re += c.re;  
    im += c.im;  
}
```

**o.k.**

# Access Control

- ▶ We use the **private section** of the struct to **encapsulate** (hide) the data type implementation
- ▶ The user can only use the type **in a safe way**, via its **public interface**
- ▶ If we change only the **implementation** of a type (= its private section), users' code is **not affected**
  - Users may still need to **recompile** their code (since the .h file could change), but **no programming effort** is required

# Abstract Data Types

```
#ifndef COMPLEX_H
#define COMPLEX_H

struct Complex {
    public:
        void init(double x, double y);
        void initFromPolar(double abs, double arg);
        void add(Complex c);
        void subtract(Complex c);
        ...
    private:
        double re, im;
};

#endif /* COMPLEX_H */
```

complex.h

public section  
(interface)

private section  
(implementation)

# Abstract Data Types

```
#include "complex.h"

int main() {
    Complex c1;
    c1.init(1.0, 1.0)
    c1.re = 0.0;      // Compilation error!
    double r = c1.abs();
    Complex c2 = c1.conjugate();
    ...
    return 0;
}
```

main.c

# Abstract Data Types

- ▶ An ADT defines only the **operations** (the interface) that the type supports
  - ◆ The underlying representation and implementation are hidden from the user
  - ◆ There can be several different implementations of the same ADT

**Abstract Data Type = Operations**

# Introducing Classes

- ▶ The **class** keyword is nearly identical to the struct keyword
  - ◆ The only difference: the contents of a class is **private** by default
- ▶ We usually implement ADTs with the **class** keyword
  - ◆ **struct** are still used for simple types, like in C

equivalent

```
struct Complex {  
    private:  
        double re, im;  
  
    public:  
        void init(double x, double y);  
        void add(Complex c);  
};
```

```
class Complex {  
    double re, im;  
  
    public:  
        void init(double x, double y);  
        void add(Complex c);  
};
```

# The *this* pointer

- ▶ Each member function needs to know the object that invoked it
- ▶ The compiler implicitly sends a pointer named *this* to the member function, pointing to the calling object
- ▶ The *this* pointer is constant – it cannot point to another target

```
void Complex::add(Complex c) {  
    re += c.re;  
    im += c.im;  
}  
C++ code
```

```
c1.add(c2);
```

compilation  
void add(Complex\* const *this*, Complex c) {  
 *this*->re += c.re;  
 *this*->im += c.im  
}

How the compiler  
sees it (not real code)

```
add(&c1, c2)
```

*this* points to c1  
(the invoking  
object)

# The *this* pointer

- ▶ The *this* pointer can be used within the member function like a normal pointer
  - ◆ However, in most cases it is **not necessary** to use it explicitly

```
void Complex::add(Complex c) {  
    this->re += c.re;  
    this->im += c.im;  
}
```

ok to use *this*, but unnecessary

```
void Complex::add(Complex c) {  
    re += c.re;  
    im += c.im;  
}
```

```
void Complex::init(double re, double im) {  
    this->re = re;  
    this->im = im;  
}
```

must use *this* explicitly due to local variables

# Chaining Member Function Calls

- ▶ A common technique is to have a member function **return** the object it was invoked for
  - ◆ This allows **chaining** several calls

```
Complex c1, c2, c3;  
c1.add(c2).add(c3);
```



increment c1 by  
c2 and then by c3

- ▶ To implement this, we need to return a **reference** to **\*this**

```
Complex& Complex::add(Complex c) {  
    re += c.re;  
    im += c.im;  
    return *this;  
}
```

uh oh... what would happen  
if we returned by value?

```
Complex Complex::add(Complex c) {  
    re += c.re;  
    im += c.im;  
    return *this;  
}
```

# Const Member Functions

- ▶ Some member functions **do not modify** their object
  - ◆ Such member functions should be declared **const**
- ▶ Writing “**const**” at the end of a member function’s prototype:
  - ◆ Guarantees that it **does not change the object** it was called on
  - ◆ Allows it to be **called on a const object**

```
class Complex {  
    double re, im;  
public:  
    void init(double x, double y);  
    Complex& add(Complex c);  
    double abs()const;
```

```
void f(Complex& c, const Complex& c2) {  
    c.add(c2);  
    c2.add(c); // error - c2 is const  
    cout << c2.abs() << endl; // ok  
}
```

```
double Complex::abs() const {  
    return sqrt(re * re + im * im);  
}
```

compilation

```
double abs(const Complex* const this) {  
    return sqrt(this->re * this->re +  
               this->im * this->im);  
}
```



how the compiler sees it  
(not real code)

# Getters and Setters

- ▶ If we try to use our Complex class, we'll quickly find that we're limited
  - ◆ We sometimes actually need to **access the real or imaginary parts**
- ▶ In general, there may be cases where we will want to allow access to a field of a class
- ▶ This can be done using **one or both** of these member functions:
  - ◆ A **getter** function for **retrieving the value**
  - ◆ A **setter** function to allow **updating the value**
- ▶ There are two common naming conventions for such functions:

```
double Complex::getReal() const {  
    return re;  
}  
  
void Complex::setReal(double x) {  
    re = x;  
}
```

```
double Complex::real() const {  
    return re;  
}  
  
void Complex::setReal(double x) {  
    re = x;  
}
```

# Getters and Setters

- ▶ Why not just make a variable **public**?
  - ◆ Making a variable public **exposes the implementation** and makes that variable **part of the class interface**
    - The **interface of a class** cannot be freely changed like the private part – it can effect a lot of user code!
      - For example: what if we decided to change Complex to use **abs & arg**?
    - ◆ Using setters and getters allows us to **control the type of access** to a variable – read only, write only, or both
    - ◆ Using a setter allows us to **verify the value being assigned**, or do other computations when the variable is modified
  - ▶ **Never** make a member variable public

# Getters and Setters

- With getters and setters, we are free to **change the internal implementation** without modifying the public interface

```
class Complex {  
    double re, im;  
  
public:  
    double getReal() const;  
    void setReal(double x);  
};
```

change to internal implementation  
**does not change the interface**

```
class Complex {  
    double abs, arg;  
  
public:  
    double getReal() const;  
    void setReal(double x);  
};
```

```
double Complex::getReal() const {  
    return re;  
}  
  
void Complex::setReal(double x) {  
    re = x;  
}
```

*real* has become a **virtual variable** – it can be “get” / “set” even though there is no actual underlying variable

```
double Complex::getReal() const {  
    return abs*cos(arg);  
}  
  
void Complex::setReal(double x) {  
    double y = this->getImag();  
    abs = sqrt(x*x + y*y);  
    arg = atan2(y, x);  
}
```

# Static Member Variables

- ▶ A **static variable** in a class is part of the class, but is not stored in any object
  - ◆ Only a **single copy** of this variable exists, and all objects of the class can access it
  - ◆ The variable exists for the entire duration of the program, *even if no object is ever created from this class*
- ▶ A static class member is somewhat similar to a **global variable**
  - ◆ But it is **part of the class namespace**
    - And it can be (and often will be) designated as **private**

**private** static  
member

```
class Complex {  
    double re, im;  
    static int initCounter;  
  
public:  
    void init(double re, double im);  
    Complex& add(Complex c);  
};
```

complex.h

```
void Complex::init(double re,  
                   double im) {
```

```
    this->re = re;  
    this->im = im;  
    initCounter++;
```

}

complex.cpp

access the static class  
member

# Static Member Variables

- ▶ A static member variable must also be defined **in a source file** to actually exist
  - ◆ The class only **declares** the variable
  - ◆ The variable can also be assigned an **initial value** in the source file (otherwise, it is **initialized to zero**)

```
complex.h
class Complex {
    double re, im;
    static int initCounter;

public:
    void init(double x, double y);
    void add(Complex c);
};
```

without this line the  
variable will not exist, and  
there will be a **link error**

```
complex.cpp
int Complex::initCounter = 0;

void Complex::init(double re,
                   double im) {
    this->re = re;
    this->im = im;
    initCounter++;
}
```

the **= 0** is optional  
(will be initialized to 0 anyway)

# Static Member Variables

- ▶ Special case: **static const** members do not need to be defined in a source file, if they are initialized inside the class

```
class Complex {  
    double re, im;  
  
public:  
    void init(double x, double y);  
    Complex& add(Complex c);  
  
    static const double PI = 3.14159;  
    static const int PRESISION = 2;  
};
```

complex.h

```
Complex c;  
c.init(Complex::PI / 2.0, 0.0);
```

main.cpp

static consts do not  
need a definition in  
a source file

can access this static member  
from outside the class  
(because Complex::PI is public)

# Static Member Functions

- ▶ A **static function** of a class is a function that is allowed to access the private parts of the class, but it is invoked without any object
  - ◆ A static function can be called even if no object has been created
  - ◆ A static function does not have access to a **this** pointer

```
complex.h
```

```
class Complex {  
    double re, im;  
    static int initCounter;  
  
public:  
    void init(double x, double y);  
    Complex& add(Complex c);  
  
    static Complex conjugate(const Complex&);  
    static Complex I();  
  
    static int getInitNum() {  
        return initCounter;  
    }  
};
```

**static functions**

```
complex.cpp
```

```
Complex Complex::conjugate(const Complex& c) {  
    Complex result;  
    result.init(c.re, -c.im);  
    return result;  
}  
  
Complex Complex::I() {  
    Complex result;  
    result.init(0, 1);  
    return result;  
}
```

can access  
private  
members

```
main.cpp
```

```
Complex c1 = Complex::I();  
Complex c2 = Complex::conjugate(c1);
```

# Constructors

- ▶ The use of functions such as init() is **inelegant** and **error-prone**
  - ◆ The programmer may **forget to call it**
  - ◆ The programmer may call it **twice**
- ▶ Instead, a special member function called a **constructor** is used

```
class Complex {  
    double re, im;  
  
public:  
    Complex(double x, double y);  
};
```

complex.h

```
Complex::Complex(double x, double y) {  
    re = x;  
    im = y;  
}
```

complex.cpp

a constructor for Complex  
initialize through the calling a c'tor

```
Complex c1(1.0, 0.0);
```

```
c1 = Complex(2.0, 1.0);
```

```
Complex *pc1 = new Complex(1.0, 2.0)
```

main.cpp

temporary object

# Constructors

- ▶ A constructor has **no return value** and has **the name of the class**
- ▶ Several constructors can be **overloaded** with different parameters
  - ◆ As with other functions in C++
- ▶ Default parameters are also allowed

```
class Complex {  
  
    double re, im;  
  
public:  
    Complex();  
    Complex(double x, double y = 0);  
};
```

all of these are calls  
to c'tors with  
different syntaxes

```
void print(const Complex& c) {  
    cout << c.getReal() << "+" << c.getImag() << "i";  
}  
  
Complex c1;          // Complex::Complex()  
Complex c2(2.0, 1.0); // Complex::Complex(x,y)  
Complex c3(1.0);     // Complex::Complex(x,0)  
Complex c4 = 3.0;    // Complex::Complex(x,0)  
Complex* pc1 = new Complex; // Complex::Complex()  
Complex* pc2 = new Complex(); // Complex::Complex()  
Complex* pc3 = new Complex(-3.0, 1.0);  
                           // Complex::Complex(x,y)  
  
print(Complex(0.0, 1.0)); // Complex::Complex(x,y);
```

# Default Constructor

- ▶ A constructor which **receives no arguments** is called a **default constructor**
- ▶ The default constructor is called when an object is created with no arguments
- ▶ The compiler will **automatically generate a default constructor** for any class that **defines no constructors on its own**. This compiler-generated constructor:
  - ◆ Calls the default constructors of all members that are objects
  - ◆ Does **not** initialize members of built-in types (they will contain garbage)

```
Complex c1;  
print(Complex());
```

initializations using  
the default c'tor

# Arrays and Constructors

- When creating an array of objects, the **default constructor** is called for **each** of the elements

```
Complex array1[10];  
Complex* pArray = new Complex[10];  
Complex* pArray2 = new Complex[10](); // same as above
```

- If the class has no default constructor, **a compilation error** will occur. But how can this happen?
  - If the class **defines any constructors explicitly**, the compiler will **not** generate a default constructor for this class
    - In that case, can add a default constructor explicitly (if needed)
- Built-in types (only) can still be created without initialization
  - This exists for compatibility with C, and efficiency reasons

```
int* pArray = new int[10]; // uninitialized (contains garbage)  
int* pArray2 = new int[10](); // initialized to zeros
```

# Destructors

- ▶ In many cases a constructor will **allocate a resource**
  - ◆ E.g., allocate memory on the heap, open a file, open a network connection, ...
  - ◆ In such cases, the resource must be **freed** when the object is destroyed
- ▶ Each class has a special member function called a **destructor**
  - ◆ The destructor is **automatically** called when the object is destroyed

```
class Array {  
    int* data;  
    int size;  
  
public:  
    Array(int size);  
    ~Array();  
    int& atIndex(int index);  
};
```

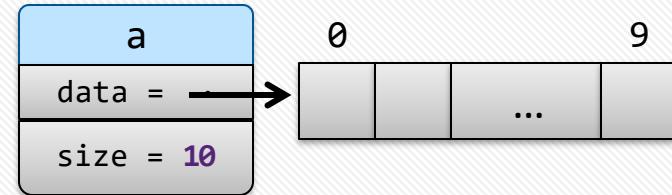
d'tor  
declaration

```
int main() {  
    Array a(10);  
    a.atIndex(3) = 2;  
    cout << a.atIndex(3) << endl;  
    return 0;  
}
```

a.**~Array()** is called

# Destructors

- ▶ Destructors take no arguments
  - ◆ Therefore, there can be **only one** per class
- ▶ The ~ sign at the beginning of the name comes from the logical **not** operator



```
array.h
class Array {
    int* data;
    int size;

public:
    Array(int size);
    ~Array();
    int& atIndex(int index);
};
```

```
array.cpp
Array::Array(int size) {
    data = new int[size];
    this->size = size;
}

Array::~Array() {
    delete[] data;
}

int& Array::atIndex(int index) {
    assert(index >= 0 && index < size);
    return data[index];
}
```

# Destructors

- ▶ A destructor is called whenever an object is released, for example:
  - ◆ When a **local** object **goes out of scope**
  - ◆ When a **dynamically allocated** object is **explicitly deleted**
  - ◆ When a **temporary value** is released at the "**end of the line**"

```
void f1(const Array& a);

void f2() {
    Array a(10);
    Array* ptr = new Array(20);
    Array* ptr2 = new Array(20);
    f1(Array(20));      // temporary Array => destructed "at the end of the line"
    delete ptr;         // calls ptr->~Array()
} // a.~Array() is called (goes out of scope)
// *ptr2 is Leaked!!
```

# Initialization List

- ▶ Lets look again at Array's constructor

- ◆ When are 'size' and 'data' **initialized**?

assignment  
(not initialization!)

```
Array::Array(int n) {  
    data = new int[n];  
    size = n;  
}
```

they were already created  
and initialized **here**  
(with garbage in this case)

- ▶ Whenever an object is created, first its **member variables are initialized**, and only then its **constructor's body is executed**
  - ◆ By default, all members are initialized with their **default constructors**
  - ◆ We can specify a different initialization with an **initialization list**

syntax for an explicit  
initialization list

```
Array::Array(int n) : data(new int[n]), size(n) {  
    // nothing left to do here (that's fine)  
}
```

custom initialization  
values

# Initialization List

- In most cases, using an initialization list or assigning values in a constructor will be essentially the same

```
Array::Array(int n) {  
    data = new int[n];  
    size = n;  
}
```

almost the same

```
Array::Array(int n) :  
    data(new int[n]), size(n)  
{}
```

- However, in some cases we **must use an initialization list**:

```
class BigInteger {  
    Array digits;  
    const int base;  
public:  
    BigInteger(int base, int len);  
    ~BigInteger();  
    // ...  
};
```

**digits** has no default c'tor

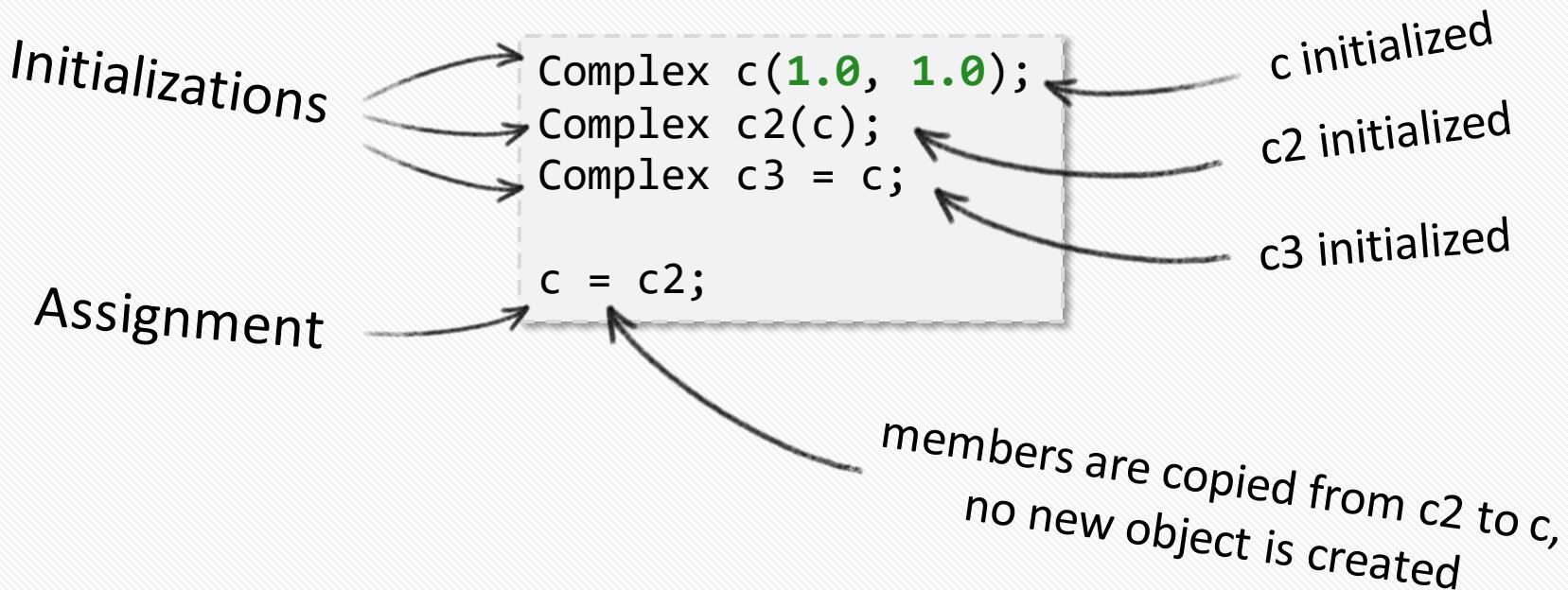
**base** cannot be assigned to

```
BigInteger::BigInteger(int base, int len) :  
    base(base), digits(len)  
{}
```

yah, this works

# Initialization vs. Assignment

- There is a critical difference between **initializing** an object and **assigning** a value to it:
  - Initialization occurs when a **new object** is created
  - Assignment modifies the values of an **existing object**



# Copy Constructor

- ▶ For a type **T**, the constructor that takes a single **const T&** as a parameter is called the **copy constructor**
- ▶ The copy constructor initializes a new object by copying an existing one

array.h

```
class Array {  
    int* data;  
    int size;  
  
public:  
    Array(int size);  
    Array(const Array&);  
    ~Array();  
};
```

array.cpp

```
Array::Array(const Array& arr) :  
    data(new int[arr.size]),  
    size(arr.size)  
{  
    for (int i = 0; i < size; ++i) {  
        data[i] = arr.data[i];  
    }  
}
```

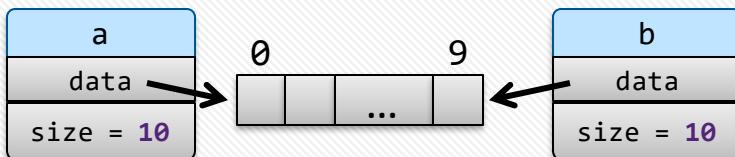
copy c'tors

```
main.cpp  
Array a(10);  
Array b(a);  
Array c = a;
```

# Copy Constructor

- ▶ The compiler will **automatically generate** a copy constructor for any class that does not define one
  - ◆ Even if other constructors have been defined!
  - ◆ This means that **every class has a copy constructor**
- ▶ The compiler-generated copy constructor simply calls the copy constructors of all class members
  - ◆ If the class uses pointers or allocates resources (such as dynamic memory), a **user-defined copy constructor is probably needed**

the **compiler-generated copy c'tor** for Array will look like this:



```
Array::Array(const Array& arr) :  
    data(arr.data),  
    size(arr.size)  
{ }
```

Uh-oh!

```
Array a(10);  
Array b = a;
```

# Copy Constructor

- The copy constructor is **extremely important** as it is called whenever an object is passed or returned from a function **by value**

'a' passed using copy c'tor

'a' passed by reference

'b' initialized to the  
temp variable value  
using a copy c'tor

```
int main() {  
    Array a(10)  
    f(a);  
    g(a);  
    Array b = h();  
  
    return 0;  
}
```

```
void f(Array a) { ... }
```

```
void g(Array& a) { ... }
```

```
Array h() {  
    return Array(10);  
}
```

return value is copied to a  
temporary variable using the  
copy c'tor

# Copy Constructor

- The compiler may decide, or be forced by the standard, to optimize and not create copies.

'a' passed using copy c'tor

'a' passed by reference

'b' initialized to the temp variable value using a copy c'tor

```
int main() {  
    Array a(10)  
    f(a);  
    g(a);  
    Array b = h();  
    return 0;  
}
```

```
void f(Array a) { ... }
```

```
void g(Array& a) { ... }
```

```
Array h() {  
    return Array(10);  
}
```

return value is copied to a temporary variable using the copy c'tor

\*This is called **copy elision** and it is beyond the scope of this course

# Initialization and Destruction Order

- When an object is **created**, its constructor first calls the constructors of all the class members, and only then executes the constructor's body
  - Members are constructed in the order they are defined in the class
- When an object is **destroyed**, it's destructor first executes the destructor's body, and then calls the destructors of all of the members
  - Member destructors are called in reverse order of creation

this is the only sensible order to do these things

```
class A {  
    // ...  
  
public:  
    A();  
    ~A();  
};
```

```
class B {  
    A a1, a2;  
  
public:  
    B() { ... }  
    ~B() { ... }  
};
```

B b;

First, **a1** is constructed using A::A()  
Next, **a2** is constructed using A::A()  
Finally, the code in { ... } is executed

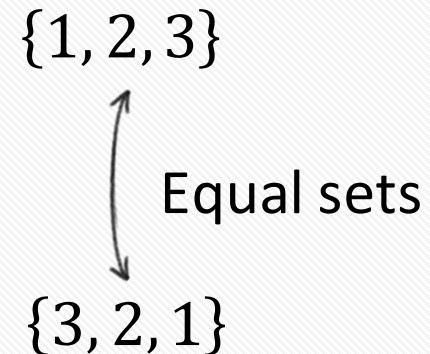
First, the code in { ... } is executed  
Then **a2** is destructed using A::~A()  
Finally **a1** is destructed using A::~A()

# Data Structures

- ▶ Consider a university database
  - ◆ A Student class represents a single student
  - ◆ How do we store **all the students** in the university?
  - ◆ An array of students?
- ▶ A simple array is not good enough
  - ◆ Has a **fixed size** (needs to be resized manually)
  - ◆ The same student could appear twice
- ▶ Idea: Create an ADT to store objects more conveniently
  - ◆ A **data structure** is an ADT for holding a collection of objects

# The Set ADT

- ▶ A set is a collection of elements, such that no duplicates are allowed
  - ◆ The main use of a set is to query whether an **element is in the set**
  - ◆ The **order** of the elements is **not important** in a set
- ▶ The interface of a set (minimum):
  - ◆ Add              ◆ Contains
  - ◆ Remove          ◆ Size
- ▶ We'll implement a set of **integers** for now (this will change...)



# Integer Set Class - Interface

set.h

```
#ifndef SET_H_
#define SET_H_
#include <string>

class Set {
public:
    Set();
    Set(const Set&);
    ~Set();

    bool add(int number);
    bool remove(int number);
    bool contains(int number) const;
    int getSize() const;

    Set& uniteWith(const Set&);
    Set& intersectWith(const Set&);

    std::string toString() const;
    ...
};

Set union(const Set&, const Set&);
Set intersection(const Set&, const Set&);

#endif /* SET_H_ */
```

Useful for  
debugging

external functions that  
operate on a Set, also  
part of the **interface**

# Integer Set Class - Usage

the sets are  
automatically  
destroyed

```
#include "set.h"
#include <iostream>
using std::cout;
using std::endl;

int main() {
    Set set1, set2;

    for (int j = 0; j < 20; j += 2) {
        set1.add(j);
    }

    for (int j = 0; j < 12; j += 3) {
        set2.add(j);
    }

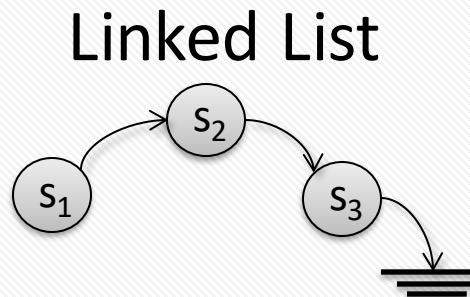
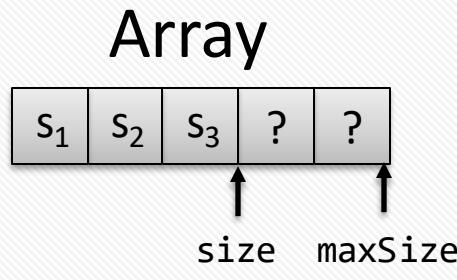
    Set unionSet = union(set1, set2);
    Set intersectionSet = intersection(set1, set2);

    cout << set1.toString() << endl;
    cout << set2.toString() << endl;
    cout << unionSet.toString() << endl;
    cout << intersectionSet.toString() << endl;
    return 0;
}
```

# Implementing a Set

## ▶ How can we implement our set?

- ◆ An array
  - ◆ Perhaps keep it sorted?
- ◆ A linked list
- ◆ Better options are taught in Data Structures



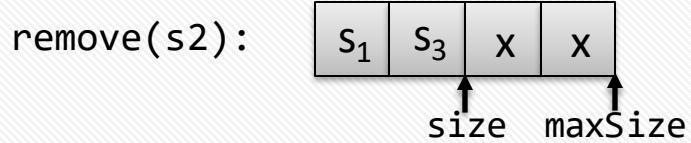
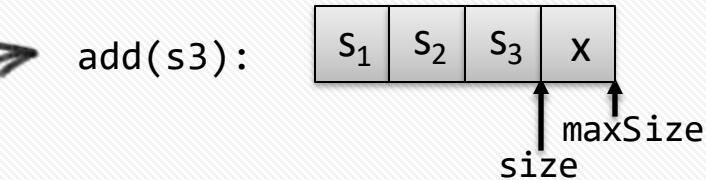
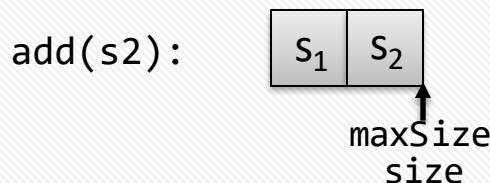
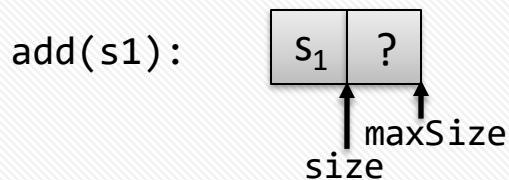
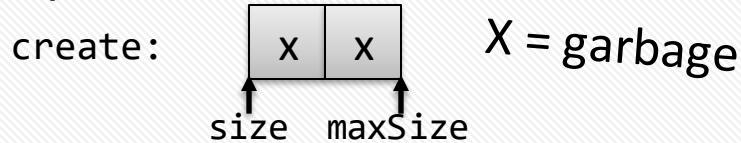
# Array Based Implementation

- ▶ Elements are stored consecutively in an array
- ▶ Limited capacity
  - ◆ However, the array can be **automatically resized** if more room is needed

Operation **complexities**  
depend on  
implementation



	Unordered	Ordered
Lookup	$O(n)$	$O(\log n)$
Insert	$O(n)$	$O(n)$
Remove	$O(n)$	$O(n)$



# Implementing the Set

- ▶ If the set is implemented as an **ADT**, its implementation **can be changed** later
- ▶ Sets are usually implemented using **more advanced** data structures
  - ◆ Typically hash tables or balanced trees
    - Not in this course
- ▶ Here we will implement the set using a simple unordered array

# Integer Set Class - Implementation

set.h

```
class Set {  
    ...  
private:  
    int* data;      // the elements of the set  
    int size;       // the current number of elements in the set  
    int maxSize;   // the allocated size of the array  
  
    int find(int number) const;  
    void expand();  
  
    /** The initial size of the set array */  
    static const int INITIAL_SIZE = 10;  
  
    /** The factor by which to expand the array when needed */  
    static const int EXPAND_RATE = 2;  
  
    /** Returned by 'find' when an element is not in the set */  
    static const int NUMBER_NOT_FOUND = -1;  
};
```

# set.cpp

the default constructor

```
#include "set.h"

Set::Set() :
    data(new int[INITIAL_SIZE]),
    size(0),
    maxSize(INITIAL_SIZE)
{ }
```

the body is empty, this  
is quite common

# set.cpp

always better to  
use a public  
function when  
possible

copy c'tor

```
Set::Set(const Set& set) :  
    data(new int[max(set.getSize(), INITIAL_SIZE)]),  
    size(set.getSize()),  
    maxSize(max(set.getSize(), INITIAL_SIZE)),  
{  
    for(int i = 0; i < size; ++i) {  
        data[i] = set.data[i];  
    }  
}
```

d'tor

```
Set::~Set() {  
    delete[] data;  
}
```

# Set::contains()

```
int Set::find(int number) const {
    for(int i = 0; i < size; ++i) {
        if (data[i] == number) {
            return i;
        }
    }
    return NUMBER_NOT_FOUND;
}
```

```
bool Set::contains(int number) const {
    return find(number) != NUMBER_NOT_FOUND;
}
```

```
int Set::getSize() const {
    return size;
}
```

C++ has a built-in  
**bool** type (no need  
for #include)



# Set::add()

increase maxSize and  
the allocated array,  
keeping the elements

```
bool Set::add(int number) {  
    if (contains(number)) {  
        return false;  
    }  
    if (size >= maxSize) {  
        expand();  
    }  
  
    data[size++] = number;  
    return true;  
}
```

failed memory allocations are handled  
with the "magic" of **exceptions**  
\*more on this when we learn exceptions

# Set::expand() Helper Function

```
void Set::expand() {  
    int newSize = maxSize * EXPAND_RATE;  
    int* newData = new int[newSize]; ←  
    for (int i = 0; i < size; ++i) {  
        newData[i] = data[i];  
    }  
    delete[] data;  
    data = newData;  
    maxSize = newSize;  
}
```

the correct solution is  
to use a vector  
\*more on this in the STL tutorial

# Set::remove()

```
bool Set::remove(int number) {
    int index = find(number);
    if (index == NUMBER_NOT_FOUND) {
        return false;
    }

    data[index] = data[--size];
    return true;
}
```

a good implementation  
should also shrink the array

# Reusability

- ▶ The Set is expected to be **reused**
- ▶ Should we supply other functions?
  - ◆ Equals
  - ◆ Contains (subset)
  - ◆ ...
- ▶ No clear answer:
  - ◆ A **fat interface** makes it **harder to maintain**
  - ◆ A (too) **thin interface** makes the user **duplicate code**

# Union and Intersection

```
Set& Set::uniteWith(const Set& other) {  
    for (int i = 0; i < other.getSize(); ++i) {  
        add(other.data[i]);  
    }  
    return *this;  
}
```



use **public functions**  
whenever possible

```
Set& Set::intersectWith(const Set& other) {  
    for (int i = 0; i < this->getSize(); ++i) {  
        if (!other.contains(data[i])) {  
            remove(data[i]);  
        }  
    }  
    return *this;  
}
```



# External Set Functions

- ▶ We implement **union** and **intersection** as external functions (not members of class Set)
- ▶ **Why?**
  1. We prefer the syntax **s3 = union(s1,s2)** over `s3 = s1.union(s2)`
  2. External functions **do not access the private section**
    - Easier to maintain, and fewer potential bugs

```
Set union(const Set& set1,  
          const Set& set2) {  
  
    Set result = set1;  
  
    return result.uniteWith(set2);  
}
```

```
Set intersection(const Set& set1,  
                 const Set& set2) {  
  
    Set result = set1;  
  
    return result.intersectWith(set2);  
}
```

return by value  
(copy c'tor)

*“Within C++, there is a much smaller and cleaner language struggling to get out.”*

- Bjarne Stroustrup

# Operator Overloading

User-Defined Operators and  
Conversions

# Operator Overloading

- ▶ One of the goals of C++ is to allow user-defined types to be as convenient as built-in types

```
void f1(double d1) {  
    double d2 = 2.0;  
    double input;  
  
    cin >> input;  
  
    cout << (d1 + d2 * input);  
}
```

convenient

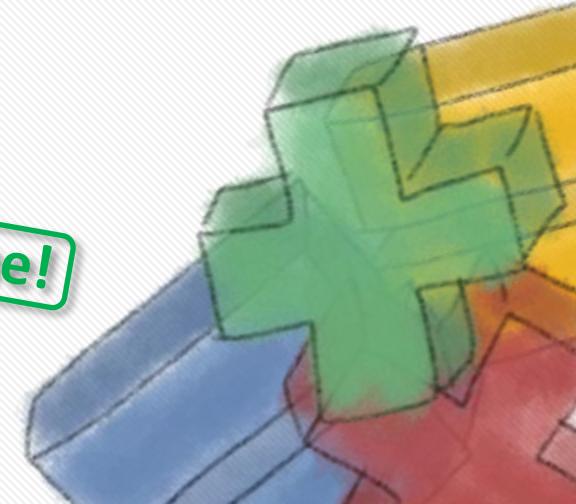
```
void f2(Complex c1) {  
    Complex c2(3.0, 2.0);  
    Complex input;  
    input.readFrom(cin);  
    c2.multiply(input);  
    c1.add(c2);  
    c2.printTo(cout);  
}
```

annoying

- ▶ For that purpose, C++ allows defining **operators** (such as + or =) for user-defined types

```
void f2(Complex c1) {  
    Complex c2(3.0, 2.0);  
    Complex input;  
    cin >> input;  
    cout << (c1 + c2 * input);  
}
```

awesome!



# Operator Overloading

- ▶ Defining operators is done using **operator overloading**
  - ◆ C++ treats an operator like a function
  - ◆ Operators can be **overloaded** the same way functions can

**operator+** is just a function name  
(incidentally, the function called for the operator '+')

```
Complex operator+(const Complex& x, const Complex& y) {  
    Complex result(x.real() + y.real(), x.imag() + y.imag());  
    return result;  
}
```

Complex c1, c2;  
c1 + c2



# What Can We Do?

- ▶ We can overload all of the following operators:

+	-	*	/	%	^	&
	~	!	=	<	>	+ =
- =	* =	/ =	% =	^ =	& =	=
<<	>>	>> =	<< =	==	!=	< =
> =	&&		++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

we won't get to these  
in this course

# What Can't We Do?

## ► We cannot:

- ◆ Overload these operators:

`:: . .* ? : sizeof typeid`

we'll see what  
this is later

- ◆ Define **new operators**

- So “ $a^{**}b$ ” for a power operator cannot be defined

- ◆ Change the **number of parameters**, **precedence** or **associativity** of existing operators

- So “ $a + b * c$ ” always means “ $a + (b * c)$ ”

- ◆ An overloaded operator must have **at least one parameter** which is a **user-defined type** (a class, struct or enum)

# Binary Operators

- ▶ A binary operator can be defined in one of two ways:
  - ◆ As a **member** function taking **one argument**
  - ◆ As a **nonmember** function taking **two arguments**
- ▶ For any binary operator **@**, the compiler interprets **a@b** as either **a.operator@(b)** or **operator@(a,b)**

```
class Complex {  
    double re, im;  
public:  
    //...  
    Complex operator+(const Complex& x) const;  
};
```

member

```
Complex operator+(const Complex& x, const Complex& y);
```

nonmember

compilation:

```
Complex c1, c2, c3  
c3 = c1+c2;
```

or

```
c3 = c1.operator+(c2);
```

```
c3 = operator+(c1,c2);
```

# Unary Operators

- ▶ A unary operator can be defined in one of two ways:
  - ◆ As a **member** function taking **no arguments**
  - ◆ As a **nonmember** function taking **one argument**
- ▶ For any prefix (left-side) unary operator **@**, the compiler interprets **@a** as either **a.operator@()** or **operator@(a)**
  - ◆ The *postfix* `++` and `--` operators are treated a bit differently (more later)

```
class Complex {  
    double re, im;  
public:  
    //...  
    Complex operator-() const;  
};
```

```
Complex operator-(const Complex& x);
```

↑ **nonmember**

compilation:

```
Complex c1, c2  
c2 = -c1;
```

or

```
c2 = c1.operator-();
```

```
c2 = operator-(c1);
```

# Other Operators

The following operators can only be overloaded  
**as member functions** (not external functions):

=        []        ()        ->

# A Rational Number Class

- ▶ A **rational number** is a number that can be expressed as a fraction  $a/b$  of two integers
- ▶ We will use operator overloading to create a rational number class that behaves **similarly to a built-in type**
  - ◆ We will keep the number in simplified form
    - The **numerator** and **denominator** will be co-prime
    - The **denominator** will be a positive number
  - ◆ The only mathematical background needed for this class is:
    - Acquaintance with the Euclidean algorithm for finding the **GCD** (greatest common divisor) of two positive integers

# A Rational Number Class

```
class Rational {  
    int numerator, denominator;  
  
    void simplify();  
    static int gcd(int x, int y);  
public:  
    Rational(int numerator = 0,  
            int denominator = 1);  
    //... functions + operators ...  
};
```

```
Rational r;  
Rational r1(2);  
Rational r2(6, 10);  
Rational r3(3, 5);
```

converts this object to simplified form

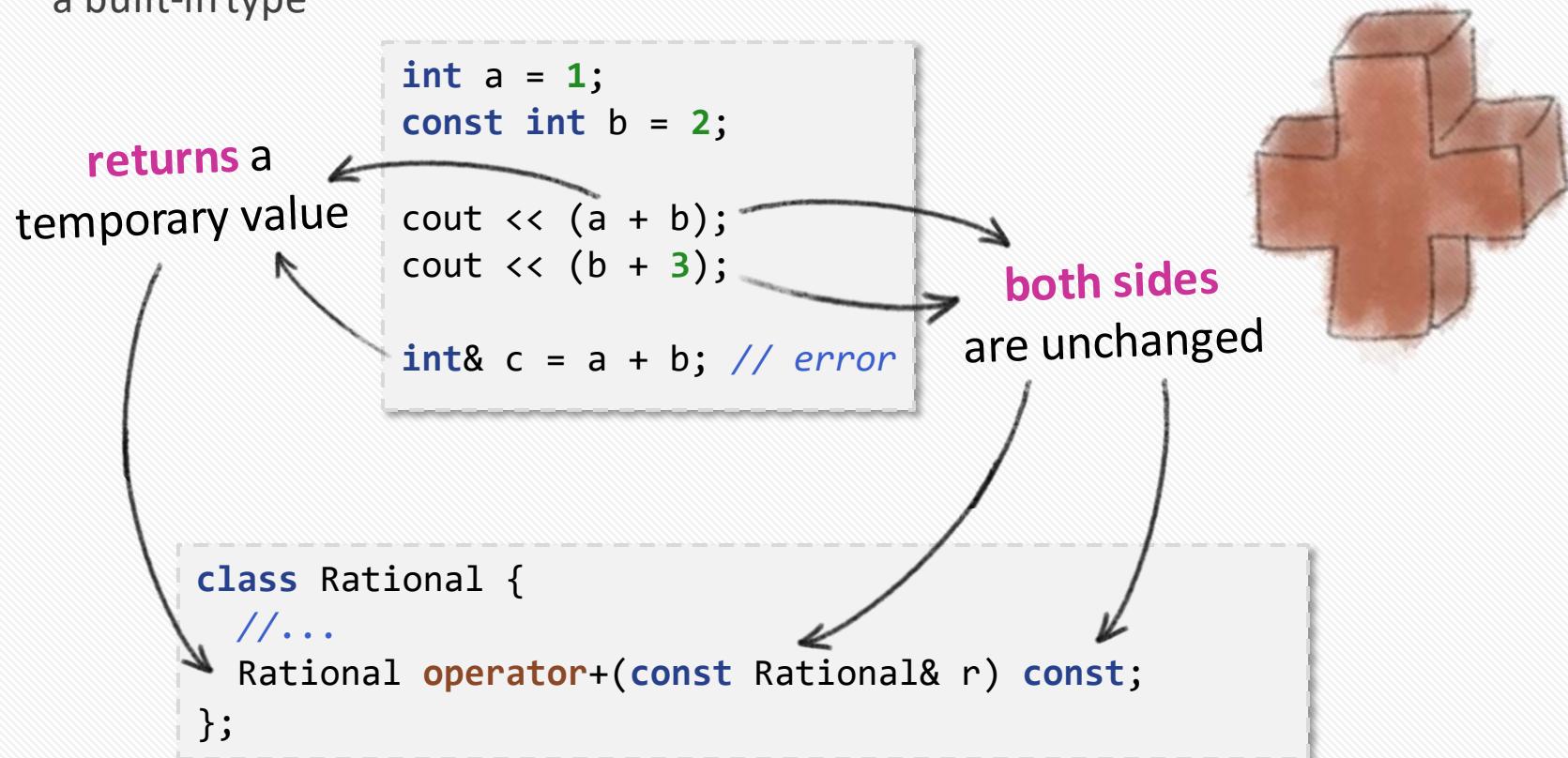
```
void Rational::simplify() {  
    if (denominator < 0) {  
        denominator = -denominator;  
        numerator = -numerator;  
    }  
  
    int common = gcd(numerator, denominator);  
    if (common != 0) {  
        numerator /= common;  
        denominator /= common;  
    }  
}
```

```
Rational::Rational(int numerator, int denominator) :  
    numerator(numerator), denominator(denominator) {  
  
    if (denominator == 0) {  
        throw DivisionByZero();  
    }  
  
    simplify();  
}
```

we'll get back to this  
when we learn  
exceptions

# Operators + and +=

- Let us start by overloading operator+ and operator +=
- What should the declaration for each of those be?
  - To answer that, we need to look at how operator+ and operator+= behave for a built-in type



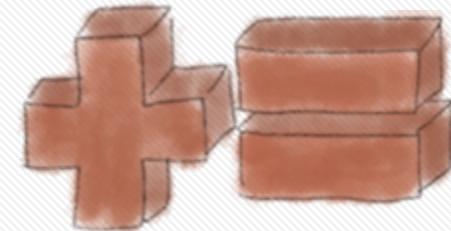
# Operators + and +=

- Let us start by overloading operator+ and operator +=
- What should the declaration for each of those be?
  - To answer that, we need to look at how operator+ and operator+= behave for a built-in type

**return value** is not a temporary!  
it's the first argument itself

```
int a = 1;  
const int b = 2;  
  
cout << (a += b);  
cout << (a += 3);  
  
(a += b) += 2;  
b += 2; // error
```

**left-hand side** is not const



**right-hand side**  
is unchanged

```
class Rational {  
//...  
Rational& operator+=(const Rational& r);  
};
```

```
Rational r1, r2, r3;  
(r1 += r2) += r3;
```

# Operators + and +=

- ▶ We can now **implement** our operators, however:
  - ◆ It seems + and += should have a similar implementation
  - ◆ How can we avoid **code duplication**?
- ▶ Somewhat surprisingly, **+ is more complex to implement than +=**
  - ◆ Operator+ involves 3 objects, while += only involves 2
  - ◆ Therefore, **we implement + using +=**

```
Rational& Rational::operator+=(const Rational& r) {  
    numerator = numerator * r.denominator + r.numerator * denominator;  
    denominator = denominator * r.denominator;  
    simplify();  
    return *this;  
}
```

```
Rational Rational::operator+(const Rational& r) const {  
    Rational result(*this);  
    return (result += r);  
}
```

```
Rational r1, r2, r3;  
(r1 += r2) += r3;  
cout << r1 + r2;
```

# A Rational Number Class

- ▶ Operators such as \* and \*= are implemented similarly
- ▶ Operator - (minus) has **two versions**:
  - ◆ A **binary** operator- , as in “a - b ”
  - ◆ A **unary** operator- , as in “-a ”

```
class Rational {  
    //...  
    Rational& operator=(const Rational& r);  
    Rational operator-(const Rational& r) const;  
    Rational operator-() const;  
};
```

```
Rational Rational::operator-() const {  
    return Rational(-numerator, denominator);  
}
```

```
Rational& Rational::operator=(const Rational& r) {  
    return *this += -r;  
}
```

```
Rational Rational::operator-(const Rational& r) const {  
    return Rational(*this) -= r;  
}
```

# Type Conversions

- ▶ C++ allows **user-defined conversions** between types
  - ◆ At least one of the types in the conversion must be user-defined
  - ◆ If a conversion is available, the compiler can **implicitly cast function arguments** when needed

```
Rational r1(1, 2);
int n = 1;
Rational r2 = r1 + n;
```

'n' is implicitly cast  
from int to Rational  
(as n/1)

- ▶ Conversions can be defined in two ways:
  - ◆ By a **constructor** which takes one argument
  - ◆ By overloading a **conversion operator**

# Conversion by Constructor

- ▶ A constructor which **takes a single argument** defines a conversion:

```
Rational::Rational(int numerator) : ←  
    numerator(numerator), denominator(1) {  
    simplify();  
}
```

convert an int  
into a Rational

- ▶ A constructor that can be called with a single argument using **default values** also defines a conversion:

```
Rational::Rational(int numerator, int denominator = 1) :  
    numerator(numerator), denominator(denominator) {  
  
    if (denominator == 0) {  
        throw DivisionByZero();  
    }  
  
    simplify();  
}
```

# Conversion by Constructor

- Let's analyze the following call to operator+:

```
int n = 2;  
Rational r2 = r1 + n;
```

1. Compiler searches for an exact match:  
`Rational::operator+(int)`

no such  
function exists

2. Compiler searches for a function of the form  
`Rational::operator+(T)`  
where `T` is a type that `int` can be converted to

3. Compiler uses the function  
`Rational::operator+(const Rational&)`

`int -> Rational`  
conversion is available  
via

code is  
implicitly  
changed to

`Rational::Rational(int)`  
`Rational r2 = r1 + Rational(n);`

# Conversion Operator

- ▶ Sometimes we will want to create a conversion **from a new type to an existing type**
  - ◆ We cannot always modify the existing type (it may be built-in!)
- ▶ A **conversion operator** may be defined using the following syntax:

**no return type**  
(it is taken from the operator's name)

```
class Rational {  
    //...  
    operator double() const;  
};
```

```
Rational::operator double() const {  
    return double(numerator) / denominator;  
}
```

- ▶ One common use of this feature is to define a conversion to type **bool**, so objects can be used as conditions:

converts to **true** if all  
input operations  
completed  
successfully

```
int input;  
cin >> input;  
if (cin) {  
    cout << "number read successfully";  
}
```

# Explicit Conversions

- ▶ **Implicit conversions** are usually a bad idea

- ◆ Code might become **ambiguous**:

just for the example!

```
class Rational {  
    //...  
    Rational(double val);  
    operator double() const;  
};  
  
Rational r(1, 2);  
cout << r + 1.2;
```

ambiguous code  
does not compile

compiler can't decide  
between:

1. convert **1.2** to Rational
2. convert **r** to double

- ◆ Nasty **bugs will compile**:

```
class Array {  
    // ...  
    Array(int size);  
};
```

```
void f(Array& a, int i) {  
    a = 5; // converts 5 to a temporary empty array  
           // of length 5, and assigns it to a.  
           // programmer probably meant a[i] = 5;  
}
```

# Explicit Conversions

- To **prevent the compiler** from converting values automatically via a constructor or conversion operator, we can declare them as **explicit**

```
class Array {  
    //...  
    explicit Array(int size);  
};
```

```
void f(Array& a, int n) {  
    a = n;  
}
```

```
class Rational {  
    //...  
    Rational(int num = 0, int denom = 1);  
    explicit operator double() const;  
};
```

does not compile anymore. if you really meant it, use

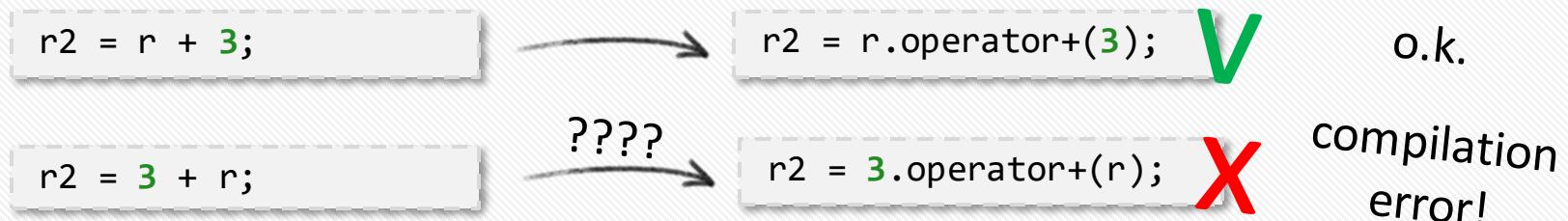
```
Rational r2 = r1 + 1;      // still compiles! Rational(int) is not explicit  
double x = cos(r1);       // will not compile (great!), must use cos((double)r1)  
cout << double(r) + 1.2; // this compiles
```

```
void f(Array& a, int n) {  
    a = Array(n);  
}
```

- Good rule of thumb:** always declare a constructor taking a single parameter as **explicit**, unless a conversion really makes sense

# Conversions and Member Functions

- Our operator+ has one problem...



- This means that our + operator for **Rationals** has **asymmetrical behavior**
  - We cannot switch the addition order!
- To solve this, we define operator+ as a **non-member** (ordinary) function:

```
Rational operator+(const Rational& r1, const Rational& r2) {  
    Rational result = r1;  
    result += r2;  
    return result;  
}
```

- Now **both sides** of the + can be converted if necessary, not just the right side

# Friends

- ▶ Lets add an **== operator**:

- ◆ operator== should return a **bool**
- ◆ operator== should not change any of its arguments
- ◆ We would like to have symmetric behavior, so it should be a **non-member**

```
Rational r1, r2;  
if (r1 == r2) {  
    ...  
}
```

```
bool operator==(const Rational& r1, const Rational& r2) {  
    return r1.numerator == r2.numerator && r1.denominator == r2.denominator;  
}
```

**Oops!** Need to access private members

- ▶ Using a **friend declaration** we can grant an **external function** access to the private section of a class

```
class Rational {  
    //...  
    friend bool operator==(const Rational& r1, const Rational& r2);  
};
```

# Friends

- ▶ A friend declaration can appear **in any part of the class** (public or private) – it doesn't matter
  - ◆ In practice, we'll usually place it in the **private section** since it's an implementation detail

```
class Rational {  
public:  
    friend void f(const Rational&);  
};
```

```
class Rational {  
private:  
    friend void f(const Rational&);  
};
```

equivalent

- ▶ Writing a friend function declaration:
  - ◆ **Grants private access** to the external function
  - ◆ **Declares the existence** of the function, so it can be used without any additional forward declaration
    - ◆ But... only if the function takes at least one parameter of the class type. If not, we'll still need a normal forward declaration for the function

← annoying  
technicality  
:(

# Overloading Input/Output Operators

- ▶ The `<<` and `>>` operators are used in C++ for **I/O operations**
  - ◆ We can overload them to support **user-defined types**
- ▶ Both operators must be defined as **non-members**
  - ◆ Their first argument is an I/O stream, not an object of the class

```
must return the  
ostream to  
allow chaining  
std::ostream& operator<<(std::ostream& os, const Rational& r) {  
    os << r.numerator;  
    if (r.denominator != 1) {  
        os << "/" << r.denominator;  
    }  
    return os;  
}
```

#include <iostream>

cout and cerr are  
both of type ostream

```
cout << r << "123" << r1;
```

```
class Rational {  
    // ...  
    friend std::ostream& operator<<(std::ostream& os, const Rational& r);  
};
```

# The Final Rational Class (1/2)

rational.h

```
class Rational {  
  
public:  
    Rational(int numerator = 0, int denominator = 1);  
    explicit operator double() const;  
  
    Rational operator-() const;  
    Rational& operator+=(const Rational&);  
    Rational& operator-=(const Rational&);  
    Rational& operator*=(const Rational&);  
    Rational& operator/=(const Rational&);  
  
private:  
    int numerator, denominator;  
    void simplify();  
    static int gcd(int x, int y);  
  
    friend bool operator==(const Rational&, const Rational&);  
    friend bool operator<(const Rational&, const Rational&);  
    friend std::ostream& operator<<(std::ostream&, const Rational&);  
};  
  
...
```

*prefer public  
before private*

# The Final Rational Class (2/2)

all of these can be implemented outside the class **using only its interface**



avoid using **friend** declarations unless necessary

...

rational.h

```
Rational operator+(const Rational&, const Rational&);  
Rational operator-(const Rational&, const Rational&);  
Rational operator*(const Rational&, const Rational&);  
Rational operator/(const Rational&, const Rational&);  
  
bool operator!=(const Rational&, const Rational&);  
bool operator>=(const Rational&, const Rational&);  
bool operator>(const Rational&, const Rational&);  
bool operator<=(const Rational&, const Rational&);
```

# Assignment

- ▶ There is a difference between **copying** and **assignment**

Handled by copy  
constructor

And here ??

```
int main() {  
    Set set1;  
  
    for (int j = 0; j < 20; j += 2) {  
        set1.add(j);  
    }  
  
    Set set2 = set1;  
  
    set2 = set1;  
  
    return 0;  
}
```

# Assignment Operator

- An assignment operator= is automatically generated by the compiler
  - The compiler-generated operator= calls operator= for each of the members of the class, i.e.

```
Rational& Rational::operator=(const Rational& r) {  
    numerator = r.numerator;  
    denominator = r.denominator;  
    return *this;  
}
```

} compiler  
generated

- In some cases, the compiler-generated operator= **will not work** as intended
  - Usually happens when the class manages a resource

**Uh-oh!** Both sets now  
share the same array

```
Set& Set::operator=(const Set& s) {  
    size = s.size;  
    maxSize = s.maxSize;  
    data = s.data;  
    return *this;  
}
```

} compiler  
generated

# Assignment Operator

- To solve it we need to **overload the assignment operator** of a class
  - This should be done only when the compiler-generated one is insufficient
  - An assignment operator should **release old resources**, and **allocate new ones**
  - We should also watch out for the case of self-assignment

allow chaining

```
Set s1, s2, s3;  
// ...  
s1 = s2 = s3;  
(s1 = s2).uniteWith(s3)
```

```
Set& Set::operator=(const Set& s) {  
    if (this == &s) {  
        return *this;  
    }  
    delete[] data;  
    data = new int[max(INITIAL_SIZE, s.getSize())];  
    size = getSize();  
    maxSize = max(INITIAL_SIZE, s.getSize());  
    for (int i = 0; i < size; i++) {  
        data[i] = s.data[i]  
    }  
    return *this;  
}
```

avoids a bug in  
case of self-  
assignment

```
Set s1;  
// ...  
s1 = s1;
```

# The Big Three

- ▶ These three functions typically **go together**:
  - ◆ Copy constructor
  - ◆ Destructor
  - ◆ Operator=
- ▶ If you need to write **one**, you probably **need all three**
- ▶ Intuitive reasoning:

**Assignment = Destructor + Copy C'tor**

# Control of Default Class Methods

- ▶ We have seen that the compiler automatically generates several methods for us
  - ◆ A **default constructor** is generated when no other constructor is explicitly defined
  - ◆ A **copy constructor** and **assignment operator** are **always generated** if the user does not explicitly define them
- ▶ This behavior **may not be what we want**

```
class Person {  
    std::string name;  
    // ...  
public:  
    Person(std::string name);  
};
```

```
class WebServer {  
    // ...  
public:  
    WebServer(std::string homepage, int port);  
    void start(int max_clients);  
};
```

WebServer w2 = w1; // no good

We **would like** a default c'tor, but the compiler will not generate one for us

We **do not want** this class to be copyable or assignable, but the compiler generates these anyway

# Control of Default Class Methods

- ▶ We can use **=default** to direct the compiler to generate a method that it would not normally generate

```
class Person {  
    std::string name;  
    // ...  
public:  
    Person() = default;  
    Person(std::string name);  
};
```

tells the compiler to generate  
the default c'tor even though  
another c'tor is defined

- ▶ Using **=default** is much better than **implementing the default behavior ourselves**

☞ fewer possible bugs – if a field is added or removed, we don't have to remember to update the default c'tor **manually**

# Control of Default Class Methods

- ▶ What if we don't want our class to be copyable or assignable at all?
  - In some cases, these operations have **no reasonable meaning**

```
class WebServer {  
    // ...  
public:  
    WebServer(const WebServer&) = delete;  
    void operator=(const WebServer&) = delete;  
};
```



Trying to copy or assign will **always** be a compiler error. Also, we can place the =delete in the public section, **as part of the interface**.

# Control of Default Class Methods

- We can also use **=default** to emphasize that we did not forget to implement a method

```
class WebClient {  
    // ...  
public:  
    WebClient(std::string host, int port);  
    void connect();  
};
```

```
WebClient(const WebClient&) = default;  
WebClient& operator=(const WebClient&) = default;
```

The compiler will generate a copy c'tor and operator= for this class. But, did the programmer really **mean for this**, or did he just **forget** to implement them?

Adding these lines to the public section is the best way to **indicate our meaning**

- Always indicate constructors and assignment operators with **=default** if you want the compiler to generate their default behavior

except for simple classes

# Handling Errors in Operators

- In the (rare) event than an operator fails, we will typically handle it with **exceptions**
  - They can't return an error value due to their syntax

```
class DivisionByZero {};
```



empty class representing  
a specific error

```
Rational& Rational::operator/=(const Rational& r) {  
  
    if (r.numerator == 0) {  
        throw DivisionByZero(); // nothing else to do  
    }  
  
    *this *= Rational(r.denominator, r.numerator);  
    return *this;  
}
```

```
Rational mean(const Rational* r, int n) {  
    Rational sum = 0;  
    for (int i = 0; i < n; ++i) {  
        sum += r[i];  
    }  
    sum /= n; // throws if n==0  
    return sum;  
}
```

# Indexing Operator

- ▶ The **indexing operator []** can be overloaded to create classes that behave like arrays
  - ◆ The parameter does not have to be of type int
- ▶ The indexing operator **must be defined as a member function**

```
class Array {  
public:  
    explicit Array(int n);  
    Array(const Array&);  
    ~Array();  
    Array& operator=(const Array&);  
    int size() const;  
    int& operator[](int index);  
  
private:  
    int* data;  
    int size;  
};
```

```
int& Array::operator[](int index) {  
    assert(index >= 0 && index < size);  
    return data[index];  
}
```

```
Array& doubleIt(Array& a) {  
    for (int i = 0; i < a.size(); i++) {  
        a[i] = 2 * a[i];  
    }  
    return a;  
}
```

←  
external function

# Indexing Operator

- ▶ Functions which **return a reference**, such as our **operator[]**, pose a problem:

uh oh...  
the call `a[i]` will **not**  
**compile** because  
`operator[]` is not `const`

```
Array arrayMult(const Array& a, int val) {  
    Array result(a.size());  
    for (int i = 0; i < a.size(); i++) {  
        result[i] = a[i] * val;  
    }  
    return result;  
}
```

# Indexing Operator

- ▶ We will need **two versions** of operator[]:

- ◆ One for **regular** objects
- ◆ One for **const** objects

```
class Array {  
public:  
    const int& operator[](int index) const;  
    int& operator[](int index);  
    // ...  
private:  
    int* data;  
    int size;  
};
```

make sure this function actually  
**protects** the object from changing  
\*even from **outside** the function!

```
const int& Array::operator[](int index) const {  
    assert(index >= 0 && index < size);  
    return data[index];  
}
```

```
int& Array::operator[](int index) {  
    assert(index >= 0 && index < size);  
    return data[index];  
}
```

# Example: Enumeration of a Set

- ▶ For most data structures, we need to offer a way to **get all the elements** in them
- ▶ This is called **iterating** over the data structure
  - ◆ e.g. “for each student, calculate their average”

iterating over the  
set's elements  
should be supported

```
#include "set.h"                                main.cpp

int main() {
    Set set;
    for (int j = 0; j < 20; j += 2) {
        set.add(j);
    }

    int sum = 0;
    for (every value in set) {
        sum += value;
    }

    // ...
}
```

# Iterators

- ▶ One way could be to provide **two pointers**:
  - ◆ **begin()** will return a pointer to the first element
  - ◆ **end()** will return a pointer to “one after the last” element

```
class Set {  
    // ...  
public:  
    // ...  
    const int* begin() const;  
    const int* end() const;  
};
```

```
const int* Set::begin() const {  
    return data;  
}
```

```
const int* Set::end() const {  
    return data + size;  
}
```

pointer to the  
“end” of the  
set

```
void setSum(const Set& set) {  
    int sum = 0;  
    for (const int* it = set.begin(); it != set.end(); ++it) {  
        sum += *it;  
    }  
    cout << "sum of set is " << sum << endl;  
}
```

using the  
iterator

# Iterators

- ▶ Returning a pointer is **implementation-dependent** and not always applicable
  - ◆ What if our set were implemented with a linked list?
- ▶ However, the syntax we get with begin() and end() is **simple** and **useful**
- ▶ Using operator overloading we can enjoy the **best of both worlds**
  - ◆ We can **create sophisticated iterators** which will be used **just like pointers** but will **hide the internal implementation**

# Iterators

- To achieve this, we must replace the simple pointer iterator with a **custom class** that will be returned by begin() and end()

note that Set::Iterator is public, and thus part of the **interface** of Set

```
class Set {  
public:  
    // ...  
    class Iterator;  
    Iterator begin() const;  
    Iterator end() const;  
private:  
    // ...  
};
```

declare a class Set::Iterator

yes, we can do that

# Minimal Operators for an Iterator

- ▶ Any iterator in C++ must support **at least** the following three operators

- So it can be used like a pointer

**operator!=()**  
to compare two  
iterators

```
void setSum(const Set& s) {  
    int sum = 0;  
    for (Set::Iterator it = s.begin(); it != s.end(); ++it) {  
        sum += *it;  
    }  
    cout << "sum of set is " << sum << endl;  
}
```

**operator\*()**  
to return the object  
currently pointed to

**prefix operator++()**  
to advance the iterator

# Optional Operators for an Iterator

- In addition, we may wish to support **all or some** of the following operators, which mimic the behavior of pointers:

**operator==()**  
for comparing iterators

**postfix operator++()**  
for advancing the iterator

**copy c'tor and operator=()**  
(note that `it2` initially points to the same  
location as `it1`, but it moves  
independently)

**prefix/postfix operator--()**  
for moving the iterator  
backwards

```
void func(const Set& s) {  
    Set::Iterator it1 = s.begin();  
    if (it1 == s.end()) {  
        return;  
    }  
    cout << "first = " << *it1 << endl;  
    it1++;  
    Set::Iterator it2 = it1;  
    cout << "second = " << *it2 << endl;  
    it1--;  
    cout << "first again = " << *it1 << endl;  
    cout << "second again = " << *it2 << endl;  
}
```

→ iterators that supports this  
are called a **bidirectional** iterators  
(we will not implement these here)

# Set::Iterator Interface

```
class Set {  
public:  
    // ...  
    class Iterator;  
  
    Iterator begin() const;  
    Iterator end() const;  
  
private:  
    int* data;  
    int size;  
    int maxSize;  
    // ...  
};
```

set.h

this appears **after** class Set

```
class Set::Iterator {  
    const Set* set;  
    int index;  
    Iterator(const Set* set, int index);  
    friend class Set;  
public:  
    const int& operator*() const;  
    Iterator& operator++();  
    Iterator operator++(int);  
  
    bool operator==(const Iterator& it) const;  
    bool operator!=(const Iterator& it) const;  
  
    Iterator(const Iterator&) = default;  
    Iterator& operator=(const Iterator&) = default;  
};
```

later in set.h

# Set::Iterator Interface

why is the c'tor  
private?

```
class Set::Iterator {  
    const Set* set;      // the set this iterator points to  
    int index;          // the current index in the set  
    Iterator(const Set* set, int index);  
    friend class Set;   // allow Set to call the c'tor  
public:  
    const int& operator*() const;  
    Iterator& operator++();      // prefix (++it)  
    Iterator operator++(int);    // postfix (it++)  
    // ...  
};
```

so no one except Set can  
create a Set::Iterator  
(careless invocations of this  
c'tor can break the set)

```
Set s1;  
Set::Iterator it(&s1, 3);  
Set::Iterator it2 = it;
```

Does this compile ?

# Set::Iterator Interface

Set may access the private members of Set::Iterator



allows Set to call Iterator's private constructor  
(Set::begin() and Set::end()  
require this to create an Iterator)

```
class Set::Iterator {  
    const Set* set;      // the set this iterator points to  
    int index;          // the current index in the set  
    Iterator(const Set* set, int index);  
    friend class Set;   // allow Set to call the c'tor  
public:  
    const int& operator*() const;  
    Iterator& operator++();      // prefix ++  
    Iterator operator++(int);   // postfix ++  
    // ...  
};
```

\*In some cases, the usage of friends may indicate bad design. However, never replace a friend declaration by **simply making things public**

# Implementing Set::Iterator

set.cpp

```
Set::Iterator::Iterator(const Set* set, int index) :  
    set(set),  
    index(index)  
{ }
```

```
const int& Set::Iterator::operator*() const {  
    assert(index >= 0 && index < set->getSize());  
    return set->data[index];  
}
```



**Set::Iterator** is inside the Set class, so it has  
access to the private section of Set

# Implementing Set::Iterator

note the difference in return  
type of `++i` and `i++`

```
Set::Iterator& Set::Iterator::operator++() {  
    ++index;  
    return *this;  
}
```

```
Set::Iterator Set::Iterator::operator++(int) {  
    Iterator result = *this;  
    **this;  
    return result;  
}
```

```
Set::Iterator it1 = s.begin();  
it2 = it1++;  
it2 = ++it1;
```

the "dummy" int parameter tells  
the compiler this is a postfix ++

# Implementing Set::Iterator

comparing iterators of two  
different sets is a bug

```
bool Set::Iterator::operator==(const Iterator& i) const {  
    assert(set == i.set);  
    return index == i.index;  
}
```

```
bool Set::Iterator::operator!=(const Iterator& i) const {  
    return !(*this == i);  
}
```

```
it != s.end()  
it == s.end()
```

# Implementing Set::Iterator

back in class **Set**, we need to implement the functions that actually return the iterators...

```
Set::Iterator Set::begin() const {
    return Iterator(this, 0);
}
```

```
Set::Iterator Set::end() const {
    return Iterator(this, size);
}
```

# Ranged-Based **for** Loops

- ▶ **Range-based for loops** simplify iterating over data structures

```
for (it = s.begin(); it != s.end(); ++it) {  
    int x = *it;  
    // use x ...  
}
```

```
for (int x : s) {  
    // use x (but not "it",  
    // there's no such variable)  
}
```

- ▶ With a range-based for loop, we can write a **clean** and **simple** loop over a Set's elements

```
int sum = 0;  
for (int value : set) {  
    sum += value;  
}
```

for every value *in* set  
apply sum += value

# Ranged-Based **for** Loops

- ▶ Range-based for loops work for **any class** with begin() and end() methods that return a **valid iterator**
  - I.e., a pointer or an object with ++, != and \* operators
- ▶ They also work with all **standard data structures** that come in the C++ library (STL) ← later!
- ▶ Note: to avoid copying large objects, declare the loop variable as a **reference**

```
for (const Student& s : students) {  
    cout << s.average() << endl;  
}
```

non-const reference only if we  
need to change the object

```
for (Student& s : students) {  
    s.grades[0].applyFactor(5);  
}
```

# A Set of Students

- ▶ How would we implement a set of **students**?
  - ◆ The code is very similar to a set of ints, except that the variables are of type Student
- ▶ Can we avoid **writing a Set for every type**?
  - ◆ **Yes!** Using **generic** classes
- ▶ We'll see that next lecture

*“The problem with using C++ is that there's a strong tendency in the language to require you to know everything before you can do anything.”*

- Larry Wall

# Templates

Generic Programming in C++

# The story so far...

- ▶ Code for a large project should be **split into modules** – a set of **functions** and **types** that constitute the project's building blocks
- ▶ Types should have an **interface** and an **implementation**
  - ◆ ADTs provide a **separation** between the type's interface and its implementation and simplify our work
- ▶ **Data structures** (such as our set of integers) are implemented as ADTs
  - ◆ But we might need a set of strings, a set of students and more sets... This causes **duplication**

# Generic Code

- ▶ To avoid code duplication, we will need to write code that can work on **multiple data types**
  - ◆ A **function** that can operate on multiple types (rather than a specific type) is called a **generic function**
  - ◆ An **ADT** that can operate on multiple types is called a **generic ADT**
- ▶ A generic function or ADT may not support *all* types, only those that fulfill certain **requirements**
- ▶ Each individual set we create will still only contain elements of a **single type**
  - ◆ The elements must be **comparable** so we can find elements, make sure the same element is not added twice, etc.

# C++ Templates

- ▶ C++ has a built-in mechanism for creating **code templates**
- ▶ The **compiler** generates code according to the template **automatically**
- ▶ The replacement of T by the type is done at **compilation time\***

```
template<class T>
T max(const T& a, const T& b) {
    return a > b ? a : b;
}
```

} template for creating a  
max function

```
int main() {
    int a = 7, b = 5;
    double d1 = 2.0, d2 = 3.0;
    cout << max(a, b) << endl;
    cout << max(d1, d2) << endl;
    return 0;
}
```

compiler defines function  
max<int> according to  
template

compiler defines max<double>

\* This makes templates very efficient in C++. In C# and Java, the replacement is done at runtime.

# Function Templates

- ▶ To define a function template:
  - ◆ Declare the **parameters** of the template
  - ◆ Write the function, using any **typename** parameter as if it is a type

```
template<class T>
T max_element(const T* arr, int size) {
    T result = arr[0];
    for (int i = 1; i < size; ++i) {
        result = max(result, arr[i]);
    }
    return result;
}
```

this template has one parameter

the **typename** keyword can be used instead of **class** (there's no difference in this context)

declare a variable of type T, and initialize it using its copy c'tor

call the function max() for two arguments of type T

# Function Templates

- ▶ Template functions can be used in two ways:
  - ◆ By explicitly stating the arguments of the template
  - ◆ Without stating the arguments, in which case the compiler will deduce them automatically

```
int n1 = 7, n2 = 5;  
double d1 = 2.0, d2 = 3.0;
```

```
cout << max<int>(n1, n2);  
cout << max<double>(d1, d2);
```

```
cout << max(n1, n2);  
cout << max(d1, d2);
```

template arguments stated  
explicitly

template arguments  
deduced by the compiler

# Argument Deduction

- ▶ The compiler can automatically choose the template arguments if they can be deduced from the function's parameters
  - ◆ If the compiler can't deduce them, they must be given explicitly

```
template<class T>
T* create() {
    return new T();
}
```

```
int* ptr = create(); // error
int* ptr2 = create<int>(); // ok
```

cannot deduce that  $T=$ int from  
the function parameters

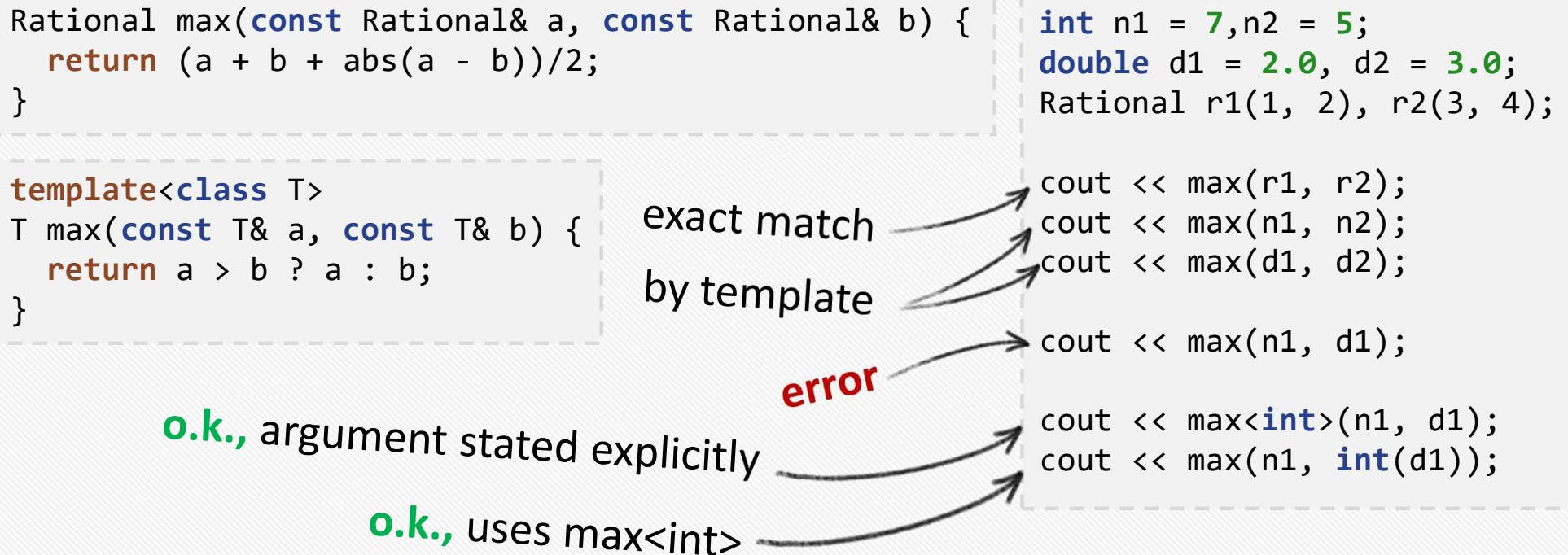
```
template<class T, class S>
T convert(const S& s) {
    return T(s);
}
```

```
int n = 7;
double d1 = convert(n); // error
double d2 = convert<double>(n);
double d3 = convert<double, int>(n);
```

$S=$ int can be deduced,  
 $T=$ double must be stated explicitly

# Argument Deduction

- ▶ This allows function templates to participate in the **overloading resolution process**:
  - ◆ The compiler will prefer **exact-match** functions to a template
  - ◆ The compiler will prefer **using a template** to using type conversions
  - ◆ The compiler will **never use type conversions when deducing types**



# Generic Classes

- ▶ The template mechanism can be used to define a **class template**
- ▶ This template will be used by the compiler to **create classes**

```
template<class T>
class Array {
    T* data;
    int size;
public:
    explicit Array(int size);
    Array(const Array& a);
    ~Array();
    Array& operator=(const Array& a);
    int size() const;
    T& operator[](int index);
    const T& operator[](int index) const;
};
```

```
template<class T>
T& Array<T>::operator[](int index) {
    assert(index >= 0 && index < size);
    return data[index];
}
```

# Generic Classes

- ▶ As with function templates, the compiler generates a **new class** for each type
- ▶ The template argument is a **part of the name of the new class**
  - ◆ There is **no automatic deduction** for template parameters in this case

Array<int> ≠ Array<double>

```
Array<int> arr1(10);
Array<double> arr2(10);
arr1 = arr2; // error, type mismatch

Array<Complex> arr3(10);
```

```
Array<double> half(const Array<int>& a) {
    Array<double> result(a.size());
    for (int i = 0; i < a.size(); i++) {
        result[i] = 0.5 * a[i];
    }
    return result;
}
```

# A Generic Set in C++

```
template <class T>
class Set {
public:
    Set();
    Set(const Set&);
    ~Set();
    Set& operator=(const Set&);

    bool add(const T& element);
    bool remove(const T& element);
    bool contains(const T& element) const;
    int getSize() const;

    Set& uniteWith(const Set&);
    Set& intersectWith(const Set&);

    class Iterator;
    Iterator begin() const;
    Iterator end() const;
}
```

set.h

```
private:
    T* data;
    int size;
    int maxSize;

    int find(const T& element) const;
    void expand();

    static const int EXPAND_RATE = 2;
    static const int INITIAL_SIZE = 10;
    static const int ELEMENT_NOT_FOUND = -1;
};

template<class T>
Set<T> unite(const Set<T>&, const Set<T>&);

template<class T>
Set<T> intersect(const Set<T>&, const Set<T>&);

template<class T>
std::ostream& operator<<(std::ostream& os,
                           const Set<T>& set);
```

set.h

Inside the scope of the class, we can omit the template arguments  
Set here means `Set<T>`, i.e., another Set of the same type

Outside the class, the `template<...>` declaration is needed

# A Generic Set in C++

```
template <class T>
class Set {
public:
    Set();
    Set(const Set&);
    ~Set();
    Set& operator=(const Set&);

    bool add(const T& element);
    bool remove(const T& element);
    bool contains(const T& element) const;
    int getSize() const;

    static Set union(const Set&, const Set&);
    static Set intersection(const Set&, const Set&);

    class Iterator;
    Iterator begin() const;
    Iterator end() const;
```

set.h

```
private:
    T* data;
    int size;
    int maxSize;

    int find(const T& element) const;
    void expand();

    static const int EXPAND_RATE = 2;
    static const int INITIAL_SIZE = 10;
    static const int ELEMENT_NOT_FOUND = -1;
};
```

```
template<class T>
std::ostream& operator<<(std::ostream& os,
                           const Set<T>& set);
```

Inside the scope of the class, we can omit the template arguments  
Set here means `Set<T>`, i.e., another Set of the same type

Outside the class, the `template<...>`  
declaration is needed

# A Generic Set in C++

each Set<T> class  
has **its own** inner  
Set<T>::Iterator  
class

```
template<class T>
class Set<T>::Iterator {

    const Set<T>*> set;
    int index;
    Iterator(const Set<T>* set, int index);
    friend class Set<T>;
```

public:

```
const T& operator*() const;
Iterator& operator++();
Iterator operator++(int);

bool operator==(const Iterator& iterator) const;
bool operator!=(const Iterator& iterator) const;

Iterator(const Iterator&) = default;
Iterator& operator=(const Iterator&) = default;
```

};

later in set.h

# Using the Generic Set

```
#include "set.h"
#include <iostream>
using std::cout;
using std::endl;

int main() {
    Set<int> set1;
    Set<int> set2;

    for (int j = 0; j < 20; j += 2) {
        set1.add(j);
    }

    for (int num : set1) {
        set2.add(2*num);
    }

    cout << set1 << endl;
    cout << set2 << endl;
    cout << unite(set1, set2) << endl;
    cout << intersect(set1, set2) << endl;
    return 0;
}
```

main.cpp

using the generic Set is identical to the integer set

we can use a **range-for loop** here because Set<int> has begin() and end() methods that return a **valid iterator**

# Using the Generic Set

```
#include "set.h"
#include <iostream>
using std::cout;
using std::endl;

int main() {
    Set<int> set1;
    Set<int> set2;

    for (int j = 0; j < 20; j += 2) {
        set1.add(j);
    }

    for (int num : set1) {
        set2.add(2*num);
    }

    cout << set1 << endl;
    cout << set2 << endl;
    cout << Set<int>::union(set1, set2) << endl;
    cout << Set<int>::intersection(set1, set2) << endl;
    return 0;
}
```

main.cpp

using the generic Set is identical to the integer set

we can use a **range-for loop** here because Set<int> has begin() and end() methods that return a **valid iterator**

# Generic Set Implementation

template code must appear in the header file -  
the compiler has to know it

therefore, all of Set's code will be in  
set.h, and there is **no set.cpp**

**"the big three":**

- copy c'tor
- operator=()
- ✓ **d'tor**

```
template<class T>
Set<T>::Set() :
    data(new T[INITIAL_SIZE]),
    size(0),
    maxSize(INITIAL_SIZE)
{}
```

```
template<class T>
Set<T>::~Set() {
    delete[] data;
}
```

# Generic Set Implementation

- "the big three":
- ✓ copy c'tor
  - operator=()
  - ✓ d'tor

```
template<class T>
Set<T>::Set(const Set& set) :
    data(new T[max(set.getSize(), INITIAL_SIZE)]),
    size(set.getSize()),
    maxSize(max(set.getSize(), INITIAL_SIZE))
{
    for(int i = 0; i < size; i++) {
        data[i] = set.data[i];
    }
}
```

set.h



must use **assignment** and not  
just **memcpy** the array –  
the type T might define an  
operator=() !!

# Generic Set Implementation

"the big three":

- ✓ copy c'tor
- ✓ **operator=()**
- ✓ d'tor

```
template<class T>
Set<T>& Set<T>::operator=(const Set& set) {
    if (this == &set) {
        return *this;
    }

    delete[] data;
    data = new T[max(INITIAL_SIZE, set.getSize())];
    size = set.getSize();
    maxSize = max(INITIAL_SIZE, set.getSize());

    for (int i = 0; i < size; ++i) {
        data[i] = set.data[i];
    }
    return *this;
}
```

set.h

# Generic Set Implementation

uses  
**operator==()** of  
type T

```
template<class T>
int Set<T>::find(const T& elem) const {
    for(int i = 0; i < size; i++) {
        if (data[i] == elem) {
            return i;
        }
    }
    return ELEMENT_NOT_FOUND;
}
```

```
template<class T>
bool Set<T>::contains(const T& elem) const {
    return find(elem) != ELEMENT_NOT_FOUND;
}
```

```
template<class T>
int Set<T>::getSize() const {
    return size;
}
```

# Generic Set Implementation

```
template<class T>
void Set<T>::expand() {
    int newSize = maxSize * EXPAND_RATE;
    T* newData = new T[newSize];
    for (int i = 0; i < size; ++i) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
    maxSize = newSize;
}
```

# Generic Set Implementation

```
template<class T>
bool Set<T>::add(const T& elem) {
    if (contains(elem)) {
        return false;
    }

    if (size >= maxSize) {
        expand();
    }

    data[size++] = elem;
    return true;
}
```

```
template<class T>
bool Set<T>::remove(const T& elem) {
    int index = find(elem);
    if (index == ELEMENT_NOT_FOUND) {
        return false;
    }

    data[index] = data[--size];
    return true;
}
```

# Generic Set Implementation

```
template<class T>
Set<T>& Set<T>::uniteWith(const Set& other) {
    for (int i = 0; i < other.getSize(); ++i) {
        this->add(other.data[i]);
    }
    return *this;
}
```

```
template<class T>
Set<T>& Set<T>::intersectWith(const Set& other) {
    for (int i = 0; i < this->getSize(); ++i) {
        if (!other.contains(data[i])) {
            this->remove(data[i]);
        }
    }
    return *this;
}
```

# Generic Set Implementation

```
template<class T>
Set<T> Set<T>::Set union(const Set& set1, const Set& set2) {
    Set unionSet;
    for (const T& x : set1) {
        unionSet.add(x);
    }
    for (const T& x : set2) {
        unionSet.add(x);
    }
    return unionSet;
}
```

```
template<class T>
Set<T> Set<T>::Set intersection(const Set& set1, const Set& set2) {
    Set intersectionSet;
    for (const T& x : set1) {
        if (set2.contains(x)) {
            intersectionSet.add(x);
        }
    }
    return intersectionSet;
}
```

# Generic Set Implementation

when using a type that is defined within a template class, we must qualify it by the **typename** keyword



this is because without knowing T, the compiler cannot know for sure if **A<T>::B** is a type or something else (e.g., a static variable or function)\*

```
template<class T>
std::ostream& operator<<(std::ostream& os,
                           const Set<T>& set) {
    os << "{";
    bool first = true;
    for (typename Set<T>::Iterator it = set.begin();
         it != set.end(); ++it) {
        if (!first) {
            os << ",";
        }
        first = false;
        os << " " << (*it);
    }
    os << "}";
    return os;
}
```

Call operator<< of the object

\*The reason is outside the scope of this course, if you're curious, search for "template specialization"

# Generic Set Implementation

```
template<class T>
std::ostream& operator<<(std::ostream& os, const Set<T>& set)
{
    os << "{";
    bool first = true;
    for (const T& elem: set) {
        if (!first) {
            os << ",";
        }
        first = false;
        os << " " << elem;
    }
    os << " }";
    return os;
}
```



This is possible because a set has begin() and end() with all the requirements

# Generic Set Implementation

```
template<class T>
typename Set<T>::Iterator Set<T>::begin() const {
    return Iterator(this, 0);
}
```

```
template<class T>
typename Set<T>::Iterator Set<T>::end() const {
    return Iterator(this, size);
}
```

# Implementing the Iterator

```
template<class T>
Set<T>::Iterator::Iterator(const Set* set, int index) :
    set(set), index(index)
{}
```

```
template<class T>
const T& Set<T>::Iterator::operator*() const {
    assert(index >= 0 && index < set->getSize());
    return set->data[index];
}
```

# Implementing the Iterator

```
template<class T>
typename Set<T>::Iterator& Set<T>::Iterator::operator++() {
    ++index;
    return *this;
}
```

```
template<class T>
typename Set<T>::Iterator Set<T>::Iterator::operator++(int) {
    Iterator result = *this;
    ++*this;
    return result;
}
```

# Implementing the Iterator

```
template<class T>
bool Set<T>::Iterator::operator==(const Iterator& i) const {
    assert(set == i.set);
    return index == i.index;
}
```

```
template<class T>
bool Set<T>::Iterator::operator!=(const Iterator& i) const {
    return !(*this == i);
}
```

# What About Operator->() ?

- ▶ There is still one pointer behavior which we **did not cover** with our current Set::Iterator implementation

```
struct Point2D {  
    double x, y;  
};
```

ordinary pointers provide  
an **operator->()** to allow  
access to struct/class  
members

```
Point2D centerOfMass(const Set<Point2D>& set) {  
  
    Point2D center = {0,0};  
  
    for (Set<Point2D>::Iterator it = set.begin() ;  
         it != set.end() ; ++it)  
    {  
        center.x += it->x;  
        center.y += it->y;  
    }  
  
    center.x /= set.getSize();  
    center.y /= set.getSize();  
  
    return center;  
}
```

# Adding Operator->()

notice that operator->()  
does not take any  
parameters

```
template<class T>
class Set<T>::Iterator {
    const Set* set;
    int index;
    Iterator(const Set* set, int index);
    friend class Set;

public:
    const T& operator*() const;
    const T* operator->() const;
    Iterator& operator++();
    Iterator operator++(int);

    bool operator==(const Iterator& iterator) const;
    bool operator!=(const Iterator& iterator) const;

    Iterator(const Iterator&) = default;
    Iterator& operator=(const Iterator&) = default;
};
```

# Implementing Operator->()

- ▶ Despite how it is used in code, operator->() **does not take any arguments**
- ▶ Instead, operator->() has a unique behavior: the compiler **automatically applies operator ->** to its return value
  - ◆ So, operator->() must return a type on which **-> can be applied again**

```
Set<Point2D>::Iterator it = s.begin();
double x = it->x;
```

compilation

```
Set<Point2D>::Iterator it = s.begin();
double x = it.operator->()->x;
```

we see that it.operator->() should  
return a pointer to the current  
element

# Implementing Operator->()

```
template<class T>
const T* Set<T>::Iterator::operator->() const {
    assert(index >= 0 && index < set->getSize());
    return &(set->data[index]);
}
```

the only change from  
operator\*()

```
Set<int>::Iterator it = s.begin();
double x = it->x; // doesn't compile
```

if the template argument is not a  
struct/class with a public member  
named x, this line will not compile

# Template Requirements

- ▶ The compiler can only do very simple syntax checking when it goes over the template code
- ▶ The actual compilation is done only when the template is **instantiated with a specific type**
  - ◆ This is when the compiler checks that the template code can work with the specific type that is used

```
template<class T>
T max(const T& a, const T& b) {
    return a > b ? a : b;
}
```

```
int main() {
    int n1 = 7, n2 = 5;
    Complex c1(2.0, 1.0), c2(0.0, 1.0);
    cout << max(n1, n2) << endl;
    cout << max(c1, c2) << endl;
    return 0;
}
```

this will compile  
compilation error: cannot compile  
max<Complex> since there is no  
operator<>() for Complex numbers

# Template Requirements

- ▶ In order for an instantiation of a template to compile, its arguments must satisfy the **requirements** imposed by the template code
  - ◆ If the code does not compile, the compiler will generate a multi-line error explaining the context in which the compilation has failed (these are usually very hard to read)
- ▶ What are the requirements from **T** in our generic set?
  - ◆ To find out, we need to look at how T is used throughout Set's code

# Template Requirements

- ▶ What are the requirements from T in our generic set?

T must have:  
- default c'tor T::T()  
- operator=()

```
template<class T>
Set<T>::Set(const Set& set) :
    data(new T[max(INITIAL_SIZE, set.getSize())]),
    size(set.getSize()),
    maxSize(max(INITIAL_SIZE, set.getSize()))

{
    for(int i = 0; i < size; i++) {
        data[i] = set.data[i];
    }
}
```

# Template Requirements

- ▶ What are the requirements from T in our generic set?

T must have:

- default c'tor T::T()
- operator=()
- operator==( )

```
template<typename T>
int Set<T>::find(const T& elem) const {
    for(int i = 0; i < size; i++) {
        if (data[i] == elem) {
            return i;
        }
    }
    return ELEMENT_NOT_FOUND;
}
```

# Template Requirements

- ▶ What are the requirements from T in our generic set?

T must have:

- default c'tor T::T()
- operator=()
- operator==()
- d'tor - T::~T()



```
template<typename T>
Set<T>::~Set<T>() {
    delete[] data;
}
```

# Conditional Template Compilation

- ▶ The compiler will only compile the function templates that are **actually used** by the user code
- ▶ A template parameter **T** must only satisfy the requirements of the functions that are compiled
- ▶ Thus, **T might not need to satisfy all the template class requirements**, if some of its functions are not used

# Conditional Template Compilation

- ▶ Example: what if we want to use Set with a type that does not have an `operator<<()` ?

to compile, this line requires T  
to have an `operator<<()`

if T does not have this operator, we  
can still create and use a `Set<T>`,  
just not use `operator<<()` on this set

```
template<class T>
std::ostream& operator<<(std::ostream& os,
                           const Set<T>& set) {
    os << "{";
    bool first = true;
    for (const T& elem : set)
    {
        if (!first) { os << ","; }
        first = false;
        os << " " << elem;
    }
    os << "}";
    return os;
}
```

# Function Objects

- An object of a class which overloads **operator()** is called a **function object** or a **functor**
  - Because it can be invoked like a function

```
class DividesBy {  
    int n;  
public:  
    DividesBy(int n) : n(n) {}  
    bool operator()(int number) const {  
        return number % n == 0;  
    }  
};
```

operator() can be overloaded for any number of parameters (in this case, just one)

```
int main() {  
    DividesBy isEven(2);  
  
    cout << isEven(9) << endl;  
    cout << isEven(8) << endl;  
  
    DividesBy isTriple(3);  
  
    cout << isTriple(9) << endl;  
    cout << isTriple(8) << endl;  
  
    return 0;  
}
```

notice the difference between calling a c'tor and operator()

# Function Objects

- ▶ Let's add a **filter function** to our set
  - ◆ The function will return a **new set** that contains only the elements that **satisfy a certain condition**
- ▶ By defining **filter()** as a template, it can accept **any function object** as a condition for filtering

```
template <class T>
class Set {
public:
    // ...

    template<class Condition>
    Set filter(Condition c) const;
    // ...
};
```

```
template<class T>
template<class Condition>
Set<T> Set::filter(Condition c) const {
    Set result;
    for (const T& elem : *this) {
        if (c(elem)) {
            result.add(elem);
        }
    }
    return result;
}
```

Yes, we defined a template inside a template

For this template to compile, **Condition** has to be a type that can be invoked with operator(), so either an ordinary **pointer to function**, or any **function object**

# Function Objects

- ▶ What is the advantage of **using function objects** compared to ordinary functions?
  - ◆ With ordinary functions, we may need to **create a new function** for every parameter value
    - For example, DividesBy2(x), DividesBy3(x), ...
  - ◆ With function objects, we can simply **pass the parameters to the object's constructor**

# Function Objects

```
class Between {  
    int min, max;  
public:  
    Between(int min, int max) :  
        min(min), max(max) {}  
  
    bool operator()(int val) const {  
        return min <= val && val <= max;  
    }  
};
```

```
void test(const Set<int>& s) {  
    DividesBy isEven(2);  
    cout << s.filter(isEven) << endl;  
  
    int x1, x2;  
    cin >> x1 >> x2;  
    cout << s.filter(Between(x1,x2)) << endl;  
}
```

A functor that checks whether a value is between its two parameters: min and max

```
int main() {  
    Between between_2_9(2, 9);  
  
    cout << between_2_9(5) << endl;  
  
    return 0;  
}
```

# Lambda expressions

- ▶ It's awkward to define a class for each function

```
void test(const Set<int>& s) {  
    DividesBy isEven(2);  
    cout << s.filter(isEven) << endl;  
    int x1, x2;  
    cin >> x1 >> x2;  
    cout << s.filter(Between(x1,x2)) << endl;  
}
```

- ▶ Lambda expressions are ad hoc functions

```
void test_with_lambdas(const Set<int>& s) {  
    cout << s.filter([](int i) { return i % 2 == 0; }) << endl;  
  
    int x1, x2;  
    cin >> x1 >> x2;  
    cout << s.filter([x1,x2](int i) { return x1 <= i && i <= x2; }) << endl;  
}
```

- ▶ Beyond the scope of this course...

# Templates - Ending Notes

- ▶ Templates allow to create **generic code** in C++
- ▶ Using templates we can decompose software into generic algorithms, data structures, and our specific problem domain
- ▶ The template mechanism is **handled at compile time** and is **type-safe**
  - ◆ If we try to add a **Student** to a **Set<int>**, we will get a **compilation error**

**Days 1 - 10**

Teach yourself variables, constants, arrays, strings, expressions, statements, functions,...



**Days 11 - 21**

Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism, ....



**Days 22 - 697**

Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.



**Days 698 - 3648**

Interact with other programmers. Work on programming projects together. Learn from them.



**Days 3649 - 7781**

Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.



**Days 7782 - 14611**

Teach yourself biochemistry, molecular biology, genetics,...



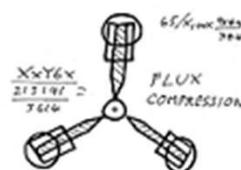
**Day 14611**

Use knowledge of biology to make an age-reversing potion.



**Day 14611**

Use knowledge of physics to build flux capacitor and go back in time to day 21.



**Day 21**

Replace younger self.



As far as I know, this is the easiest way to

"Teach Yourself C++ in 21 Days".

<http://abstrusegoose.com/249>

# Inheritance

Inheritance of Classes,  
Polymorphism and UML

# Designing a Large Project

- ▶ Example: We wish to develop a **card game**
- ▶ **Some** of the requirements are known in advance:
  - ◆ The game should support **multiple players**
  - ◆ Every player has **money** and **health** stats
  - ◆ On their turn, a player will draw one card and play it, if they can pay its **cost**
  - ◆ There can be **many types** of cards with different effects
    - ◆ E.g., a card may **increase health** if purchased



# Designing a Large Project

- ▶ How do we start? Jump right to coding? **No!**
- ▶ How do we **design** our project?
- ▶ How do we **describe** and **discuss** it with other team members?
- ▶ How do we **document** our design?
- ▶ How do we keep **updating** our project as more requirements arrive?

**A picture is worth a 1000 words, so ...**

# Unified Modeling Language

- ▶ UML is a modeling language that provides a **standard way** to visualize the design of a system
  - ◆ It's easier to **design, develop, explain, discuss** and **Maintain** a large project using **images** rather than code
- ▶ UML contains a huge number of diagrams and roles
- ▶ We will focus on a small (but important) portion:
  - ◆ **Class** diagrams
  - ◆ **Sequence** diagrams

# Example: Card Game

```
class CardGame {  
    Player* currentPlayer; ←  
    Array<Player> players;  
    Array<Card> cards;  
  
public:  
    void playTurn();  
};
```

pointer to a player  
in the **players** array

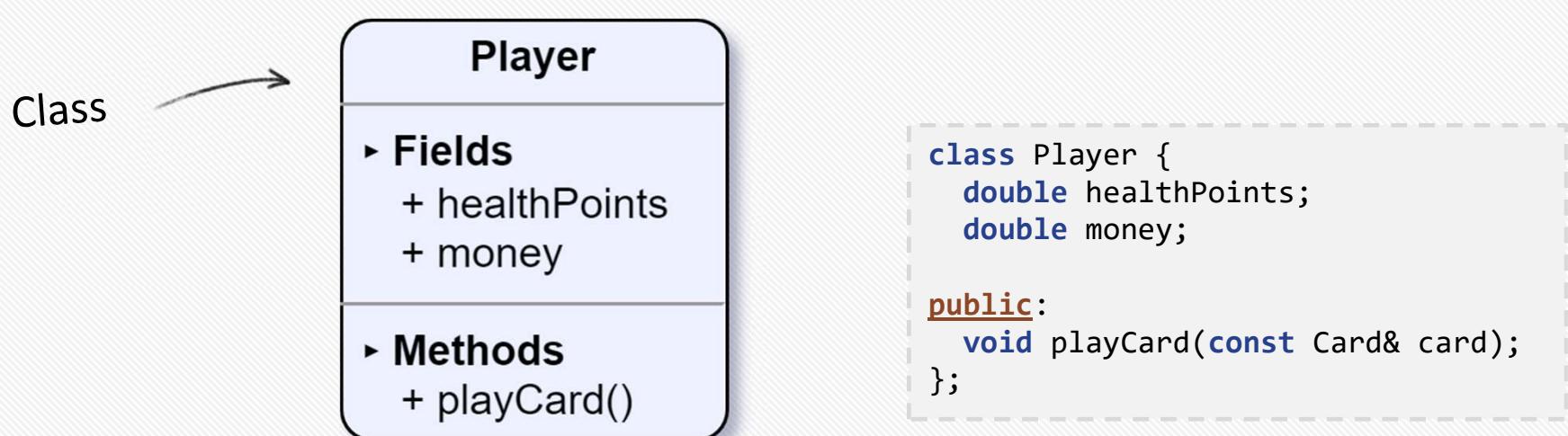
```
class Card {  
    double price;  
    double healthGain;  
};
```

This is just a **very basic** implementation,  
there is a lot missing like getters/setters,  
constructors/destructors and more ..

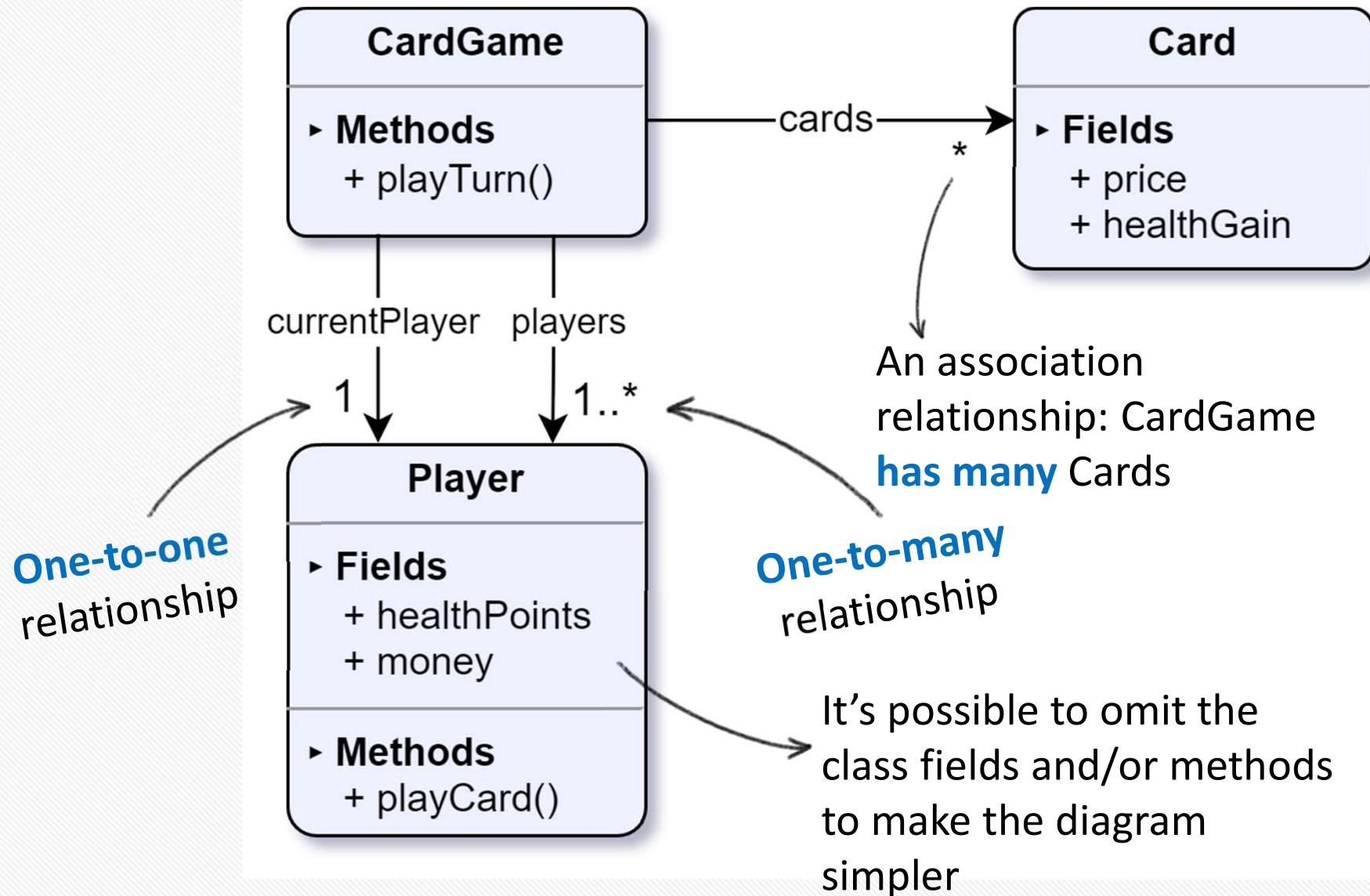
```
class Player {  
    double healthPoints;  
    double money;  
  
public:  
    void playCard(const Card& card);  
};
```

# Class Diagram

- ▶ Describes the **static structure** of a system
- ▶ Shows the **classes**, their attributes and methods, and the relationships between them



# Class Diagram

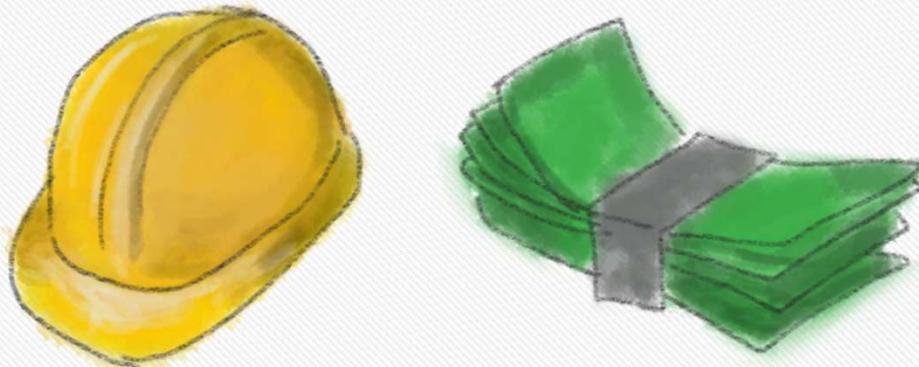


# Unified Modeling Language

- ▶ We will see many more class diagrams throughout this lecture
  - ◆ They are most useful with **inheritance**
- ▶ We will leave **sequence diagrams** for a later lecture

# A Human Resources Problem

- ▶ Assume we wish to create software to handle human resources in our company
- ▶ We identify that we have **two types** of employees in our company:
  - ◆ Engineers
  - ◆ Sales people
- ▶ What classes should we create?



# One-Class Solution

- ▶ Creating one class for all types of employees is bad for maintenance and will cause **incoherent code**:

```
class Employee {  
public:  
    // ...  
    string getName() const;  
    void giveRaise(int amount);  
    enum Type { ENGINEER, SALESPERSON };  
  
private:  
    string name;  
    Date birth;  
    int salary;  
    Type type;  
    set<string> degrees; // for an engineer  
    double comissionRate; // for a sales person  
};
```

signs of a class trying to  
fill too many roles

# Two-Class Solution

- ▶ Creating one class for engineers, and another for sales people, will cause **code duplication**:

```
class Engineer {  
public:  
    // ...  
    string getName() const;  
    void giveRaise(int amount);  
  
private:  
    string name;  
    Date birth;  
    int salary;  
    set<string> degrees;  
};
```

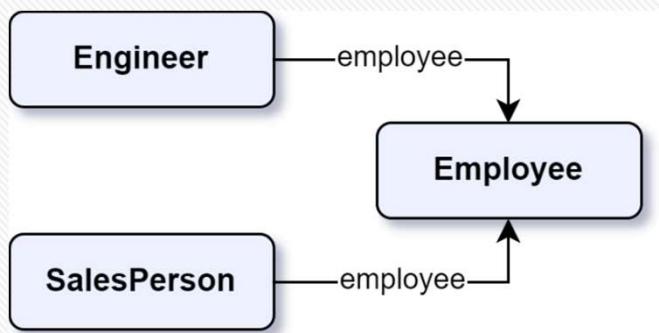
```
void printName(const Engineer& e);  
void printName(const SalesPerson& s);
```

```
class SalesPerson {  
public:  
    // ...  
    string getName() const;  
    void giveRaise(int amount);  
  
private:  
    string name;  
    Date birth;  
    int salary;  
    double comissionRate;  
};
```

need to **duplicate functions**  
to handle both classes

# Identifying the Common Part

- ▶ Engineers and SalesPerson have **common functionality**
  - ◆ We can extract this part into a third class:



```
class Employee {  
public:  
    // ...  
    string getName() const;  
    void giveRaise(int amount);  
private:  
    string name;  
    Date birth;  
    int salary;  
};
```

Common to all employees

just calls the  
function  
employee.  
getName()

```
class Engineer {  
public:  
    // ...  
    string getName() const;  
    void giveRaise(int amount);  
private:  
    Employee employee;  
    set<string> degrees;  
};
```

```
class SalesPerson {  
public:  
    // ...  
    string getName() const;  
    void giveRaise(int amount);  
private:  
    Employee employee;  
    double comissionRate;  
};
```

# Using the Common Part

- ▶ This solution is better, but still has problems:
  - ◆ We need to write **extra code** to expose the functionality of the common class in the specialized classes:

```
string Engineer::getName() const {  
    return employee.getName();  
}
```

- ◆ What if we want code that can work on **any** employee?

```
void printEmployeeDetails(const Employee& emp) {  
  
    cout << "Employee details: " << endl;  
    cout << "Name: " << emp.getName() << endl;  
    cout << "Salary: " << emp.getSalary() << endl;  
  
}
```



this function will **not work** on an  
Engineer or SalesPerson object,  
even though both have getName()  
and getSalary() functions

# Inheritance

- ▶ In object-oriented languages, we solve this using **inheritance**
- ▶ Inheritance allows us to take an existing class, and **derive new classes** from it
  - ◆ So Engineer and SalesPerson can **inherit the functionality** of an Employee class

```
class Employee {  
public:  
    // ...  
    string getName() const;  
    void giveRaise(int amount);  
private:  
    string name;  
    Date birth;  
    int salary;  
};
```

```
class Engineer : public Employee {  
public:  
    // engineer-only functions ...  
  
private:  
    set<string> degrees;  
};
```

# Inheritance

- ▶ The inheriting class has all the **fields and methods** of the base class, plus the **new ones** it defines

```
class Employee {  
public:  
    // ...  
    string getName() const;  
    void giveRaise(int amount);  
private:  
    string name;  
    Date birth;  
    int salary;  
};
```

```
void f(Engineer& engineer) {  
    engineer.addDegree("B.Sc. Computer Science");  
    engineer.giveRaise(10);  
}
```

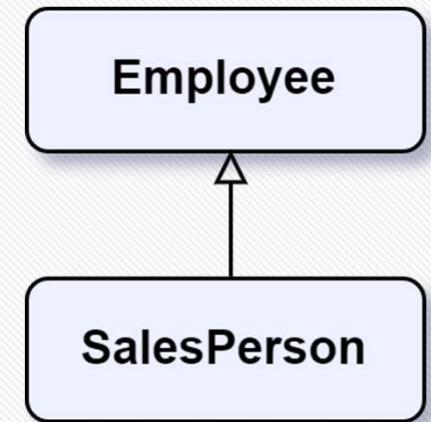
defined in Engineer

defined in Employee

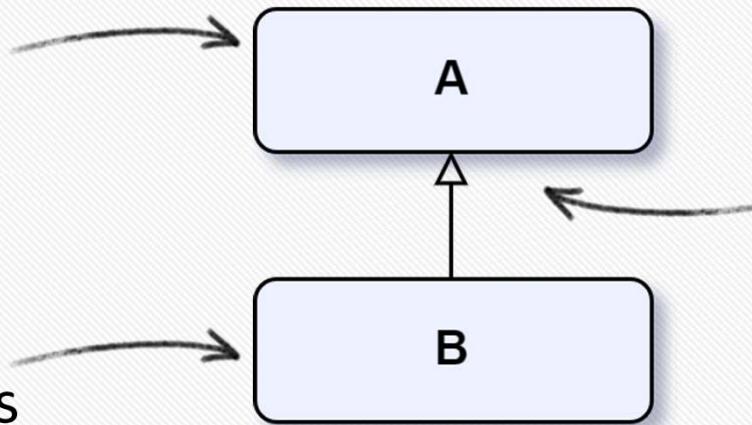
```
class Engineer : public Employee {  
public:  
    void addDegree(string degree);  
private:  
    set<string> degrees;  
};
```

# Inheritance - Terminology

- ▶ There are several equivalent terms for describing inheritance relations:
  - ◆ Class B **inherits** class A
  - ◆ Class B **derives from** class A
  - ◆ A is the **base class** and B is the **derived class**
  - ◆ A is the **superclass** and B is the **subclass**



superclass or  
base class



subclass or  
derived class

for describing inheritance  
in UML, we use an **empty  
arrowhead**

# Access Control

- ▶ The derived class **cannot access the private section** of the base class (even though it contains it!)

```
class Employee {  
public:  
    string getName() const;  
    void giveRaise(int amount);  
private:  
    string name;  
    Date birth;  
  
    int salary;  
};
```

```
class Engineer : public Employee {  
public:  
    void addReview(string text, int score);  
private:  
    list<string> reviews;  
};
```

```
void Engineer::addReview(string text, int score) {  
    reviews.push_front(text);  
    if (score == 5) {  
        salary += 1000;  
    }  
}
```

**error:** salary  
is private!

```
void f(Engineer& engineer) {  
    engineer.salary = 1000;  
}
```

Note: we can still use these  
private members through the  
**public interface** of the base  
class

# Access Control

- ▶ We can declare class members as **protected**
  - ◆ Protected members can be accessed from within **any class deriving from the base class**
  - ◆ Protected members **cannot be accessed** from outside the class

```
class Employee {  
public:  
    string getName() const;  
    void giveRaise(int amount);  
private:  
    string name;  
    Date birth;  
protected:  
    int salary;  
};
```

```
class Engineer : public Employee {  
public:  
    void addReview(string text, int score);  
private:  
    list<string> reviews;  
};
```

```
void Engineer::addReview(string text, int score) {  
    reviews.push_front(text);  
    if (score == 5) {  
        salary += 1000;  
    }  
}
```

*o.k., salary is protected*

```
void f(Engineer& engineer) {  
    engineer.salary = 1000;  
}
```

*error: salary is protected*

# Subtyping

- ▶ A derived class such as Engineer is a **subtype** of the base class Employee
  - ◆ This means that an Engineer can be used anywhere that an Employee is acceptable
- ▶ This is possible because **all the public members** of the base class are available in the derived class as well

```
void giveRaiseAndLog(Employee& e) {  
    e.giveRaise(1000);  
    cout << "gave raise to ";  
    cout << e.getName() << endl;  
}
```

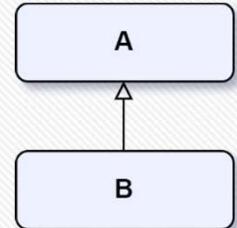
```
Engineer engineer("Jane Doe");  
SalesPerson salesPerson("John Doe");  
giveRaiseAndLog(engineer);  
giveRaiseAndLog(salesPerson);
```

An engineer *is an*  
employee

# Subtyping

- In C++, if B is a subtype of A, this means that **B\* can be used as A\***, and **B& can be used as A&**

- ◆ But not vice versa!
- ◆ Note that this requires use of either **pointers** or **references**



```
Engineer engineer("Jane Doe");
SalesPerson salesPerson("John Doe");
Employee employee("John Smith");

Employee& ref = engineer; // o.k.
Employee* ptr = &salesPerson; // o.k.

Engineer* ptr2 = &employee; // error

Employee copy = engineer; // compiles,
                           // but probably a bug

Employee* array[] = {
    &engineer, &salesPerson, &employee
};
```

Engineer is an Employee

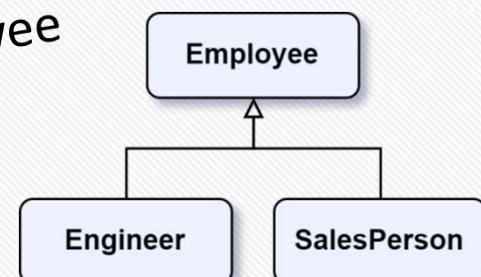
SalesPerson is an Employee

An Employee is not necessarily an Engineer

Copies **only the "Employee part"** of

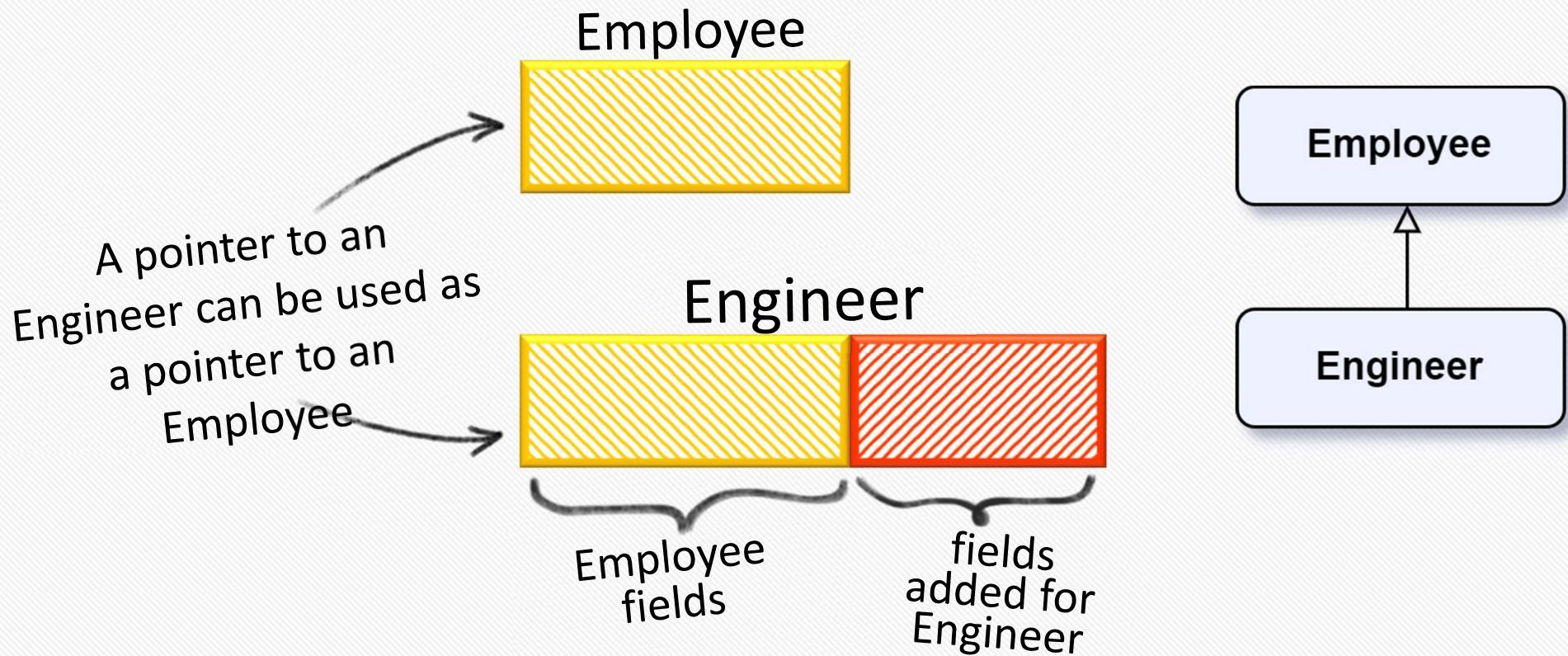
Engineer  
Subtyping allows **grouping**  
all types of employees

(in an **array** or any **other container** of pointers)



# Subtyping

- ▶ The common implementation of subtyping is to add the data of the derived class **at the end** of a base class object



# Example: Manager

## ▶ Let's add a **Manager** class

- ◆ Using subtyping, existing code for Employees will work with this new class!
- ◆ **No extra code** is needed for this

```
class Manager : public Employee {  
public:  
    Manager(string name);  
    bool isManagerOf(const Employee* emp) const;  
    void addEmployee(Employee* emp);  
private:  
    set<Employee*> employees;  
};
```

↑  
some of the employees can  
be Managers themselves

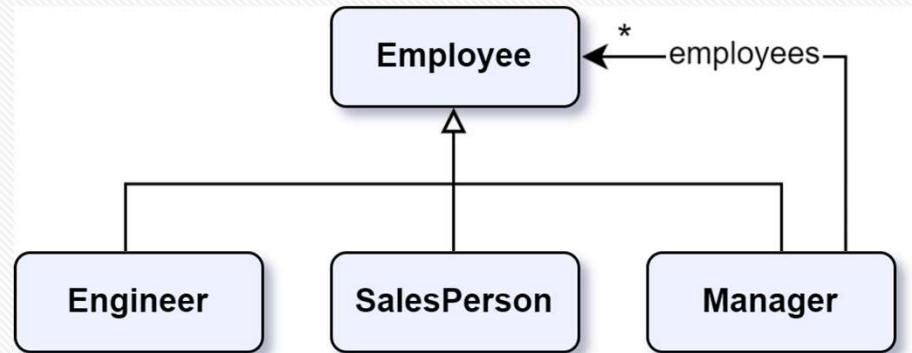
```
Manager ceo("Big Boss");  
  
Manager cfo("Money Bags");  
ceo.addEmployee(&cfo);  
  
Manager cto("Smart T. Pants");  
ceo.addEmployee(&cto);  
  
Engineer engineer("Sir Worksalot");  
cto.addEmployee(&engineer);  
  
giveRaiseAndLog(cto);
```

this function will work with a Manager  
object, even though the class didn't  
even exist when the function was  
written!

# Example: Manager

```
class Employee {  
public:  
    Employee(string name);  
private:  
    string name;  
    int salary;  
};
```

```
class Manager : public Employee {  
public:  
    Manager(string name);  
private:  
    set<Employee*> employees;  
};
```



design pattern known as a **Composite**, we'll talk more about design patterns in the next lectures

# Constructors

- When constructing an object of a derived class, its **base class part is constructed first**. The order of operations is:
  - The **base class** members are constructed
  - The **base class** constructor code is called
  - The **derived class** members are constructed
  - The **derived class** constructor code is called
- We can pass arguments to the base class constructor in the **initialization list**
  - Otherwise, the base class default constructor is used

```
class Employee {  
public:  
    Employee(string name);  
private:  
    string name;  
    int salary;  
};
```

```
class Manager : public Employee {  
public:  
    Manager(string name);  
private:  
    set<Employee*> employees;  
};
```

Employee part is  
constructed first

```
Employee::Employee(string name) :  
    name(name), salary(STARTING_SALARY)  
{}
```

```
Manager::Manager(string name) :  
    Employee(name), employees()  
{}
```

what would happen without this  
explicit initialization?

a compilation error, since the compiler would try  
to call Employee::Employee(), which does not  
exist

# Constructors

- ▶ The base class constructors are **not inherited** by the derived class
  - ◆ We must explicitly define any constructors the derived class needs

```
class Employee {  
public:  
    Employee(string name);  
private:  
    string name;  
    int salary;  
};
```

```
class Manager : public Employee {  
public:  
    Manager(string name) :  
        Employee(name) {}  
private:  
    set<Employee*> employees;  
};
```

the constructor from  
Employee that takes a **string**  
**name** is not inherited – we  
must define it explicitly

we can omit employees from  
the initialization list because  
we just want its default  
constructor

# Constructors

- ▶ Like any other class, a derived class automatically has a **copy constructor** and an **assignment operator**
  - ◆ The compiler-generated **copy constructor** will first call the base class copy constructor, and then the copy constructors of all the derived class members
  - ◆ The compiler-generated **assignment operator** will first call the base class assignment operator, and then the assignment operators of all the derived class members
- ▶ A **default constructor** will be generated only if no other constructor is defined for the derived class
  - ◆ The compiler-generated **default constructor** calls the base class default constructor, and then the default constructors of all the derived class members

# Destructors

- ▶ Destructing a derived class is done **in reverse order to construction**
  1. The **derived class** destructor code is called
  2. The **derived class** members are destructed
  3. The **base class** destructor code is called
  4. The **base class** members are destructed
- ▶ There is **no need to explicitly call** the base class destructor from the derived class destructor – it is called automatically

```
class Manager : public Employee {  
public:  
    ~Manager();  
private:  
    set<Employee*> employees;  
};
```

```
class Engineer : public Employee {  
public:  
    void addReview(string text, int score);  
private:  
    list<string> reviews;  
};
```

```
Manager::~Manager() {  
    // ...  
}
```

Employee part is  
destructed here

the compiler-generated  
Engineer::~Engineer() simply  
calls Employee::~Employee()

# Constructors and Destructors: Example

- ▶ What will the following code print?

```
class Engine {  
public:  
    Engine() { cout << "Engine ctor" << endl; }  
    ~Engine() { cout << "Engine dtor" << endl; }  
};  
  
class Battery {  
public:  
    Battery() { cout << "Battery ctor" << endl; }  
    ~Battery() { cout << "Battery dtor" << endl; }  
};  
  
class Car {  
    Engine engine;  
public:  
    Car() { cout << "Car ctor" << endl; }  
    ~Car() { cout << "Car dtor" << endl; }  
};  
  
class ElectricCar : public Car {  
    Battery battery;  
public:  
    ElectricCar() { cout << "ElectricCar ctor" << endl; }  
    ~ElectricCar() { cout << "ElectricCar dtor" << endl; }  
};
```

```
int main() {  
    ElectricCar car;  
    cout << "===== " << endl;  
    return 0;  
};
```

```
Engine ctor  
Car ctor  
Battery ctor  
ElectricCar ctor  
=====  
ElectricCar dtor  
Battery dtor  
Car dtor  
Engine dtor
```

# Overriding Methods

- ▶ What if different employees have different computations for giveRaise()?
- ▶ We can achieve this by **overriding** the base class function
  - ◆ Member variables cannot be overridden

```
class Employee {  
public:  
    // ...  
    void giveRaise(int amount);  
};
```

```
class SalesPerson : public Employee {  
public:  
    // ...  
    void giveRaise(int amount);  
};
```

the overridden function is declared again in the derived class

```
void Employee::giveRaise(int amount) {  
    salary += amount;  
}
```

```
void SalesPerson::giveRaise(int amount) {  
    Employee::giveRaise(amount + comissionRate*sales);  
}
```

the overridden method of the base class can be called using the base class name  
(it is common for the overriding function to call the overridden one)

# Overriding Methods

- ▶ Using overrides, each class in the hierarchy can have a **different behavior** for the **same message**:

```
class Employee {  
public:  
    // ...  
    void giveRaise(int amount);  
};
```

```
class SalesPerson : public Employee {  
public:  
    // ...  
    void giveRaise(int amount);  
};
```

```
class Manager : public Employee {  
public:  
    // ...  
    void giveRaise(int amount);  
};
```

```
void SalesPerson::giveRaise(int amount) {  
    Employee::giveRaise(amount + comissionRate * sales);  
}
```

```
void timeForBonus(SalesPerson& salesperson,  
                  Manager& manager,  
                  Engineer& engineer) {  
  
    salesPerson.giveRaise(1000);  
    manager.giveRaise(1000);  
    engineer.giveRaise(1000);  
}
```

SalesPerson::giveRaise(int)  
Manager::giveRaise(int)  
Employee::giveRaise(int)  
(not overridden)

# Overriding Methods: Caveat

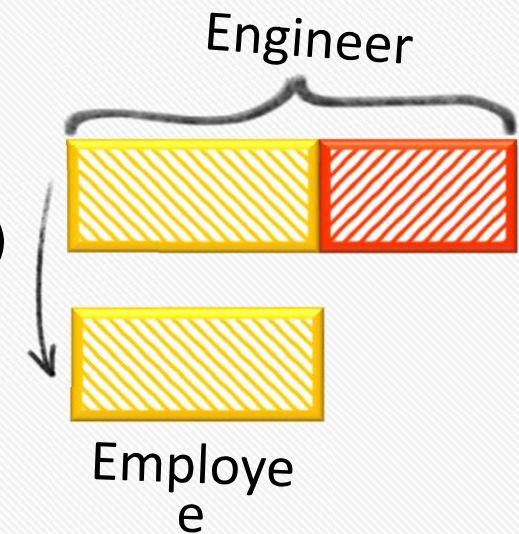
- ▶ We would like a container to store **all employees**
  - ◆ However, we cannot store them by value:

```
Engineer engineer("Jane Doe");
SalesPerson salesPerson("John Doe");
Manager manager("Mr. Bossman");

Employee employees[] = {
    engineer, salesPerson, manager
};
```

**Employee(const Employee&)**  
is used for copying

the object is **sliced**



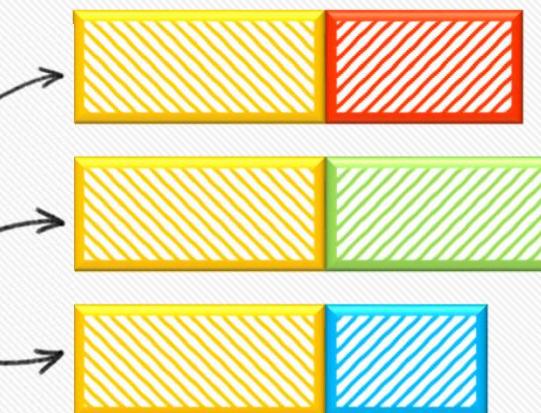
- ▶ When we wish to create an array (or any other container) of **mixed types**, we must **store pointers**

```
Employee* employees[] = {
    &engineer, &salesPerson, &manager
};

list<Employee*> employees2;
employees2.push_back(&engineer);
employees2.push_back(&salesPerson);
employees2.push_back(&manager);
```

**Array of pointers**

[0]
[1]
[2]



# Overriding Methods: Caveat

- ▶ Calling a function through a pointer (or reference) to a base class poses **a problem**:
  - ◆ The compiler has to choose which function to call **during compilation** ← this is called “static binding”
  - ◆ But the function that **should** be invoked is only known **at run-time**

```
Engineer engineer("Jane Doe");
SalesPerson salesPerson("John Doe");
Manager manager("Mr. Bossman");

Employee* employees[] = {
    &engineer, &salesPerson, &manager
};

for (int i = 0; i < 3; ++i) {
    employees[i]->giveRaise(100);
}
```

Which function should be called here?

Engineer::giveRaise(int)  
SalesPerson::giveRaise(int)  
Manager::giveRaise(int)  
Employee::giveRaise(int)

the compiler only knows that employees[i] is an Employee pointer, so Employee::giveRaise(int) is invoked!

# Overriding Methods

- ▶ How can we solve this problem?
  - ◆ First attempt: **explicitly determine the object's type at run-time** using a field with the object's type

```
class Employee {  
    string type;  
public:  
    // ...  
    Employee(string type) : type(type) {}  
    string getType() const;  
};
```

```
class Manager : public Employee {  
public:  
    // ...  
    Manager() : Employee("manager") {}  
};
```

```
Employee* employees[] = { ... };  
  
for (int i = 0; i < n; ++i) {  
    Employee *e = employees[i];  
    if (e.getType() == "manager") {  
        ((Manager*)e)->giveRaise(100);  
    }  
    // repeat for every type ...  
}
```

# static\_cast

- ▶ First improvement: replace C-style cast with C++ **static\_cast**
  - ◆ A static cast is **checked by the compiler**
  - ◆ If there is **no reasonable connection** between the two types, the code **will not compile**
    - Unlike C-style casts, which will compile with **undefined behavior**

```
Employee* employees[] = { ... };

for (int i = 0; i < n; ++i) {
    if (employees[i].getType() == "manager") {
        static_cast<Manager*>(employees[i])->giveRaise(100);
    }
    // repeat for every type ...
}
```

# static\_cast

- ▶ A **static\_cast** will only compile if the conversion **makes sense**

- ◆ Examples: int  $\leftrightarrow$  double, int\*  $\leftrightarrow$  void\*, Employee\*  $\leftrightarrow$  Engineer\*
- ◆ When converting to a **more specific type** (e.g., base class to derived class, void\* to int\*, ...), it's **up to the programmer** to ensure correctness

casting not  
needed

```
Manager* m = ...;  
Employee* emp = ...;  
Engineer* e = static_cast<Engineer*>(m);  
m = static_cast<Manager*>(emp);  
Employee* emp2 = m;
```

**compilation error:** cannot cast  
from Manager\* to Engineer\*

o.k.

Remember: in a **static\_cast** from type A to B, the compiler only checks that the conversion A->B can **potentially** make sense. It does NOT check that the argument is really of type B (and if it's not, the result is still **undefined!**)

# Virtual Functions

- ▶ Better solution: instead of “**asking**” each employee for its type, let’s simply “**tell**” it to invoke the right behavior
- ▶ For this we declare a function in the base class as **virtual**
  - ◆ A virtual function has **dynamic binding** (as opposed to **static binding**), which means the identity of the function is **determined at run-time**

```
class Employee {  
public:  
    // ...  
    virtual void giveRaise(int amount);  
};
```

```
class SalesPerson : public Employee {  
public:  
    // ...  
    void giveRaise(int amount);  
};
```

when `giveRaise()` is invoked -  
choose the correct function at run-  
time

```
SalesPerson salesPerson("John Doe");  
Employee* emp = &salesPerson;  
  
emp->giveRaise(1000);
```

calls `SalesPerson::giveRaise()`

# Virtual Functions

```
class Employee {  
public:  
    // ...  
    virtual void giveRaise(int amount);  
};
```

```
class SalesPerson : public Employee {  
public:  
    // ...  
    void giveRaise(int amount);  
};
```

```
class Engineer : public Employee {  
public:  
    // ...  
    void giveRaise(int amount);  
};
```

```
class Manager : public Employee {  
public:  
    // does not override giveRaise()  
};
```

```
Engineer engineer("Jane Doe");  
SalesPerson salesPerson("John Doe");  
Manager manager("Mr. Bossman");  
  
Employee* employees[] = {  
    &engineer, &salesPerson, &manager  
};  
  
for (int i = 0; i < 3; ++i) {  
    employees[i]->giveRaise(1000);  
}
```

this will automatically call the **correct function** for each object!

**Engineer**::giveRaise for employees[0]  
**SalesPerson**::giveRaise for employees[1]  
**Employee**::giveRaise for employees[2]

# The **override** keyword

- ▶ The **override** keyword allows to explicitly indicate that a function intends to override a virtual function of a base class
  - ◆ Using it guarantees a **compilation error** if there is no virtual function with a matching signature in the base class

```
class Employee {  
public:  
    virtual void giveRaise(int amount);  
};  
  
class QASpecialist : public Employee {  
    // ...  
public:  
    QASpecialist(int qualification);  
    void giveRaise(int amount) override;  
    // ...  
};  
  
void QASpecialist::giveRaise(int amount) {  
    Employee::giveRaise(amount * qualification);  
}
```

if class Employee has no  
function  
**void giveRaise(int amount)**, or  
if that function is not virtual,  
this line will **not compile**  
use the override keyword to:  
- improve **code readability**  
- avoid **nasty bugs**

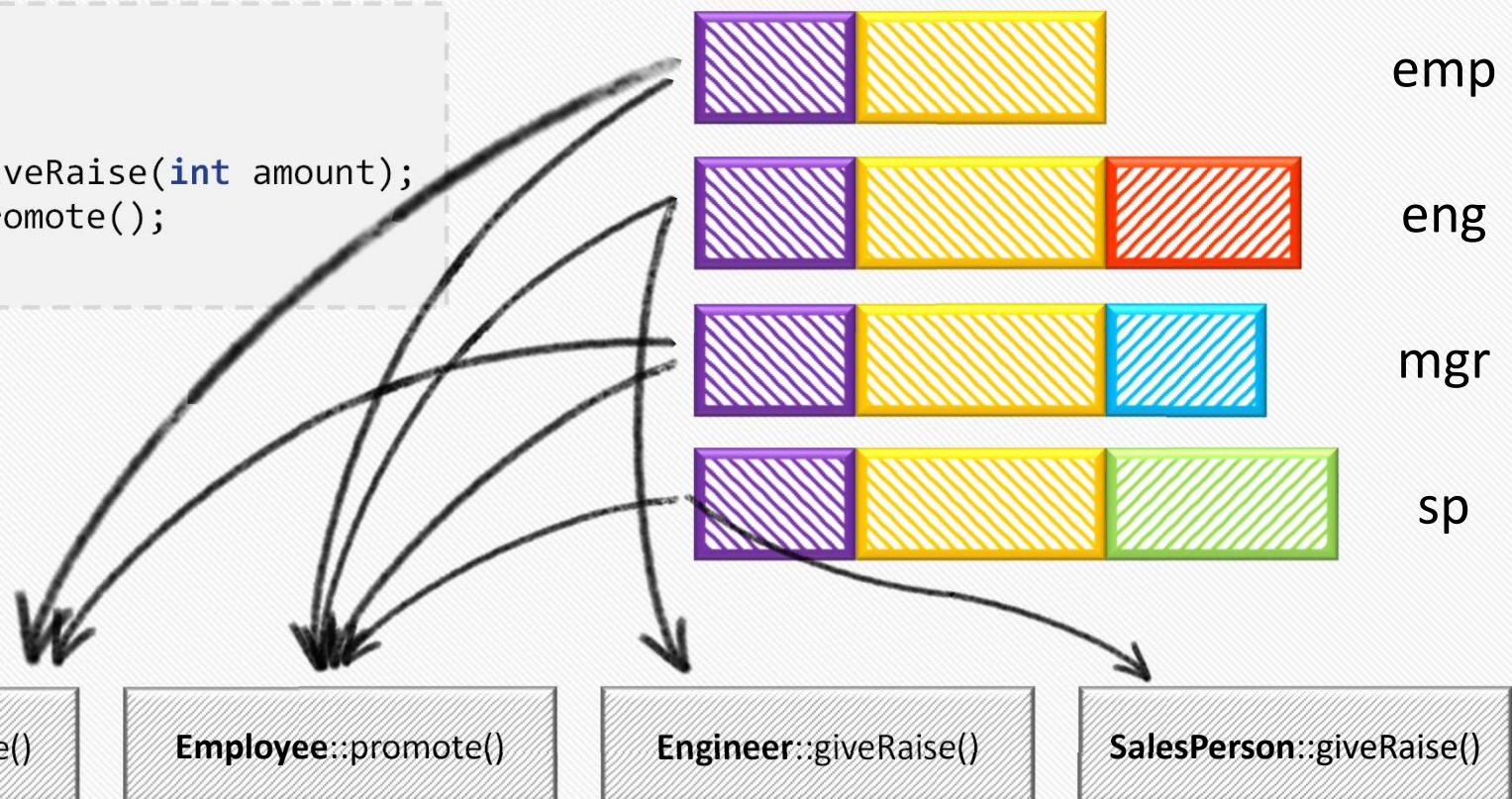
you are required to  
use **override** in our  
course

# Implementing Virtual Functions

- ▶ **First attempt:** Add a list of **function pointers** to each object when it is created

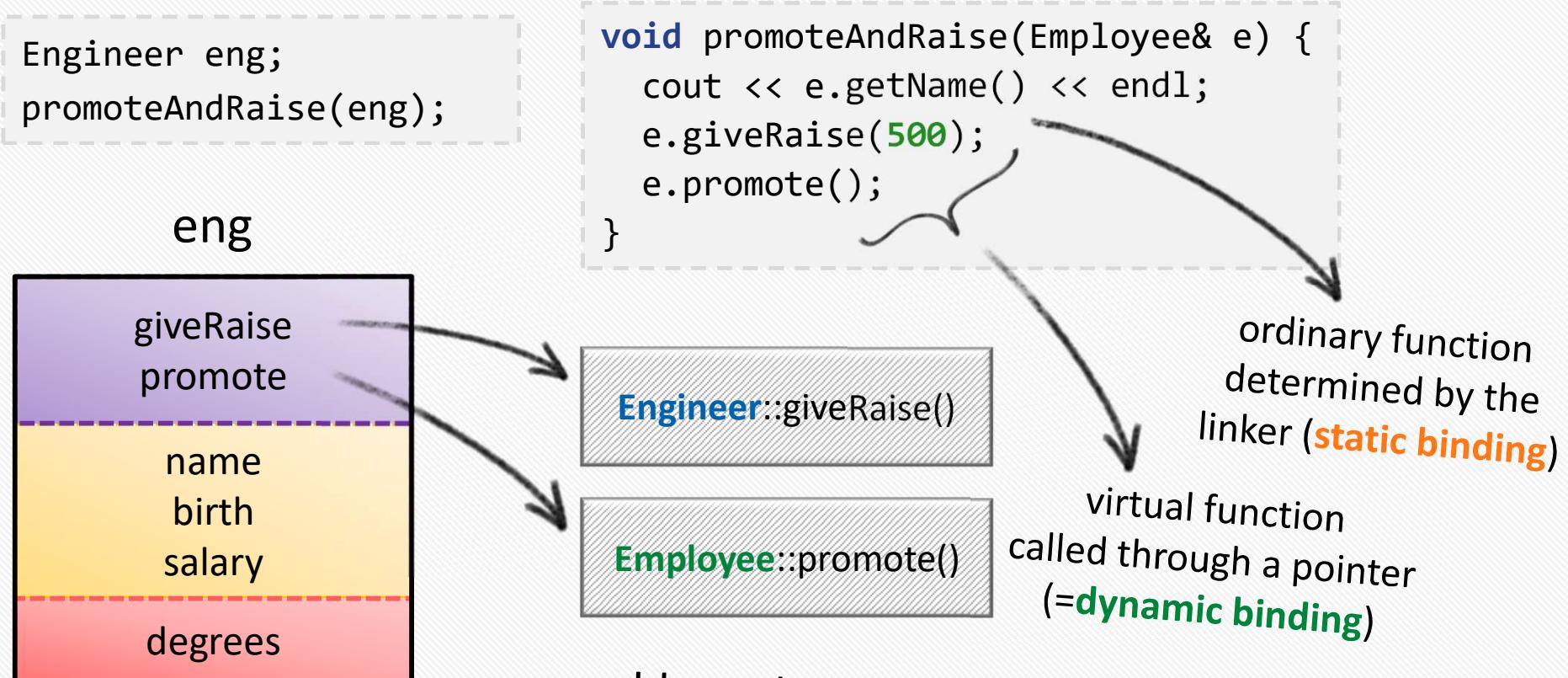
```
class Employee {  
public:  
    // ...  
    virtual void giveRaise(int amount);  
    virtual void promote();  
};
```

```
Employee emp;  
Engineer eng;  
Manager mgr;  
SalesPerson sp;
```



# Implementing Virtual Functions

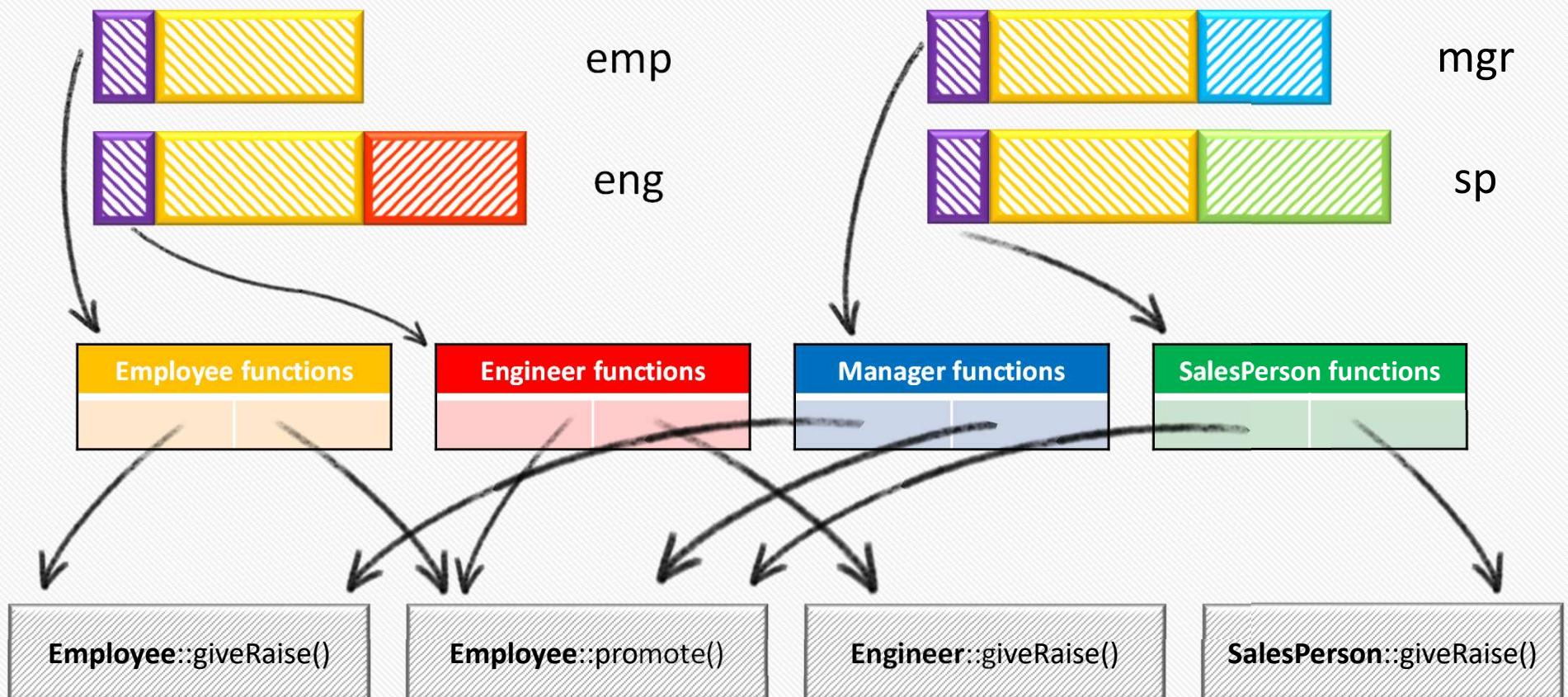
- ▶ **First attempt:** Add a list of **function pointers** to each object when it is created



problem: this solution **increases the memory** of each object by the number of virtual functions

# Implementing Virtual Functions

- ▶ **Solution:** create a list of function pointers **per class**
- ▶ Add **just one pointer** to each object, pointing to this list



# Virtual Function Table

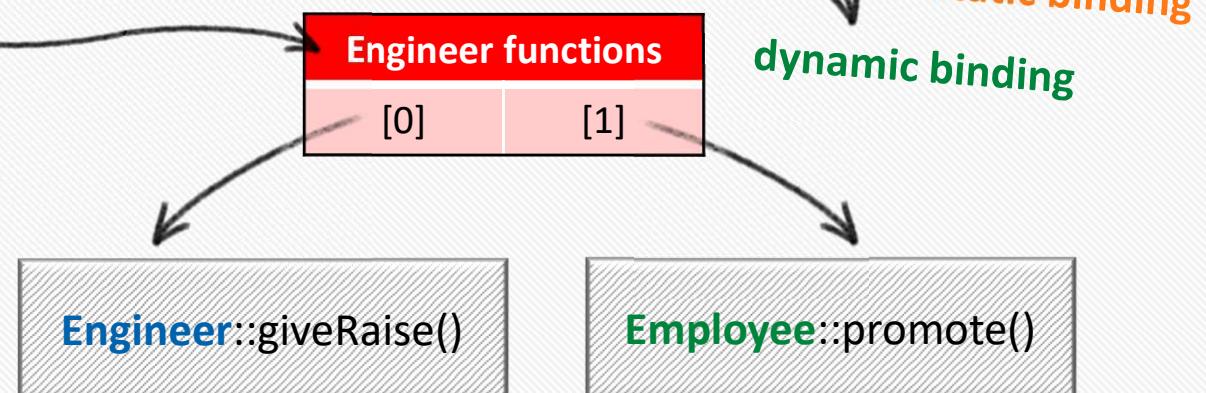
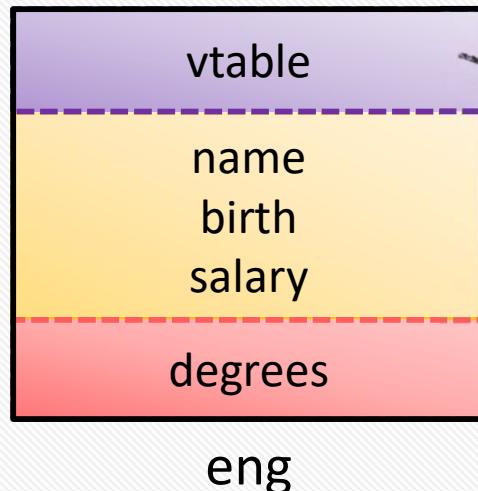
- For each class that has virtual functions, the program stores **an array of function pointers** to the correct functions
- Every time we create an object of such a class, the program **adds a pointer to its virtual table** at the beginning of the object

compilation

this is called the  
“**virtual table**”

```
void promoteAndRaise(Employee& e) {  
    cout << e.getName() << endl;  
    e.giveRaise(500);  
    e.promote();  
}
```

```
void promoteAndRaise(Employee& e) {  
    cout << e.getName() << endl;  
    e.vtable[0](500);  
    e.vtable[1]();  
}
```



# Abstract Classes

- ▶ In many cases, the base class represents an **abstract concept**
  - ◆ There is no reasonable implementation for some of the virtual functions
- ▶ In such cases, we can declare a **pure virtual function**
  - ◆ Pure virtual functions have **no implementation** in the base class

```
class Employee {  
public:  
    // ...  
    virtual void giveRaise(int amount) = 0;  
};
```

this function is not implemented for a general Employee

```
class SalesPerson : public Employee {  
public:  
    // ...  
    void giveRaise(int amount) override;  
};
```

SalesPerson provides a **concrete implementation** of giveRaise

# Abstract and Concrete Classes

- ▶ A class with **one or more pure virtual functions** is called an **abstract class**
  - ◆ Objects of an abstract class **cannot be created**
  - ◆ **Pointers** and **references** to abstract classes **can still be created**
- ▶ A derived class that **implements all the pure-virtual functions** becomes a **concrete class**, and objects can be created from this class
  - ◆ Otherwise, it remains **abstract**

```
class Employee {  
public:  
    // ...  
    virtual void giveRaise(int amount) = 0;  
};
```

```
class Manager : public Employee {  
public:  
    // ...  
    void giveRaise(int amount) override;  
};
```

```
Employee e; // error (abstract)  
Manager m; // o.k.
```

```
Employee* ptr = &m;  
Employee& ref = m;
```

# Constructors and Destructors

- ▶ Constructors are never virtual – we always use the constructor of the precise class we want to create
- ▶ Destructors must be virtual when using inheritance
  - ◆ This ensures that if we delete an object through a pointer to its base class, the correct destructor is called

```
class Employee {  
    // ...  
public:  
    virtual ~Employee() {}  
    // ...  
};
```

=default can also be used

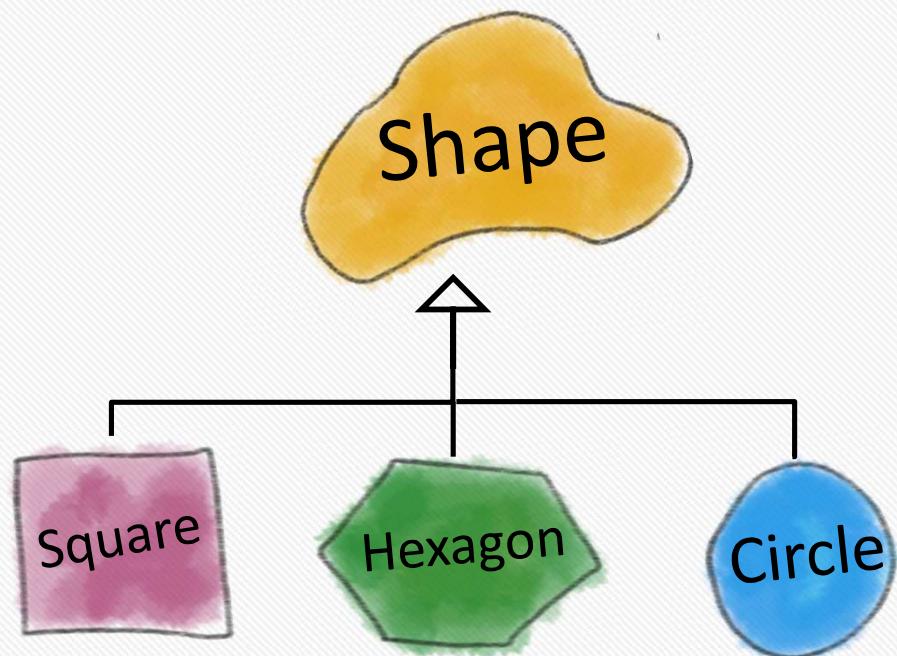
```
Employee* emp = new Engineer("John");  
  
// ...  
  
delete emp;
```

without virtual, only Employee::~Employee() would be called.  
with virtual, the correct d'tor Engineer::~Engineer() is called.

# Interfaces

- ▶ An abstract class that has **only pure virtual functions** is called an **interface**
- ▶ An interface describes the **general behavior** of a class, without specifying any concrete implementation

```
class Shape {  
public:  
    virtual void rotate(double angle) = 0;  
    virtual void draw() const = 0;  
    virtual double area() const = 0;  
};
```



# Interfaces

```
class Circle : public Shape {  
    double center_x, center_y, radius;  
public:  
    Circle(double cx, double cy, double rad) :  
        center_x(cx), center_y(cy), radius(rad) {}  
    void rotate(double angle) override {}  
    void draw() const override;  
    double area() const override {  
        return radius * radius * PI;  
    }  
};
```

```
class Square : public Shape {  
    double top, left, length;  
public:  
    Square(double top, double left, double len) {  
        top(top), left(left), length(len) {}  
    }  
    double area() const override {  
        return length * length;  
    }  
    // ...  
};
```

```
double sumArea(Shape* shapes[], int N) {  
    double totalArea = 0;  
    for (int i = 0; i < N; ++i) {  
        totalArea += shapes[i]->area();  
    }  
    return totalArea;  
}
```

we just “tell” the Shape to compute its area, and the Shape itself will know how to perform the right computation



# Polymorphism

- ▶ The **main use** of inheritance and virtual functions is to enable **polymorphic behavior**

```
class Employee {  
public:  
    virtual void giveRaise(int amount);  
    virtual double getYearlyCost() const;  
    virtual ~Employee() {}  
};
```

... and the code will even work  
for **future Employee types** that  
haven't been written yet!

```
double yearlyCost(const Employee* employees[],  
                  int n) {  
    double totalCost = 0.0;  
    for (int i = 0; i < n; ++i) {  
        totalCost += employees[i]->getYearlyCost();  
    }  
    return totalCost;  
}
```

we **don't need to know** the real type of  
each employee, we just ask it for its cost!

# Polymorphism

- ▶ **Polymorphism** – from Greek, meaning “**having multiple forms**”
- ▶ Polymorphic code works on all types that share a **common base class**, **without having to know the real type of each object**
- ▶ The polymorphic code will perform a different action depending on the exact type **passed to it at runtime**, without requiring **manual if's**

# Polymorphism and Modularity

- Without polymorphism, **code like this** might look familiar...

```
double sumArea(Shape* shapes[], int N) {  
    double totalArea = 0;  
    for (int i = 0; i < N; ++i) {  
        if (shapes[i]->type == CIRCLE) {  
            totalArea += circleArea(shapes[i]);  
        }  
        else if (shapes[i]->type == SQUARE) {  
            totalArea += squareArea(shapes[i]);  
        }  
        else if //...  
    }  
    return totalArea;  
}
```

- How much work is it to add a **new shape**?
  - Imagine if's like this in **dozens of places** in the code, in multiple files
  - How many **bugs** would this change introduce?

# Polymorphism and Modularity

- ▶ With polymorphism, we just **add a new class**
  - ◆ All existing Shape code will work out of the box

```
double sumArea(Shape* shapes[], int N) {  
    double totalArea = 0;  
    for (int i = 0; i < N; ++i) {  
        totalArea += shapes[i]->area();  
    }  
    return totalArea;  
}
```

```
class NewShape : public Shape {  
public:  
    double area() const override;  
    // ...  
};
```

- ▶ All shape-specific code is **localized in that shape's class**
  - ◆ Easy to read and understand the code
  - ◆ Easy to extend the code
  - ◆ Much easier to test and debug!

# Polymorphism

- ▶ Polymorphism can improve code:
  - ◆ Simplifies the code by separating type-specific details from the rest of the codebase
    - Most code will be oblivious to the specific type of the object
  - ◆ Allows extending the code in the future without changing existing code
- ▶ First hint to consider polymorphism:
  - ◆ The code contains if statements that select a different behavior based on the “type” of the object (e.g., if it’s a square, do X, else if it’s a circle...)

# Polymorphism and Copy Constructors

- Remember that polymorphism only works with **pointers and references**, never with “simple” variables

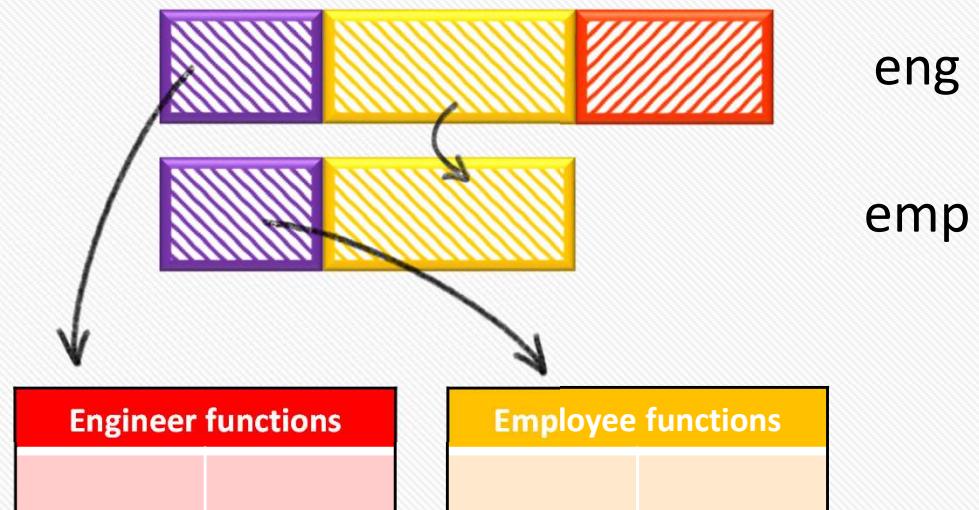
~~Engineer eng("Bjarne Stroustrup");  
Employee emp = eng;~~

Which copy c'tor is applied here?

Only the “Employee” part of eng is copied. This is called **slicing** and it **should be completely avoided**.



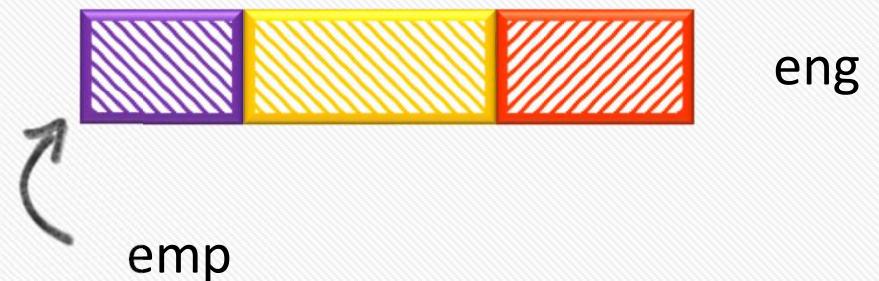
emp will even have the **virtual functions of an Employee**, **not an Engineer**, so all Engineer behavior is **lost** by this copy



# Polymorphism and Copy Constructors

- With a pointer or reference, the **original object is maintained**, along with all its virtual functions

```
✓ Engineer eng("Bjarne Stroustrup");
Employee* emp_ptr = &eng;
Employee& emp_ref = eng;
```



- Copy constructors are not useful with inheritance and polymorphism
  - But what if we do need to **create a copy** of an object?

# Object Cloning

- ▶ **Fact:** with inheritance and abstract classes, we usually store objects using a **pointer to the base-class type**
  - ◆ For example, a set of Shape\*
- ▶ **Question:** what if we need to create an **exact copy** of one of these objects?
- ▶ Let's give it a try...

somewhere in  
PowerPoint's code...

```
Shape* duplicate(const Shape* s) {  
    Shape* newShape = new Shape(*s); // compilation error!  
    newShape->moveX(10);  
    newShape->moveY(10);  
    newShape->show();  
    return newShape;  
}
```

# Object Cloning

- ▶ **Oh oh!** This will not work
  - ◆ We cannot directly construct a Shape object (it's abstract)
  - ◆ Shape doesn't even have a copy c'tor (or any c'tor in fact...)
- ▶ We need to figure out the **real type** of s, and call the copy constructor of **that type**
  - ◆ And do it without if's that **depend on the objects type**

```
Shape* duplicate(const Shape* s) {  
    Shape* newShape = ??? // new Circle(*s)? new Square(*s)?  
    newShape->moveX(10);  
    newShape->moveY(10);  
    newShape->show();  
    return newShape;  
}
```

# Object Cloning

- ▶ **Solution:** define a virtual `clone()` function in the base class
  - ◆ Sometimes referred to as a “virtual copy constructor”
  - ◆ At runtime, the **correct clone function** for the true object type will be called!

```
class Shape {  
    // ...  
public:  
    virtual Shape* clone() const = 0;  
    // ...  
};
```

```
class Circle : public Shape {  
    // ...  
public:  
    Circle(double radius);  
    Shape* clone() const override;  
    // ...  
};
```

```
Shape* Circle::clone() const {  
    return new Circle(*this);  
}
```

```
Shape* ptr = new Circle(3.0);  
  
Shape* copy = ptr->clone();
```



the true type of `*copy` will  
be `Circle` - the **same type** as  
`*ptr`

# Run Time Type Information

- ▶ Sometimes virtual functions are not enough and we must **query the type of an object**
  - ◆ For example, say we want to print all engineers and their degrees

```
class Engineer : public Employee {  
    // ...  
public:  
    void printDegrees() const;  
};
```

a function that **only**  
**engineers** have

```
void printDegrees(const Employee* employees[], int n)  
{  
    for (int i = 0; i < n; ++i) {  
        if (/** employees[i] is an Engineer */) {  
            cout << employees[i]->getName() << endl;  
            employees[i]->printDegrees();  
        }  
    }  
}
```

**Oh no!** This line will not compile since  
employees[i] is of type **Employee\***,  
which does not have a printDegrees()  
function

# Run Time Type Information

- ▶ We wish to avoid the dirty solution of adding a "**type field**" or a "**type method**":

```
class Employee {  
public:  
    virtual string type() const = 0;  
    // ...  
};
```

```
class Engineer : public Employee {  
    // ...  
public:  
    string type() const override {  
        return "engineer";  
    }  
};
```

```
void printDegrees(const Employee* employees[], int n)  
{  
    for (int i = 0; i < n; ++i) {  
        if (employees[i]->type() == "engineer") {  
            const Engineer* eng =  
                static_cast<const Engineer*>(employees[i]);  
            cout << eng->getName() << endl;  
            eng->printDegrees();  
        }  
    }  
}
```

# Run Time Type Information

- ▶ C++ allows querying the **dynamic type** of an object using the **dynamic\_cast** operator
- ▶ **dynamic\_cast** converts from a base class pointer to a derived class pointer, and is **checked for correctness at run-time**
  - ◆ If the cast is **correct**, we get a pointer with the real type of the object
  - ◆ If the cast **fails**, dynamic\_cast returns NULL

```
void printDegrees(const Employee* employees[], int n) {
    for (int i = 0; i < n; ++i) {
        const Engineer* eng = dynamic_cast<const Engineer*>(employees[i]);
        if (eng != NULL) {
            eng->printDegrees(); // OK - eng is of type Engineer*
        }
    }
}
```

- ▶ **Note:** dynamic\_cast will only work if the base class is **polymorphic**, i.e., it has **at least one virtual function**

# Casting in C++

- ▶ C-style casting is **unsafe**, and will allow dangerous casts without errors

```
const double height = 175.3;
int* height_ptr = (int*) &height;
```

- ▶ C++ defines **more specific** cast operations, that notify the compiler **what we want to do** so it can produce an **error** if we make a mistake
- ▶ **C-style casts** are still supported in C++, but their use should be minimized

# Casting in C++

- ▶ C++ defines four types of cast operations:
  - ◆ **static\_cast**: casts between types that have a known conversion between them, or between pointers/references that have an inheritance relation
    - Checked at compilation (not runtime)
    - When converting to a more specific type, it's **up to the programmer** to ensure correctness
    - Will **not remove const**
  - ◆ **dynamic\_cast**: casts between pointers/references that have an inheritance relation
    - The program checks correctness **at runtime**
    - Requires a **polymorphic** base class

# Casting in C++

- ▶ C++ defines four types of cast operations:

- ◆ **const\_cast**: removes const from a pointer or reference
  - Will not perform any additional conversion
  - Typically for compatibility with old code
  - Use with **caution**

```
void printStats(const int* arr, int n) {  
    cout << "Array sum: " <<  
        sumArray(const_cast<int*>(arr), n) << endl;  
    // ...  
}
```

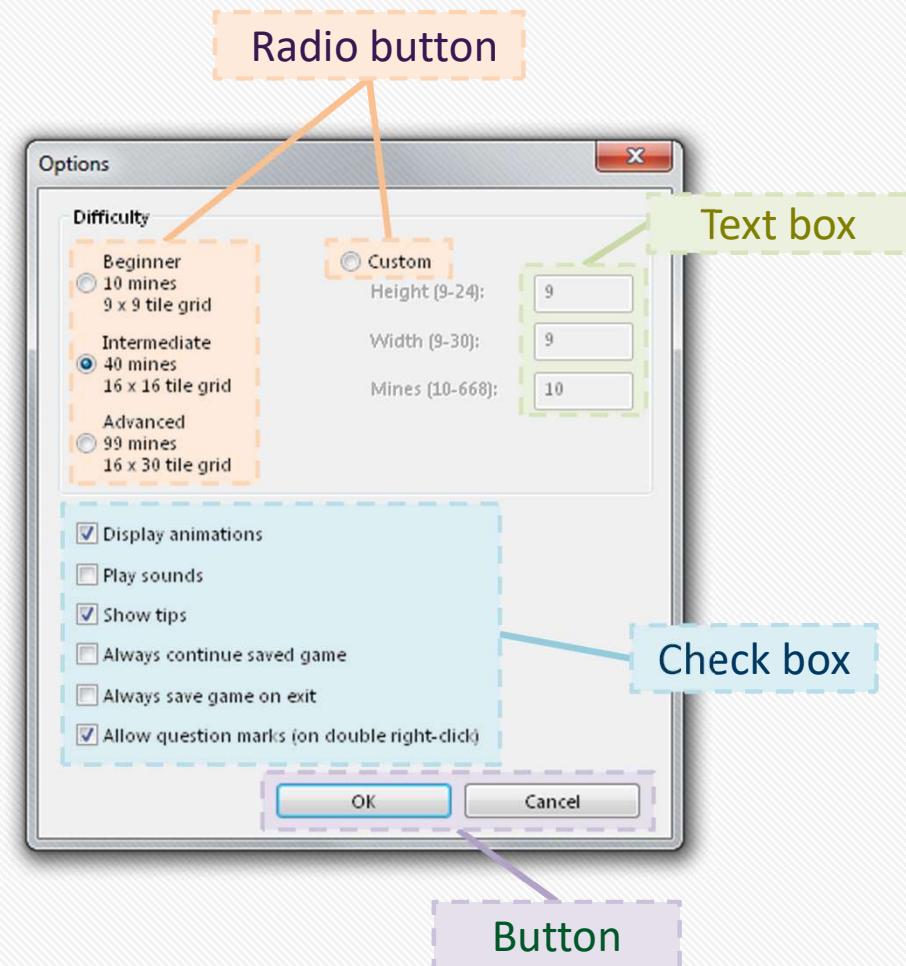
Some old C code

```
int sumArray(int* arr, int n);
```

- ◆ **reinterpret\_cast**: low-level conversion that reinterprets the memory bytes as a different type
  - Only for advanced operations and debugging
  - Not used in this course

# Inheritance Example: GUI

- ▶ A **Graphical User Interface (GUI)** defines various interface elements known as **widgets**
- ▶ A widget has some basic **common** properties
  - ◆ Width, height, position, ...
- ▶ However, widgets also have properties that are **specific** to them
  - ◆ A text box has a text value
  - ◆ A button has a function to be called when clicked
- ▶ A **window** object may contain multiple widgets



# Inheritance Example: GUI

- ▶ How should we design the system?
  - ◆ We want to be able to **store different widget types** in the same data structure/array
  - ◆ We want to avoid **code duplication** due to functionality that's common to all widgets
- ▶ We can have all concrete widget classes inherit from an abstract **Widget** base class
- ▶ We will be able to store and handle all Widgets using **the same code**

```
class Widget {  
    int x, y, width, height;  
public:  
    int getWidth() const;  
    int getHeight() const;  
    virtual void redraw() const = 0;  
    // ...  
};
```

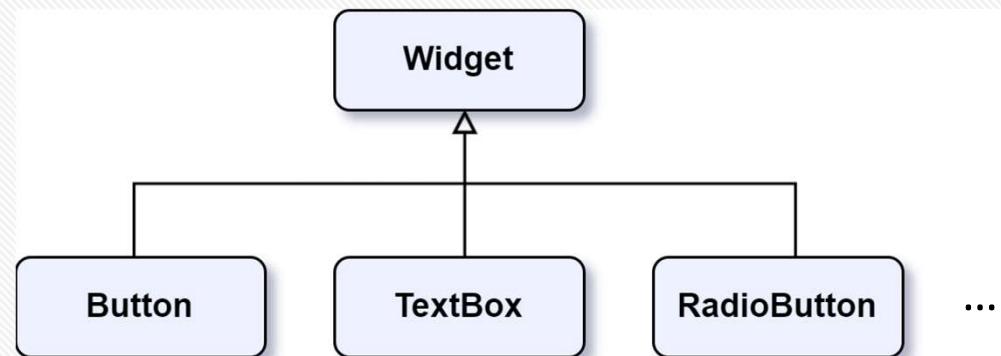
# Inheritance Example: GUI

- ▶ Let's look at some concrete widgets

```
class Button : public Widget {  
    // ...  
public:  
    Button(string label);  
    void onClick();  
    void redraw() const override;  
    // ...  
};
```

```
class TextBox : public Widget {  
    // ...  
public:  
    TextBox(int maxChars);  
    string getText() const;  
    void redraw() const override;  
    // ...  
};
```

- ▶ Our design so far:

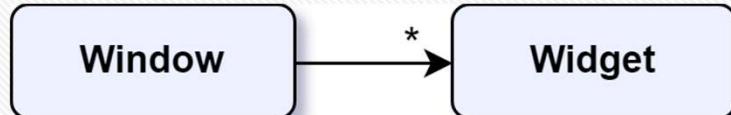


# Inheritance Example: GUI

- ▶ We can now write code that will handle **any Widget**:

```
class Window {  
    set<Widget*> children;  
    // ...  
public:  
    void redraw() const;  
};
```

```
void Window::redraw() const {  
    for (const Widget* widget : children) {  
        widget->redraw();  
    }  
};
```



adding a new widget type will  
require **no change** to Window's  
code!

- ▶ With polymorphism, we have no need to query the type of widget and call the correct function. We simply invoke `redraw()`, and the **object itself** will know what action to take

# Inheritance Example: GUI

- ▶ Let's now consider another part of the design:  
**what action** should the **Button** take on click?
  - ◆ The action should be supplied to the Button **during construction**, and called from the onClick() method
  - ◆ The **user** should be allowed to implement any action they need

```
class Button : public Widget {  
public:  
    void onClick(); // ???  
    // ...  
};
```

# Inheritance Example: GUI

- ▶ We can provide an **Action interface** that the **user** will implement!

```
class Action {  
public:  
    virtual void execute(int eventTime) const = 0;  
};
```

```
class Button : public Widget {  
    string label;  
    const Action* clickAction;  
public:  
    Button(string label, const Action* clickAction)  
        : label(label), clickAction(clickAction) {}  
  
    void onClick() {  
        clickAction->execute(currentTime());  
    }  
    // ...  
};
```



the user can implement a  
**file save action** for  
example

```
class SaveAction : public Action {  
public:  
    SaveAction(string filename)  
    void execute(int eventTime) const override;  
};
```

# Correct Usage of Inheritance

- ▶ Say we want to implement two classes:
  - ◆ A **list** of integers
  - ◆ A sorted **list** of integers

```
List list = { 3, 2, 4, 1, 5 };  
  
for (int item : list) {  
    cout << item << ", "  
}
```



**prints:** "3, 2, 4, 1, 5,"

```
SortedList list = { 3, 2, 4, 1, 5 };  
  
for (int item : list) {  
    cout << item << ", "  
}
```



**prints:** "1, 2, 3, 4, 5,"

- ▶ **Should** SortedList inherit from List?
  - ◆ Is SortedList a List?

# Correct Usage of Inheritance

- ▶ Let's look at the following **test code** for List:

```
void test_insert(List& list) {  
    list.insertFirst(10);  
    assert(*list.begin() == 10);  
}
```

```
List list = { 3, 2, 4, 1, 5 };  
test_insert(list); // this should pass  
  
SortedList sorted = { 3, 2, 4, 1, 5 };  
test_insert(sorted); // uh-oh!
```

**surprise: assertion fails!**



- ▶ This means that SortedList **is not** a List!
  - ◆ If SortedList were a kind of List, then it could be used **anywhere that a list can be used**
  - ◆ Clearly it cannot, because it fails a simple test that List passes

# Liskov Substitution Principle

- ▶ The reason SortedList is **not** a List is because some of List's functionality **does not apply to a SortedList**
  - ◆ For example, insertFirst() / insertLast() have no meaning in a SortedList because it keeps its elements sorted
- ▶ This is an example of the **Liskov Substitution Principle**:

Class **B** may inherit from class **A** only if type **B** can be used anywhere that a type **A** can be used.

↳ Should a **Square** inherit from a **Rectangle**? **Why not?**

# Inheritance vs Composition

- ▶ In the case of SortedList, the correct way to implement it without **duplicating List's code** is using **composition**
  - ◆ Composition = using an existing class as a **private member** of another class to simplify its implementation
  - ◆ Also referred to as a **has-a** or a **uses-a** relationship

```
class SortedList {  
    List elements;  
public:  
    int insert(int value);  
    void remove(int index);  
    int findLessEqual(int value) const;  
    // ...  
};
```

```
int SortedList::insert(int value) {  
    int id = findLessEqual(value);  
    if (id < 0) {  
        elements.insertFirst(value);  
        return 0;  
    }  
    elements.insertAfter(id, value);  
    return id + 1;  
}
```

- ▶ Composition is usually an implementation detail
  - ◆ To the user, List and SortedList are **unrelated types**

# When Should We Use Inheritance?

- ▶ Inheritance can be **easily misused** to create **confusing** and **complicated** code
- ▶ Use inheritance **when you need polymorphism**
  - ◆ Make sure you know **what code will benefit** from this design choice
- ▶ Otherwise, **prefer composition** over inheritance
  - ◆ A new class can always be implemented using an existing class as a member variable

*“There are two ways of constructing a software design:  
One way is to make it so simple that there are obviously  
no deficiencies, and the other way is to make it so  
complicated that there are no obvious deficiencies.”*

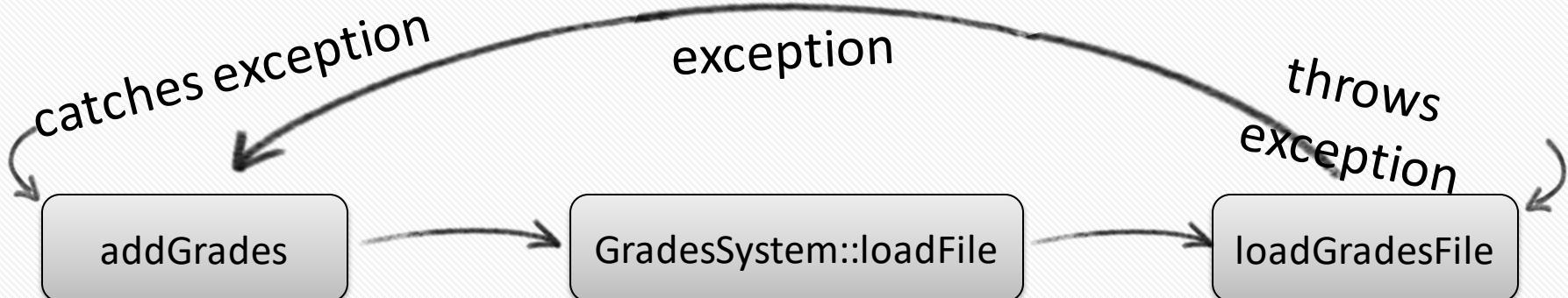
- C.A.R. Hoare, The 1980 ACM Turing Award Lecture

# Exceptions and Resource Management

Handling Errors and Resource  
Acquisition through Initialization

# Recap: Error Handling in C++

- ▶ C++ has a dedicated error handling system with **exceptions**
- ▶ A function that **detects a problem** but cannot handle it, **throws an exception**
- ▶ A higher-level function that **can handle the error**, but is not the one that detects it, will **catch** the exception
- ▶ Functions in between the two can usually **safely ignore it**



# Recap: Error Handling in C++

- ▶ A function that **throws an exception** immediately **stops** its execution, and starts **stack unwinding**.
- ▶ **Stack unwinding** – collapsing the stack until a try block, destructing local variables, parameters, etc.
- ▶ If the calling function does not catch the exception, it throws it upwards and the process **continues**.



# Defining Exceptions

- ▶ Exceptions are **caught** by their **type**
  - ◆ We will define **exception classes** that represent the different types of errors that can occur

```
class OutOfRange {  
    int index;  
public:  
    OutOfRange(int index) : index(index) {}  
    int getIndex() const { return index; }  
};
```

```
template<class T> class Array {  
    T* data;  
    int size;  
public:  
    // ... more methods ...  
    int size() const;  
    T& operator[](int index);  
};
```

```
template<class T>  
T& Array::operator[](int index) {  
    if (index < 0 || index >= size) {  
        throw OutOfRange(index); ←  
    }  
    return data[index];  
}
```

the illegal index will be available to the catcher

# Exceptions and Inheritance

- ▶ Exceptions can be **caught through their base class**
  - ◆ Just like function arguments
- ▶ We can **group several exceptions** under one base class
  - ◆ This will allow handling all these exceptions at once

```
class GradesException {  
public:  
    virtual ~GradesException() {}  
};
```

```
class NotEnrolled : public GradesException {};
```

```
class NoSuchStudent : public GradesException {};
```

```
void GradesSystem::printGrade(string student, int course) {  
    try {  
        cout << "grade is: " << getGrade(student, course);  
    } catch (const NotEnrolled&) {  
        cerr << studentName << " is not registered";  
    } catch (const GradesException&) {  
        cerr << "Data not available";  
    }  
}
```

GradesException

NotEnrolled

NoSuchStudent

This catches **any type of GradesException**, including future ones

Note the &:

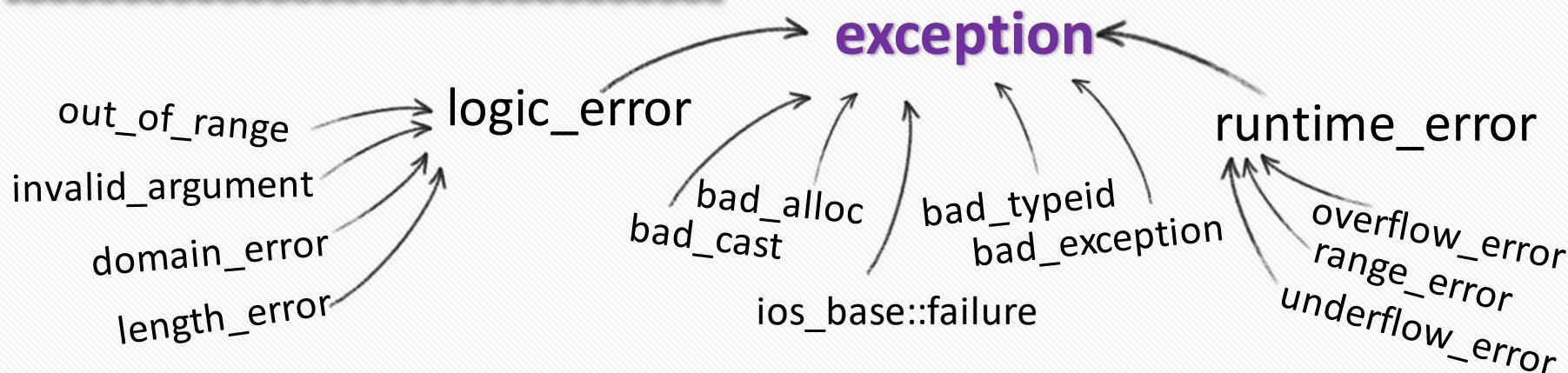
1. Avoids copying
2. Critical for **polymorphism**

# Exceptions and the Standard Library

- The standard C++ library defines a **basic hierarchy** of exception classes
  - Prefer **deriving your exceptions** from these classes

```
class exception {  
public:  
    exception();  
    exception(const exception&);  
    exception& operator=(const exception&);  
    virtual ~exception();  
    virtual const char* what() const;  
};
```

returns an error message that can be printed to the screen



# Exceptions and the Standard Library

- Deriving from `std::exception`, or from one of its derived classes, allows the user to catch "all exceptions" and still get some description of the error

at the very least, this makes debugging easier

if `all exceptions` are derived from `std::exception`, we can use `e.what()` for a descriptive error

here the caught object is unknown, so it **cannot be accessed**

```
int main() {
    try {
        f();
    } catch (const std::exception& e) {
        cerr << e.what() << endl;
    } catch (...) {
        cerr << "???" << endl;
    }
    return 0;
}
```

# Constructors and Exceptions

- ▶ A constructor will typically throw an exception if it cannot create a valid object
  - ◆ A common error in constructors is **out of memory**

```
template<class T>
Set<T>::Set() :
    data(new T[INITIAL_SIZE]),
    size(0),
    maxSize(INITIAL_SIZE)
{}
```

throws an **std::bad\_alloc** if  
the array cannot be allocated

no additional code  
required!

# Constructors and Exceptions

- ▶ If a constructor throws an exception, the object's destructor will **not be invoked**
  - ◆ If something was dynamically allocated, it must be **manually freed** before throwing the exception

```
template<class T>
Set<T>::Set(const Set<T>& set) :
    data(new T[set.getSize()]), size(set.getSize()), maxSize(set.getSize())
{
    try {
        for (int i = 0; i < size; i++) {
            data[i] = set.data[i];
        }
    } catch (const std::exception&) {
        delete[] data;
        throw;
    }
}
```

*T::operator=(const T&) calls  
leaving us with memory leak*

*we must free data explicitly,  
d'tor will not be called  
in case of an exception*

# Exception Safety

- ▶ Exceptions can **simplify code**, but we can still face the following problem.

Can something go wrong here?

```
template<typename T>
Set<T>& Set<T>::operator=(const Set<T>& set) {
    if (this == &set) {
        return *this;
    }
    delete[] data;
    data = new T[max(INITIAL_SIZE, set.getSize())];
    size = set.size;
    maxSize = max(INITIAL_SIZE, set.getSize());
    for (int i = 0; i < size; ++i) {
        data[i] = set.data[i];
    }
    return *this;
}
```

# Exception Safety

- ▶ We need to keep our objects in a **consistent state** in case of an exception:

if **new** fails, an exception is thrown and data is left **pointing to garbage**

```
template<typename T>
Set<T>& Set<T>::operator=(const Set<T>& set) {
    if (this == &set) {
        return *this;
    }
    delete[] data;
    data = new T[max(INITIAL_SIZE, set.getSize())];
    size = set.size;
    maxSize = max(INITIAL_SIZE, set.getSize());
    for (int i = 0; i < size; ++i) {
        data[i] = set.data[i];
    }
    return *this;
}
```

# Exception Safety

- ▶ Memory re-allocations should be handled the same way you **change apartments**:
  - ◆ Find a **new** apartment
  - ◆ **Move** all your stuff
  - ◆ Only then, **leave** the old apartment

```
template<typename T>
Set<T>& Set<T>::operator=(const Set<T>& set) {
    if (this == &set) {
        return *this;
    }
    T* temp_data = new T[max(INITIAL_SIZE, set.getSize())];
    for (int i = 0; i < size; ++i) {
        temp_data[i] = set.data[i];
    }
    delete[] data;
    data = temp_data;
    size = set.size;
    maxSize = max(INITIAL_SIZE, set.getSize())
    return *this;
}
```

*nothing is allowed to fail after this point*

Are we done?

# Exception Safety

- ▶ T can be any type, so T::operator=() might fail
  - ◆ We don't even know with **what exception!**

*T::operator=(const T&)*  
calls  
*T::operator=(const T&)*

if it fails, the object will remain in a **legal state** (unchanged), but we still can have a **memory leak!**

```
template<class T>
Set<T>& Set<T>::operator=(const Set<T>& set) {
    if (this == &set) {
        return *this;
    }
    T* temp_data = new T[max(INITIAL_SIZE, set.getSize())];
    for (int i = 0; i < size; ++i) {
        temp_data[i] = set.data[i];
    }
    delete[] data;
    data = temp_data;
    size = set.size;
    maxSize = max(INITIAL_SIZE, set.getSize())
    return *this;
}
```

# Exception Safety

- ▶ So, need to add **even more** error handling code...!

the code is correct, but...

**now we're back to  
square one!!!**

(and with worse syntax!)



we have to **explicitly** clean  
up and re-throw in  
functions that use **new**!

is there a **solution**?

```
template<class T>
Set<T>& Set<T>::operator=(const Set<T>& set) {
    if (this == &set) {
        return *this;
    }
    T* data_temp = new T[max(INITIAL_SIZE, set.getSize())];
    try {
        for (int i = 0; i < size; ++i) {
            temp_data[i] = set.data[i];
        }
    } catch (const std::exception&) {
        delete[] temp_data;
        throw;
    }
    delete[] data;
    data = temp_data;
    size = set.size;
    maxSize = max(INITIAL_SIZE, set.getSize())
    return *this;
}
```

# Resource Acquisition is Initialization

- ▶ There is an **important feature of C++** which can help us:
  - ◆ Destructors for local variables are **called automatically** when an exception is thrown
- ▶ We should **use destructors** to clean up after us instead of doing so **manually**

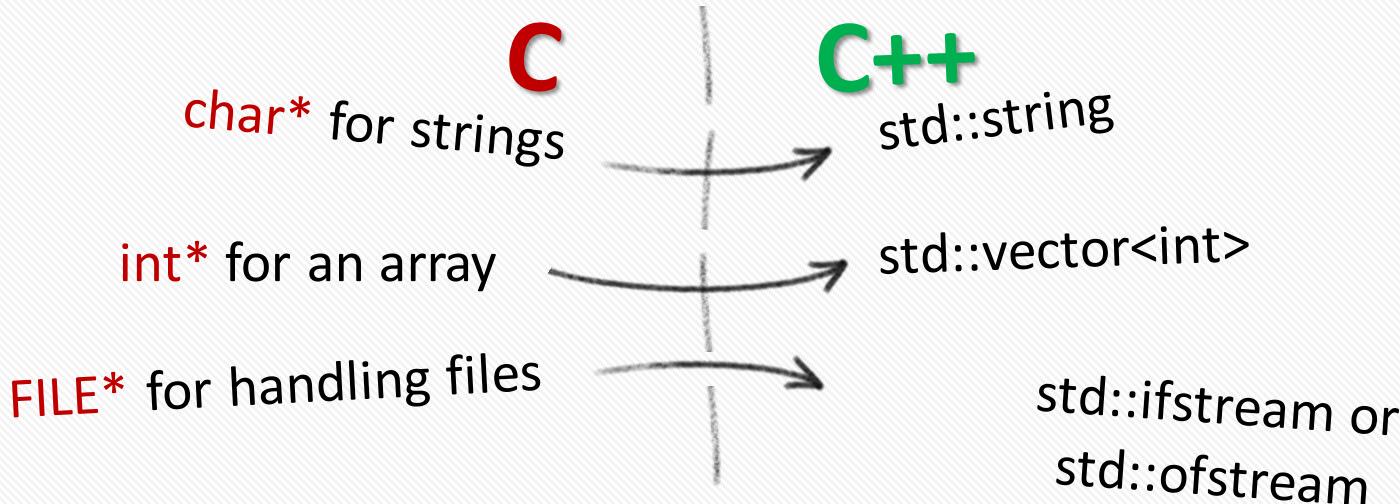
```
void bad(const char* str) {  
    char* str1 = new char[strlen(str) + 1];  
    strcpy(str1, str);  
    int* array = new int[10];  
  
    // ... code that may throw exceptions  
  
    delete[] str1;  
    delete[] array;  
}
```

```
void good(const char* str) {  
    string str1(str);  
    std::vector<int> array(10);  
  
    // ...  
    // returning or throwing is safe!  
    // ...  
}
```

# Resource Acquisition is Initialization

- ▶ Good C++ code follows the **Resource Acquisition is Initialization** design pattern (**RAII**)
  - ▶ When allocating a resource, use an **object that manages that resource**
    - ◆ This allows the program to **clean up after us automatically**

this is the **single most important technique** for good programming in C++



# Resource Acquisition is Initialization

- ▶ **RAII** means **using an object with a destructor** for **any operation** that requires cleaning up
  - ◆ No need to explicitly **manage memory**
    - Or files, or any other resource
  - ◆ **Explicit catching and re-throwing is rare** – exceptions can usually be propagated upwards without interference

- ▶ What about **polymorphism**?
  - ◆ Polymorphism requires **pointers**
  - ◆ Pointers **do not clean up after themselves**

Can throw an exception

using pointers to the base class is very common

```
void bad() {  
    Employee* e = new Engineer("David");  
    try {  
        f();  
    } catch (const std::exception&) {  
        delete e;  
        throw;  
    }  
}
```

# Smart Pointers

- ▶ We solve this problem using **better pointers**
  - ◆ Instead of using a simple C pointer, we will use **smart pointer classes** which automatically **delete** their data when they are destructed.

```
void good() {  
    smart_ptr<Employee> emp(new Engineer("David"));  
    // safe to use code that throws exceptions!!  
    emp->giveRaise(500); // our smart_ptr has operator->()  
    f(); // no problem if this throws an exception  
    // the engineer will be automatically deleted  
}
```

*Employee*'s destructor will automatically delete our engineer whenever **good()** terminates – whether normally or due to an exception

# A Simple Smart Pointer

- ▶ Creating a basic smart pointer is easy
  - ◆ We will avoid copying and assignment behavior for now

```
template<class T>
class smart_ptr {
    T* data;
public:
    explicit smart_ptr(T* ptr = nullptr);
    smart_ptr(const smart_ptr&) = delete;
    smart_ptr& operator=(const smart_ptr&) = delete;
    ~smart_ptr();
    T& operator*() const;
    T* operator->() const;
};
```

```
template<class T>
smart_ptr::smart_ptr(T* ptr)
    : data(ptr) {}
```

```
template<class T>
smart_ptr::~smart_ptr() {
    delete data;
}
```

```
template<class T>
T& smart_ptr::operator*() const {
    return *data;
}
```

```
template<class T>
T* smart_ptr::operator->() const {
    return data;
}
```

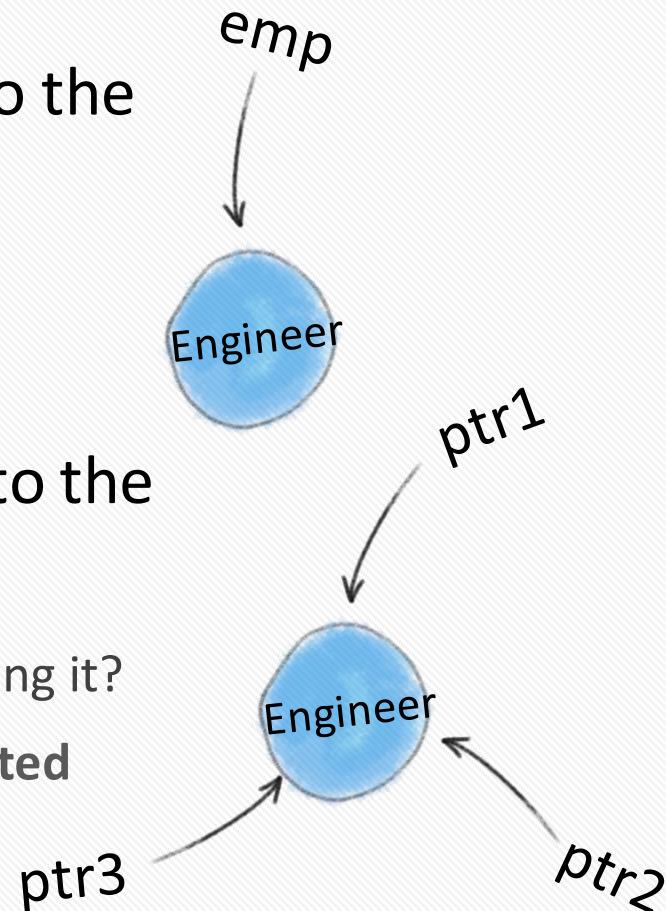
**Reminder:** operator-> must return a pointer or something that has an operator->

# Smart Pointers

- ▶ The **automatic** destruction of the object by our smart pointer:
  - ◆ **Significantly** reduces **memory leaks** caused by forgetting to delete objects
  - ◆ Solves **ownership** problems: An object may be allocated in one place, then passed to another, and so on... **Who should delete the object?** With smart pointers, this is taken care of **automatically**

# Smart Pointers

- ▶ So how should **copying** and **assigning** smart pointers behave?
- ▶ **Option 1:** Allow **only one pointer** to the allocated object
  - ◆ And **disallow** copying and assigning
  - ◆ Implemented in C++ by **std::unique\_ptr**
- ▶ **Option 2:** Allow **multiple pointers** to the same object
  - ◆ But then, which one is in charge of releasing it?
    - ◆ Should be the **last one that is destructed**
  - ◆ Implemented in C++ by **std::shared\_ptr**



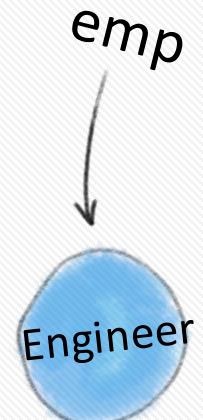
# std::unique\_ptr

- ▶ **std::unique\_ptr** represents a smart pointer that uniquely owns its object
  - ▶ It is clear who **owns** the object, who is **responsible** for destroying it, and **when** it will be destroyed (when its owner is destructed!)
  - ▶ It's a **highly efficient** smart pointer that has **no overhead** compared to a normal C pointer

```
#include <memory>

void f() {
    std::unique_ptr<Employee> emp(new Engineer("David"));
    emp->giveRaise(500);

    // ... safe to return or to use code that can throw exceptions
}
```



# std::unique\_ptr

- ▶ **std::unique\_ptr** cannot be copied or assigned
  - ◆ Achieved by deleting the copy c'tor and assignment operator (like we did with smart\_ptr<T>)
- ▶ However, we can **transfer ownership** between two unique\_ptrs if needed
  - ◆ The original owner will point to nullptr

```
unique_ptr<int> p1(new int(5));
unique_ptr<int> p2 = p1;    // compilation error
unique_ptr<int> p3 = std::move(p1); // OK!
```

part of a large subject called  
**move semantics** that is not  
covered in this course

p3 **gains ownership** of the int,  
p1 now points to **nullptr**

# std::unique\_ptr

- ▶ A function can also allocate an object and **return a unique\_ptr** to it
  - ◆ Ownership is transferred **automatically** to the receiving variable

```
unique_ptr<Employee> createEngineer(const char* name, int salary) {  
    unique_ptr<Engineer> eng(new Engineer(name));  
    eng->setSalary(salary);  
    return eng;  
}
```

yes, a unique\_ptr<Engineer> is also a  
unique\_ptr<Employee>, like ordinary C  
pointers

```
unique_ptr<Employee> emp = createEngineer("David", 10000);
```



no need for std::move() here,  
ownership is transferred  
automatically

# `std::unique_ptr`

- ▶ Can we pass a `unique_ptr` to a function?
  - ◆ Not by value, since it does not have a copy c'tor
- ▶ But a function will typically just need to **access the object**, not own or destruct it
  - ◆ We can just use `ordinary C pointers` for this!

# std::unique\_ptr

- ▶ A **C pointer** that points to an object but is **not responsible for deleting it** is called a **non-owning pointer**
  - ◆ They are still very useful, even with smart pointers

```
void printEmployee(const Employee* e) {  
    cout << "Name: " << e->getName() << endl;  
    // ...  
}
```

this function can be used with **any** smart pointer **or** C standard pointer

```
unique_ptr<Employee> emp = createEngineer("David", 10000);  
printEmployee(emp.get());
```

get() returns a standard C pointer to the object

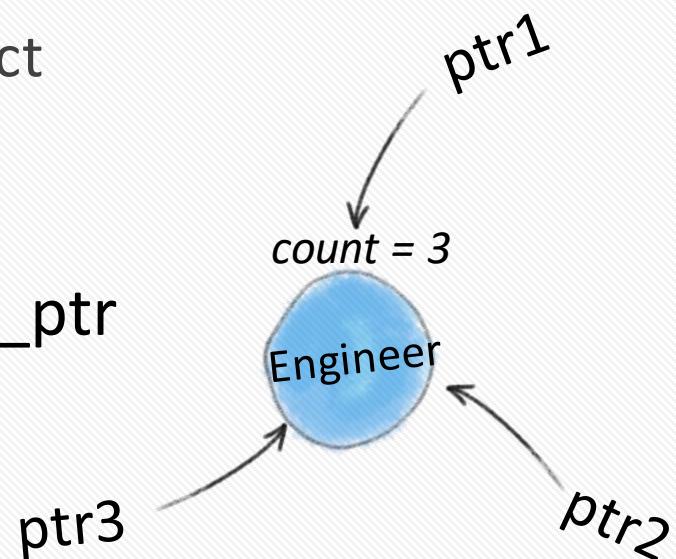
# std::unique\_ptr

- ▶ **unique\_ptr** can be used as a **member variable** of a class
  - ◆ However, the object will **not be copyable or assignable**
    - ◆ Since unique\_ptr does not have those operations
  - ◆ We can implement those operations **manually** if needed (and duplicate the unique member for each object)

```
class ChessGame {  
    vector<Piece> m_whitePieces;  
    vector<Piece> m_blackPieces;  
    unique_ptr<ChessBoard> m_gameBoard;  
    // ...  
}
```

# `std::shared_ptr`

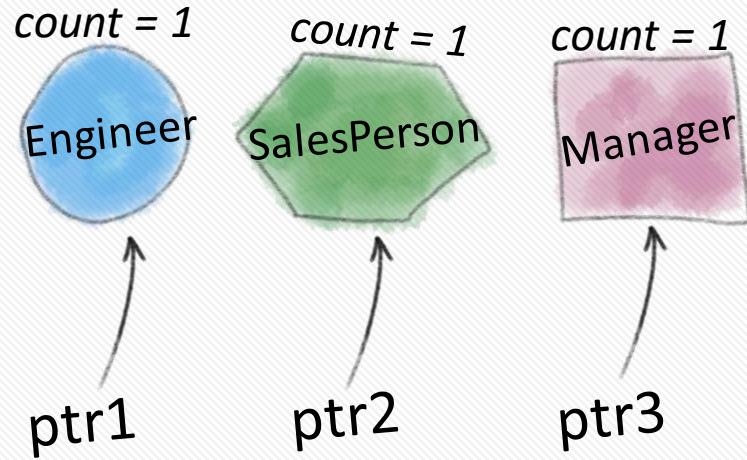
- ▶ `std::shared_ptr` is a smart pointer that allows **multiple** smart pointers to point to the same object
  - ◆ They all **share ownership** of that object
- ▶ `shared_ptr` uses **reference counting**
  - ◆ It maintains a **counter** of all smart pointers referring to the same object
  - ◆ When the counter reaches 0, the object is released
- ▶ It has more overhead than `unique_ptr`



# std::shared\_ptr

- ▶ **std::shared\_ptr** is useful when we need multiple objects to refer to a **shared** object, and we do not know **which of them will be destroyed last**
  - ◆ The object will exist as long as any of these objects exists
  - ◆ Once the last one is destroyed, the object is destructed

```
shared_ptr<Employee> f() {  
    shared_ptr<Employee> ptr1(new Engineer("Dave"));  
    shared_ptr<SalesPerson> ptr2(new SalesPerson(...));  
    shared_ptr<Employee> ptr3(new Manager("John"));  
  
    ptr3 = ptr2;  
  
    cout << ptr3->getYearlyCost() << endl;  
  
    Employee& emp = *ptr1;  
    return ptr3;  
}
```

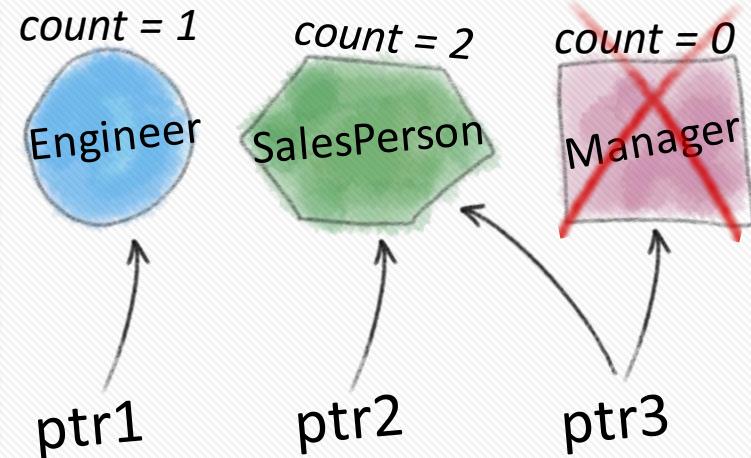


same syntax as pointers!

# std::shared\_ptr

- ▶ **std::shared\_ptr** is useful when we need multiple objects to refer to a **shared** object, and we do not know **which of them will be destroyed last**
  - ◆ The object will exist as long as any of these objects exists
  - ◆ Once the last one is destroyed, the object is destructed

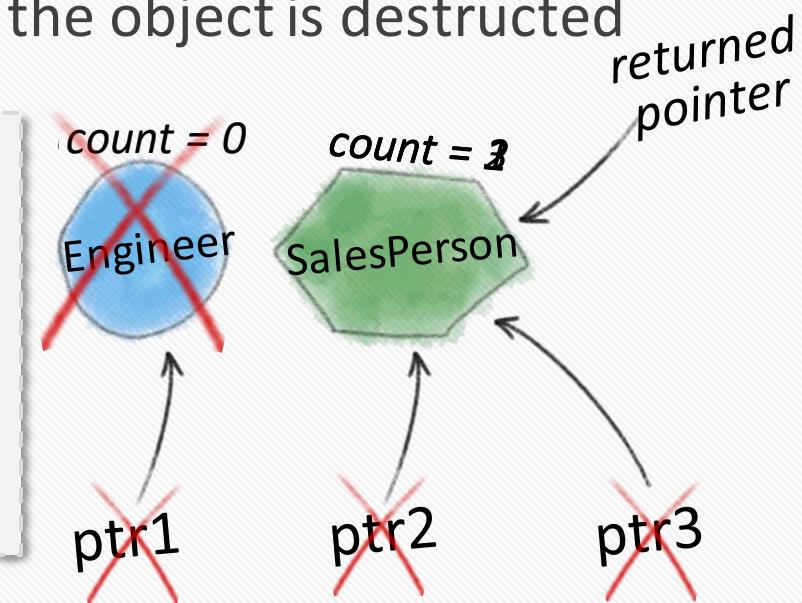
```
shared_ptr<Employee> f() {  
    shared_ptr<Employee> ptr1(new Engineer("Dave"));  
    shared_ptr<SalesPerson> ptr2(new SalesPerson(...));  
    shared_ptr<Employee> ptr3(new Manager("John"));  
  
    ptr3 = ptr2;  
  
    cout << ptr3->getYearlyCost() << endl;  
  
    Employee& emp = *ptr1;  
    return ptr3;  
}
```



# std::shared\_ptr

- ▶ **std::shared\_ptr** is useful when we need multiple objects to refer to a **shared** object, and we do not know **which of them will be destroyed last**
  - ◆ The object will exist as long as any of these objects exists
  - ◆ Once the last one is destroyed, the object is destructed

```
shared_ptr<Employee> f() {  
    shared_ptr<Employee> ptr1(new Engineer("Dave"));  
    shared_ptr<SalesPerson> ptr2(new SalesPerson(...));  
    shared_ptr<Employee> ptr3(new Manager("John"));  
  
    ptr3 = ptr2;  
  
    cout << ptr3->getYearlyCost() << endl;  
  
    return ptr3;  
}
```



# std::shared\_ptr – Notes

- ▶ The basic method of reference counting fails if **cycles are created**
  - ◆ Most classes don't cause such cycles though

```
struct Node {  
    shared_ptr<Node> next;  
    int data;  
};
```

```
void memory_leak() {  
    shared_ptr<Node> ptr(new Node());  
    ptr->data = 1;  
    ptr->next = ptr;  
}
```

count increases to 2

count decreases to 1 -  
**delete will not be  
called**

- ▶ There's a price in efficiency for using shared\_ptr
  - ◆ However, it's often **negligible**
- ▶ Don't use shared\_ptr when **precise memory management** is needed
  - ◆ For example, for managing nodes of a linked list

# std::make\_shared/unique

- ▶ Another way to initialize smart pointers

- ◆ Instead of

```
shared_ptr<Engineer> ptr1(new Engineer("Dave"));
```

- ◆ We can do

```
shared_ptr<Engineer> ptr1 = make_shared<Engineer>("Dave");
```

- ◆ Instead of

```
unique_ptr<Engineer> ptr2(new Engineer("Dave"));
```

- ◆ We can do

```
unique_ptr<Engineer> ptr1 = make_unique<Engineer>("Dave");
```

# Choosing the right pointer

- ▶ The choice of smart pointer should reflect **who owns the object**
  - ◆ In many cases, there is **just one owner**
    - All other objects can just receive a simple pointer to it

```
class MSOutlook {  
    unique_ptr<Mailbox> m_mailbox;  
    unique_ptr<Calendar> m_calendar;  
public:  
    ...  
}
```

```
class MeetingInvitation {  
    const Calendar* m_calendar;  
    DateTime m_startTime;  
    TimeDuration m_length;  
public:  
    bool isTimeSlotAvailable() const;  
}
```

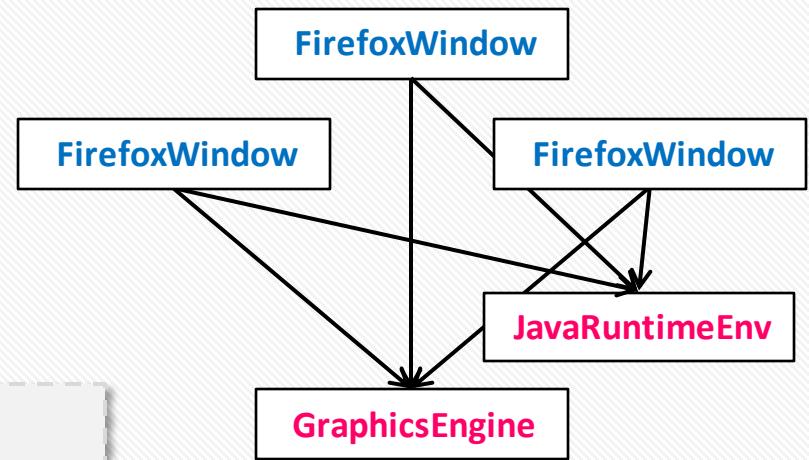
↑ **owns** the Calendar  
just needs **access** to it

# Choosing the right pointer

- When we have multiple owning of an object

```
class FirefoxWindow {  
    shared_ptr<GraphicsEngine> m_renderer;  
    shared_ptr<JavaRuntimeEnv> m_javaServer;  
public:  
    ...  
}
```

```
class GraphicsEngine {  
public:  
    void DrawShape(const Shape*, FirefoxWindow*) const;  
    ...  
}
```



once the last Firefox window is closed, the **shared** GraphicsEngine and Java Runtime Environment will be **automatically destructed**

# Summary

- ▶ Exceptions are **not a silver bullet**
  - ◆ Bugs in handling errors are still common in real software
- ▶ But exceptions **keep code from inflating** and when used correctly **simplify our work**
  - ◆ Which allows creating more complex software
- ▶ **RAII** is a necessary technique for writing big software in C++
  - ◆ Frees us from having to constantly handle and debug resource management problems
  - ◆ Using **unique\_ptr** and **shared\_ptr** allows us to safely store objects in pointers, and simplifies the usage of polymorphism

*“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”*

- Brian Kernighan

# Design

---

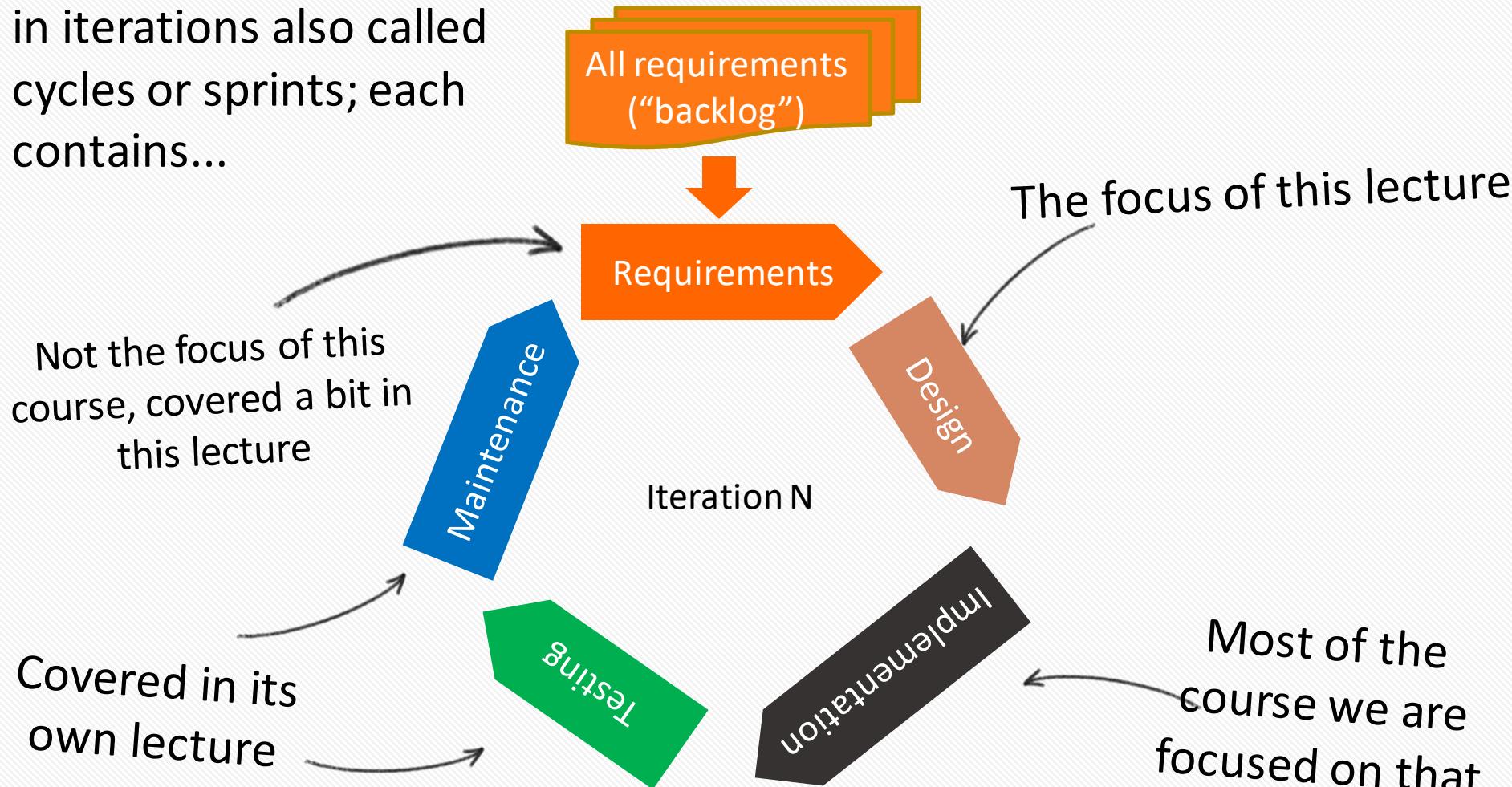
- ❖ General design issues
- ❖ UML sequence diagram
- ❖ Design patterns

# Design

- ▶ So... Let's recap:
- ▶ No software project can be done just by coding
  - ◆ Complicated
  - ◆ hard to developed
  - ◆ hard to cooperate among many team members
  - ◆ Hard to maintain
- ▶ Impossible to know all the details in advance
- ▶ With time and better understanding of the project, requirements and code design change and improve

# Development model

Projects are developed in iterations also called cycles or sprints; each contains...

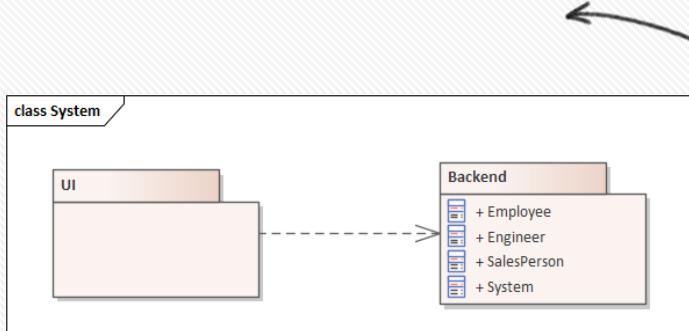


# Design

- ▶ When a project is **big**, we need to divide it into **packages/modules**.
  - ◆ We need to design the interaction between packages
  - ◆ For each such module, we build its own unified modelling language (UML) diagrams
  - ◆ Then do the coding. Usually each has its own namespace.
- ▶ In a lot of cases a software involves **interacting with other systems**, or a hardware/device. We need to describe that also.

# Employee's handling system

- ▶ As an example, let's go back to our **employee's handling system**. For cycle 1, we start with the following requirements:
  - ◆ Build a system for handling the company employees.
  - ◆ The system will contain a **user interface** and a **remote backend** side.

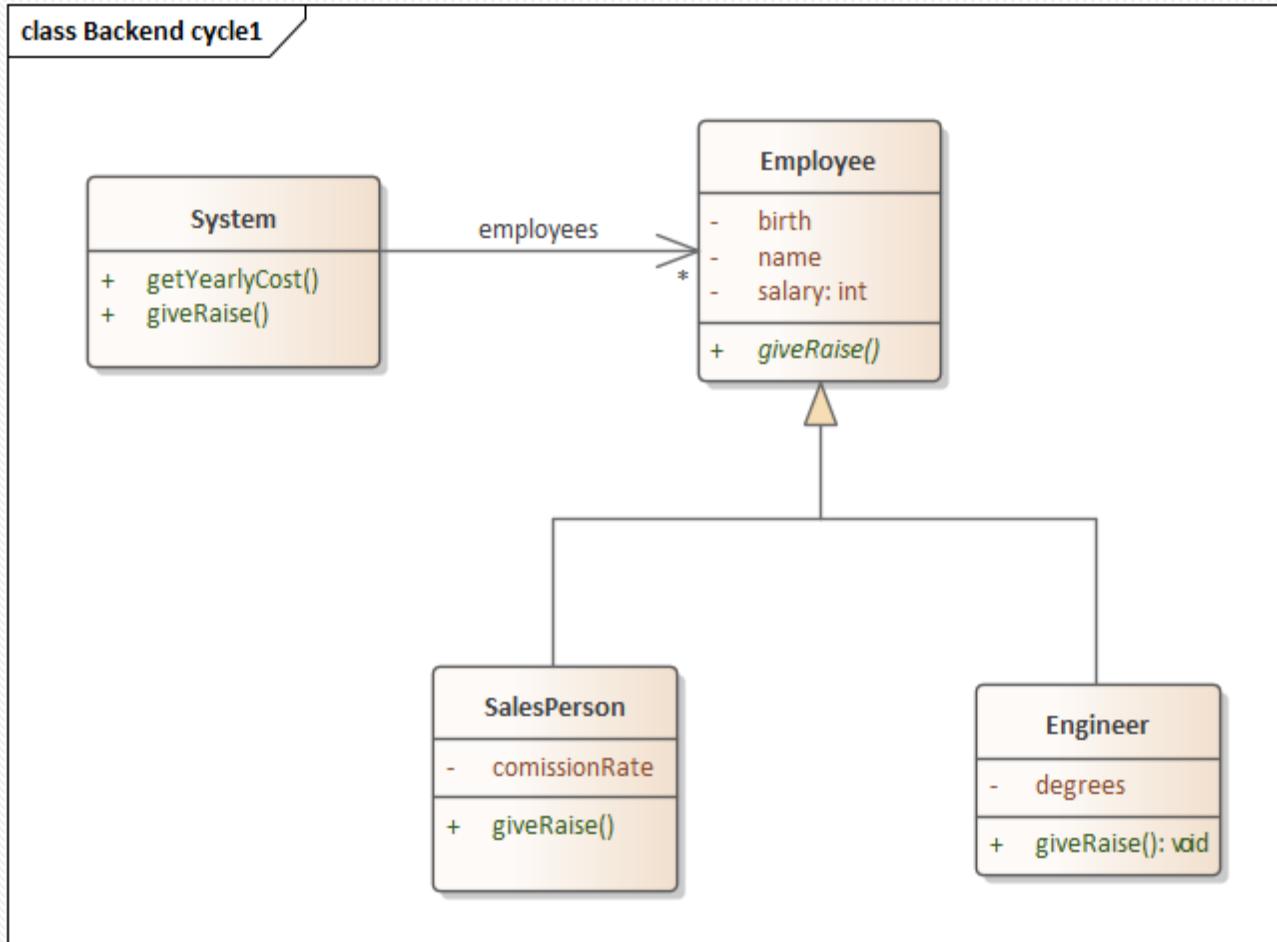


As for the course limitations we will focus only on the backend side, and not handle the UI and the interaction between it and the backend side.

# Employee's handling system

- ▶ For cycle 1, we start with the following requirements for the backend module:
  - ◆ Handle all company employees.
  - ◆ Each employee has a name, birth date and basic salary.
  - ◆ Two types of employees: salesperson and engineer
  - ◆ Compute the company yearly cost
  - ◆ Give a raise/bonus to all employees. Each has a different way of calculating the bonus:
    - Engineer gets a bonus using their degrees
    - Salesperson gets a bonus using their commission rate

# Cycle 1



# Cycle 1

```
void System::giveRaise
{
    for each employee in employees {
        employee->giveRaise();
    }
}
```

```
float System::getYearlyCost
{
    float cost = 0;
    for each employee in employees {
        cost += employee->getSalary();
    }
    return cost;
}
```



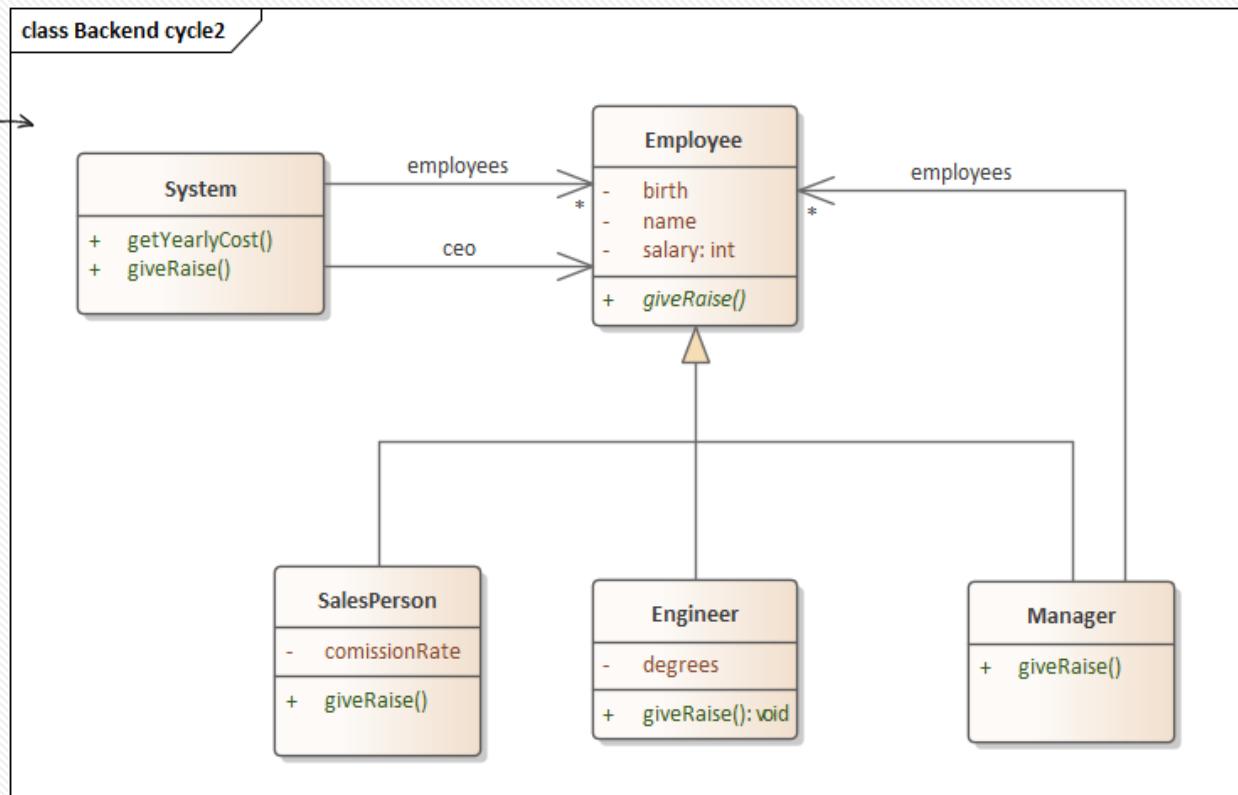
Only a **pseudo code**, at this design stage, the exact code and the type of the collections we use are less important.

- ▶ We finish designing, coding and testing
- ▶ We show it to our client and then ...

# Cycle 2

- ▶ A new requirement: adding manager
- ▶ Each manager has its own workers
- ▶ Each manager decides whether to give a bonus to their employees

We didn't have to  
change the current  
design, only add to it !!!  
A big advantage of  
object-oriented design



# Cycle 2

```
void System::giveRaise
{
    CEO->giveRaise();
}
```



A minimal change to giveRaise.  
Now we start from the CEO which calls giveRaise to its employees and from here it propagates down the tree.

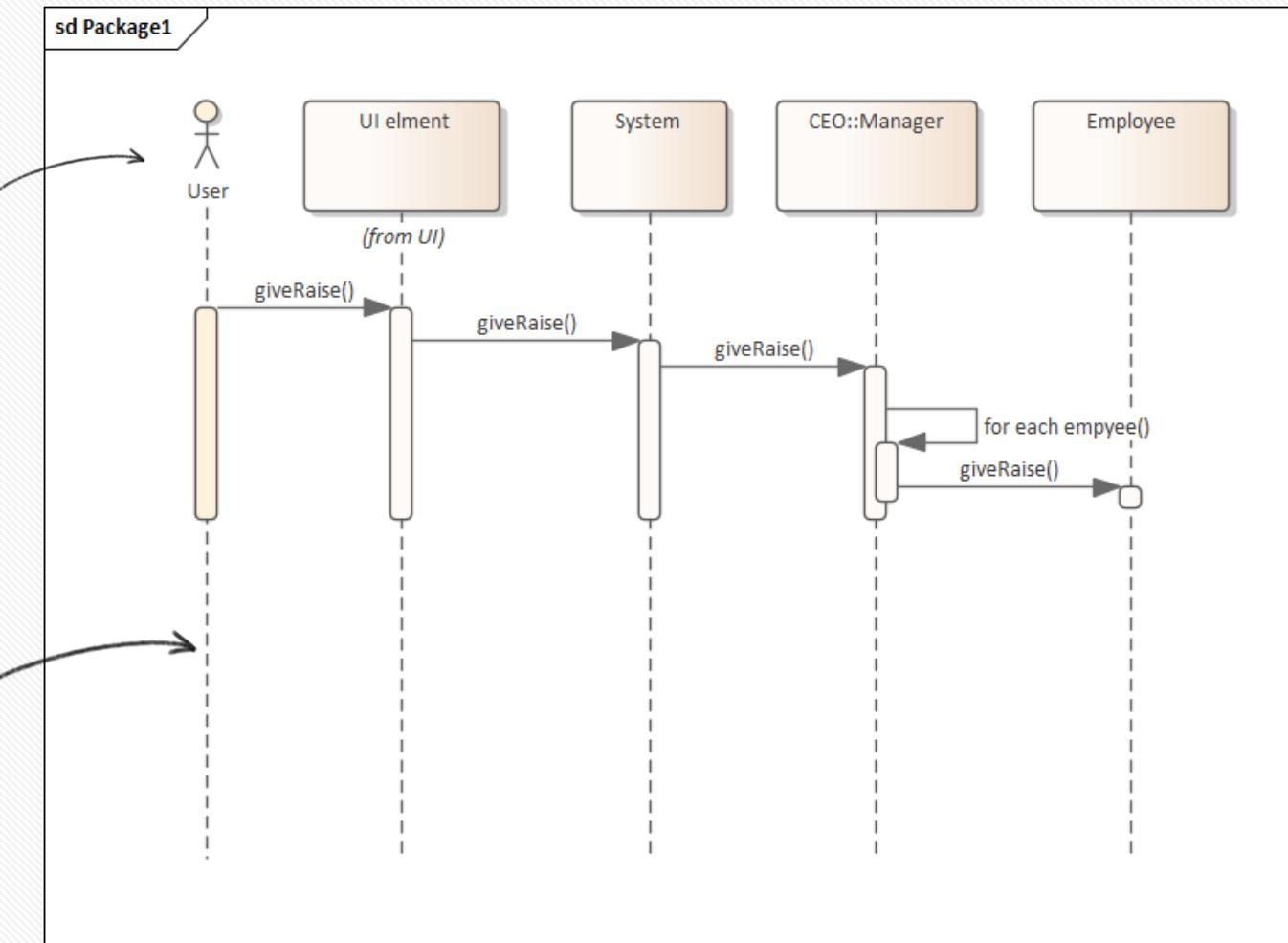
- ▶ Class diagram only describes the static state of the system. It is not enough.

# Sequence diagram

- ▶ Sequence diagram describes a dynamic flow in a system.

An **actor** - define a user or a different system or a device that the system interacts with

An object lifeline



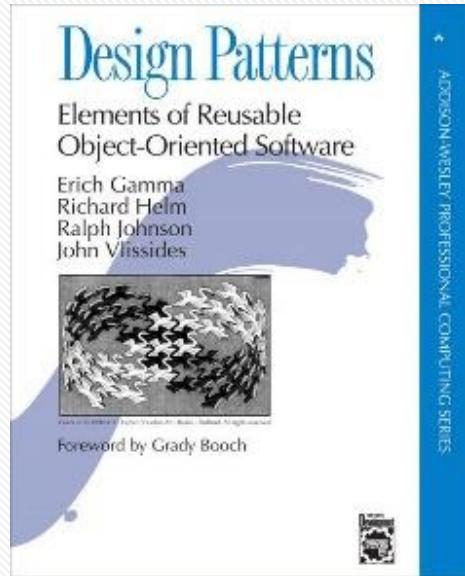
# Sequence diagram

- ▶ Describes **one** flow scenario in a system
- ▶ Usually, we describe the main scenarios using multiple sequence diagrams
- ▶ Useful to describe a bug in the system

# Design Patterns

- ▶ General **reusable** solutions to **common** problems in software design
- ▶ Not a finished solution
- ▶ With names to identify them
- ▶ They deal with
  - ◆ relationships between classes or other collaborators
  - ◆ Abstractions on top of code
- ▶ Patterns are not concerned with
  - ◆ Algorithms
  - ◆ Specific implementation or classes

# Design patterns history



In 1994, The famous book called “Gang of four” (or GoF) came out.

Current version:

Design patterns : Elements of Reusable Object Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides 2002

# Why are they important?

- ▶ Gives a **shared standard** design language
- ▶ Avoid reinventing the wheel
- ▶ Starting point for a solution => shorter time to production
- ▶ Improve system design

# Design Pattern Categories

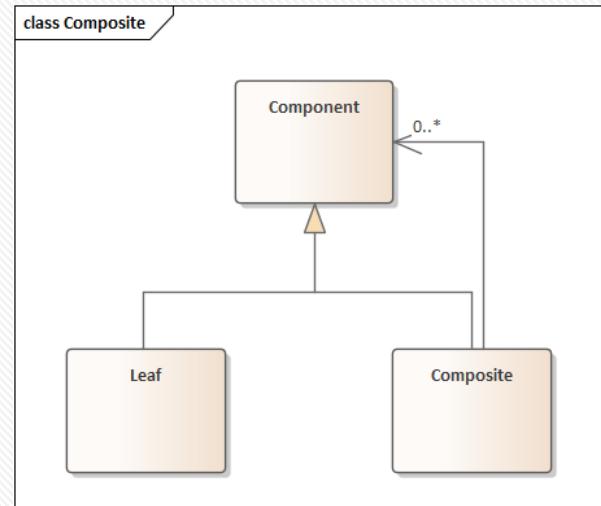
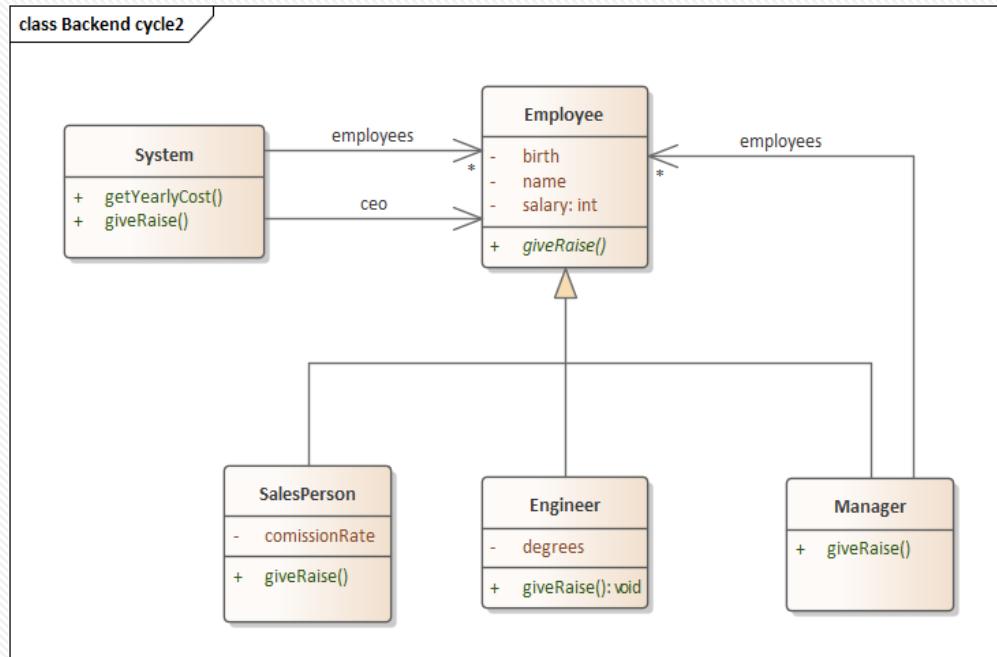
There are three basic categories of design patterns:

- ▶ **Behavioural**
  - ◆ Concerned with **communication** between objects
  - ◆ Examples taught in class: **Iterator, Strategy, Command**
- ▶ **Structural**
  - ◆ Concerned with class and object **composition**
  - ◆ Examples taught in class: **Composite, Adapter, Decorator**
- ▶ **Creational**
  - ◆ Concerned with **creating** and accessing objects
  - ◆ Examples taught in class: **Factory, Abstract Factory, Singleton**

Already seen!

# Composite

- ▶ Already seen, used to describe a hierarchy



The general case

# Example: Poker Game

We're building a **Poker Game** application; each player makes a move on their turn.

- ▶ A player can **call**, **fold**, or **raise**.

```
class Game {  
    Player *players;  
    Player *currentPlayer  
  
public:  
    void fold();  
    void call();  
    void raise(int amount);  
};
```

Works on current player

For **design simplicity**,  
ignore memory  
management for now.

```
class Player {  
public:  
    string getName();  
    int getMoney();  
  
    void makeMove(Game& game)  
};
```

Uses game and chooses whether  
to call, fold or raise.

# Strategy: Motivation

- ▶ Add a computer player with different behaviours:  
**Cautious, Aggressive, Bluffing**, etc.
- ▶ Each behaviour plays **makeMove** differently
- ▶ How should we design our **Player** class?

# Strategy: using enums?

What if we kept an **enum** for every behaviour?

```
enum Behaviour {  
    AGGRESSIVE, CAUTIOUS, BLUFFING  
};  
  
class Player {  
    Behaviour behaviour;  
public:  
    void makeMove(Game& game) {  
        switch(behaviour) {  
            case AGGRESSIVE: ...  
            case CAUTIOUS: ...  
            case BLUFFING: ...  
        }  
    }  
};
```

In each behaviour we have a different logic for calling game.fold(), game.call() or game.Raise()



# Strategy: using enums?

Problems with this solution:

- ▶ Adding new behaviours is hard and tiresome
  - ◆ It requires modifying the **Player** class
- ▶ A given behaviour can influence more than one method.
  - ◆ For example, other methods than **makeMove** are influenced by behaviour.
    - ◆ If we used enums, our implementation would be split up across multiple methods.
- ▶ We want to add a behaviour with as little change to the existing classes.

# Strategy: using inheritance?

We can make **Player** abstract (pure virtual)?

```
class Player {  
public:  
    virtual void makeMove(Game& game) = 0;  
};  
  
class AggressivePlayer : public Player {  
public:  
    void makeMove(Game& game) override;  
};
```

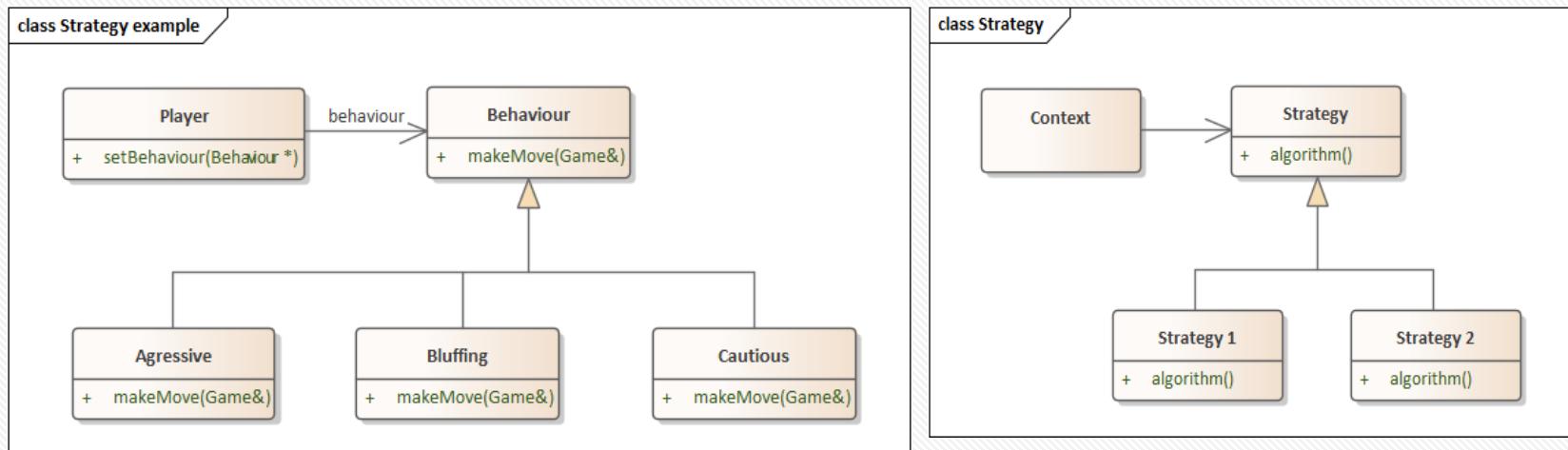
Problem?

- ▶ Changing a player's behaviour requires creating new object and copying the variables from the old object to the new

# Strategy: Solution

Instead of solving the problem inside **Player**, we decompose the problem

- ▶ Extract the behaviour to an external class
- ▶ Let **Player** set a behaviour



# Strategy: Solution

```
class Behaviour {  
    virtual void makeMove(Game& game) = 0;  
};  
  
class AggressiveBehaviour : public Behaviour {  
public:  
    void makeMove(Game game&) override;  
};  
  
class Player {  
    Behaviour *m_behaviour;  
  
public:  
    void setBehaviour(Behaviour *behaviour) { m_behaviour = behaviour; }  
    void makeMove(Game& game) { m_behaviour->makeMove(game); }  
};
```

# Command: Motivation

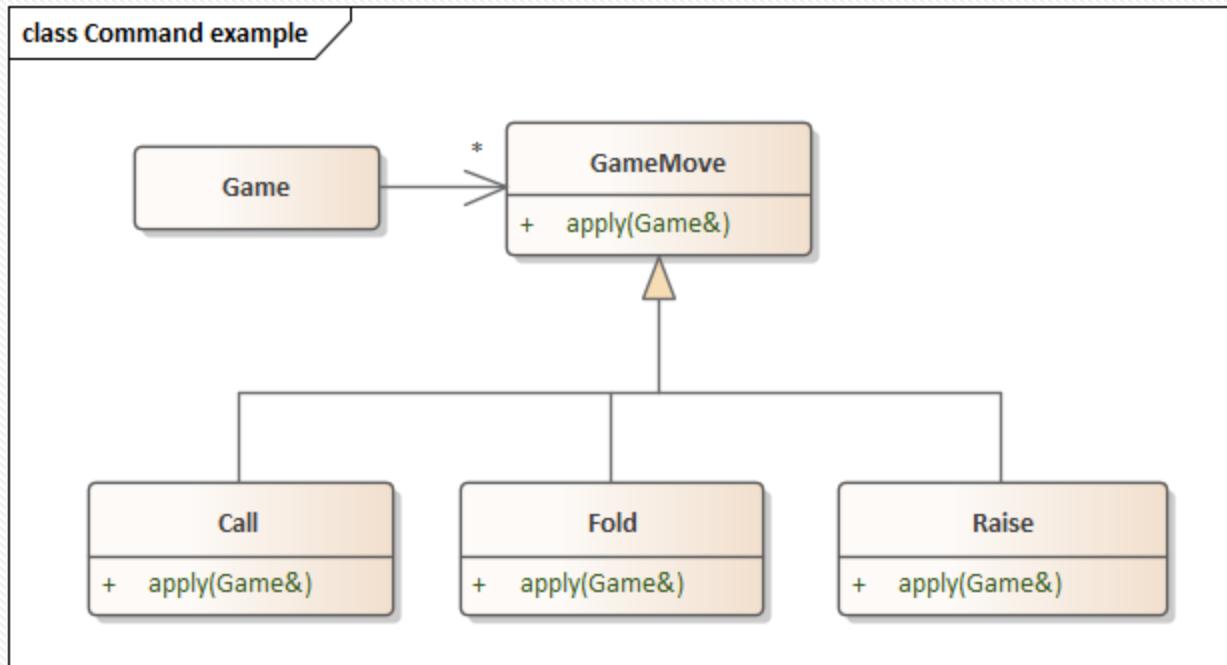
- ▶ How will we design the moves: call, fold, or raise?
- ▶ We also want to **keep track** of all moves
  - ◆ For example, to allow move **undo**

Problems:

- ▶ **Game** can already be a very **large** class; we don't want to make it bigger.
- ▶ We will want to add more game moves in the future, with a little change to the system.
- ▶ The logic around making a move is **identical**
  - ◆ Log the move made
  - ◆ Check if round finished
  - ◆ Move control to the next player if not

# Command: Solution

Solution: Separate **GameMove** and all its types into a separate class hierarchy.

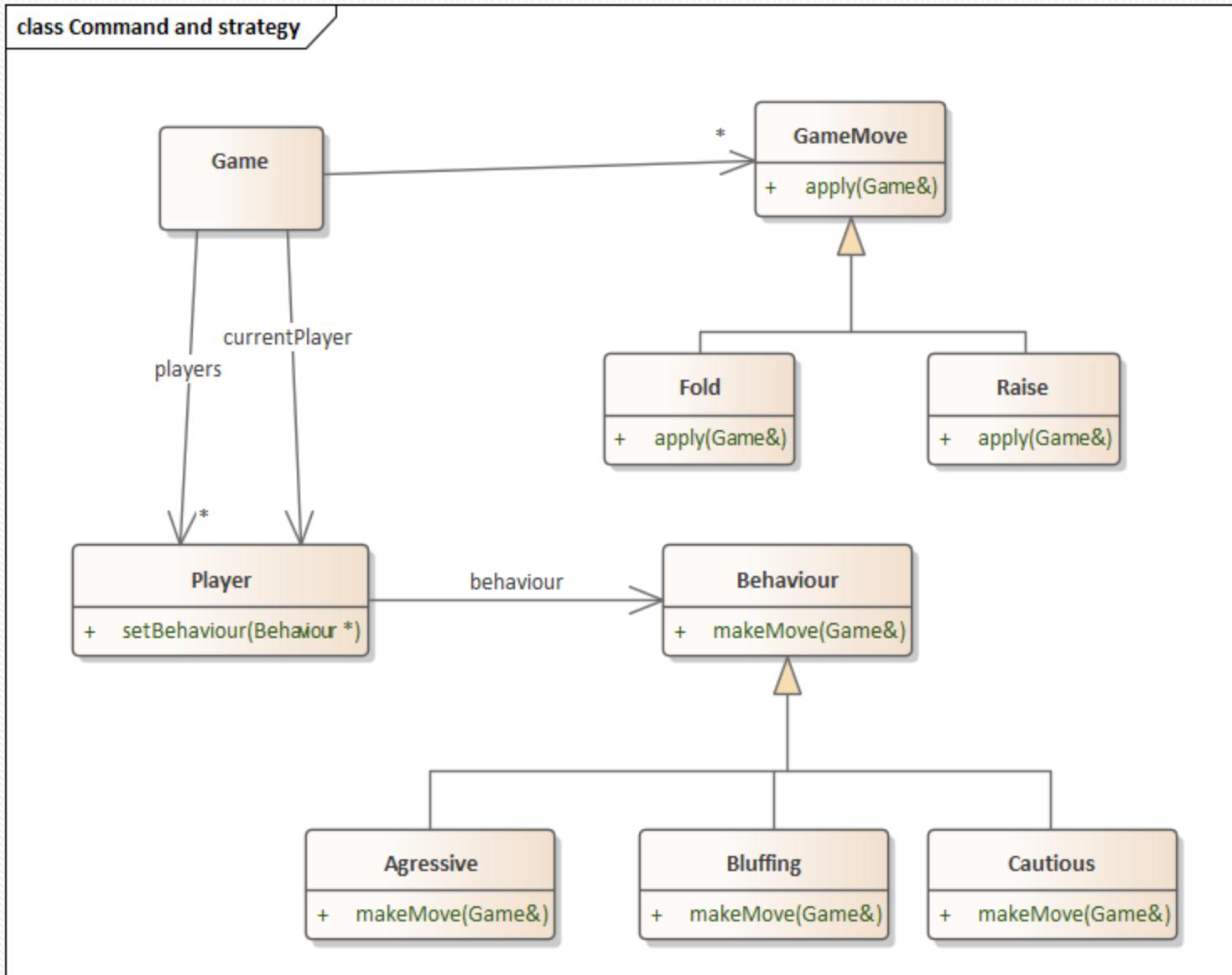


# Command: Solution

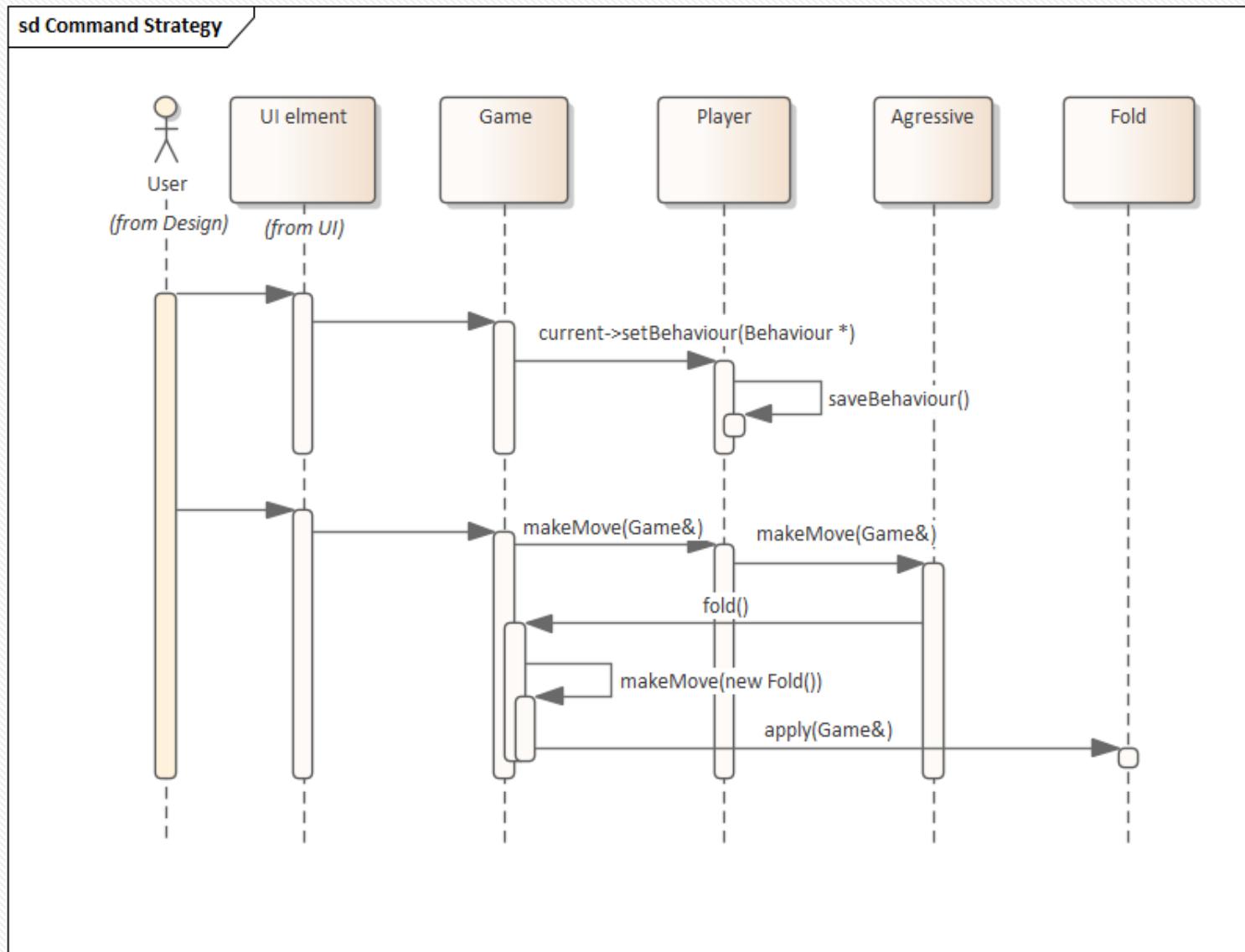
```
class Game {  
    vector<GameMove*> moves;  
public:  
    void makeMove(GameMove* move) {  
        cout << "Applied: " << move;  
        moves.push_back(move);  
        move->apply(*this);  
        nextPlayer();  
    }  
  
    void undo() {  
        int n = moves.size();  
        if (n > 0) {  
            prevPlayer();  
            moves[n - 1]->undo(*this);  
            moves.pop_back();  
        }  
    }  
};
```

```
class GameMove {  
public:  
    virtual void apply(Game&) const = 0;  
    virtual void undo(Game&) const = 0;  
};  
  
class Fold : public GameMove {  
public:  
    void apply(Game& game) const override;  
    void undo(Game&) const override;  
};  
  
class Call : public GameMove {  
public:  
    void apply(Game& game) const override;  
    void undo(Game&) const override;  
};  
  
class Raise : public GameMove {  
    int amount;  
public:  
    void apply(Game& game) const override;  
    void undo(Game&) const override;  
};
```

# Combining command and strategy



# Combining command and strategy



# Adapter: Motivation

- ▶ Let's create a Graphical User Interface (GUI) for our poker game. Luckily, someone already wrote a complete general card game GUI library.
- ▶ Unfortunately, and expectedly, our types don't match
  - ◆ The library defines its own classes for Card, Suit, Player, etc.
- ▶ How can we leverage the library?

```
namespace Poker {  
    class Hand;  
    class Player;  
    class Game;  
}
```

Our code

No conflicts!  
(Thank you, namespaces!)

Library code

```
namespace GuiLib {  
    typedef int Suite;  
    typedef int Rank;  
    struct Card;  
    class Cards;  
    class Agent;  
    class CardGame;  
  
    class GuiWindow;  
}
```

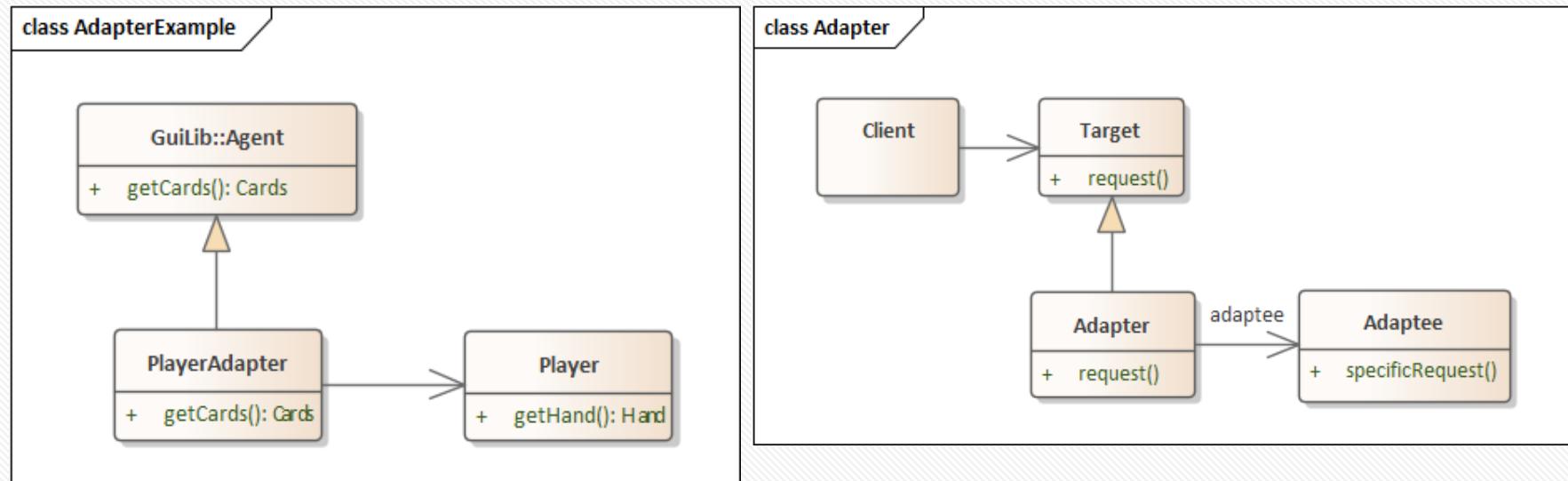
# Adapter: Motivation

What if we just replaced all our types with the library's types?

- ▶ Pros:
  - ◆ A single unified type is easier to use and reason about
- ▶ Cons:
  - ◆ A lot of work! Our types are already used all over our program
  - ◆ We give up control of our abstractions
    - ◆ Our types are probably better suited to our application than those of a generic library
  - ◆ What happens if we need to use another third-party library for AI playing card games?
    - ◆ We can't replace our types with *both* libraries!

# Adapter: Solution

Instead of replacing one type with another, we'll create special ad-hoc adapters



# Adapter : Solution

```
namespace Poker {
    class Player {
        public:
            virtual string getName() const = 0;
            virtual Hand getHand() const = 0;
    };
}
```

```
namespace GuiLib {
    class Agent {
        public:
            virtual string getName() const = 0;
            virtual Cards getCards() const = 0;
    };
}
```

```
namespace Poker {
    class PlayerAdapter : public GuiLib::Agent { Explicit namespaces
        Player* m_player;                                can improve readability
        public:
            PlayerAdapter(Player* player) : m_player(player) { }
            string getName() const override {
                return m_player->getName();
            }
            GuiLib::Cards getCards() const override {
                return convertHandToCards(m_player->getHand());
            }
    };
}
```

Plain old data conversion  
(not an adapter)

# Decorator: Motivation

We want to enhance our GUI:

- ▶ Example features:
  - ◆ Profile picture for every player
  - ◆ Animations when folding/raising/calling
  - ◆ Show additional statistics near every player
    - ◆ Total cash, number of wins, etc.
- ▶ Choose some features and compose them one after the other
  - ◆ Each time different features can be chosen.

How do we go about implementing these features?

# Decorator: Motivation

Why not just add these functions to our main GUI class?

- ▶ We don't want our classes to get too big
  - ◆ We don't want to complicate it with this new demands
- ▶ Picking and toggling features is hard to maintain

# Decorator: Solution (Inheritance)

We can use basic inheritance to wrap some method calls with additional logic:

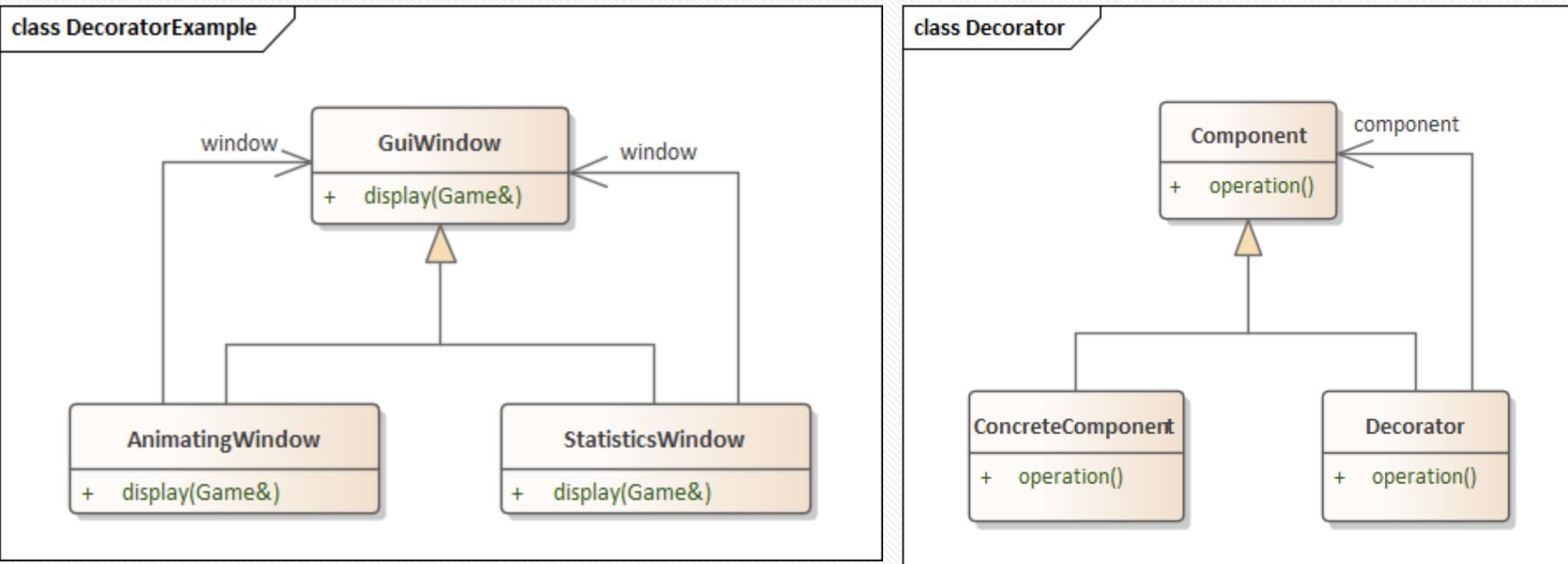
```
class GuiWindow {  
public:  
    virtual void display(CardGame& game);  
};  
  
class AnimatingWindow : public GuiWindow {  
public:  
    void display(CardGame& game) override {  
        GuiWindow::display(game);  
        animate(game);  
    }  
};  
  
class StatisticsWindow : public GuiWindow {  
public:  
    void display(CardGame& game) override {  
        GuiWindow::display(game);  
        addStatistics(game);  
    }  
};
```

Doesn't support  
picking and toggling  
multiple features

# Decorator: Solution (Composition)

- ▶ Again, we use a wrapper, and **prefer composition over inheritance**.
- ▶ This will work even if **GuiWindow** is itself **abstract**
- ▶ The downside - we have to add more manual **delegation code**.

# Decorator: Solution (Composition)

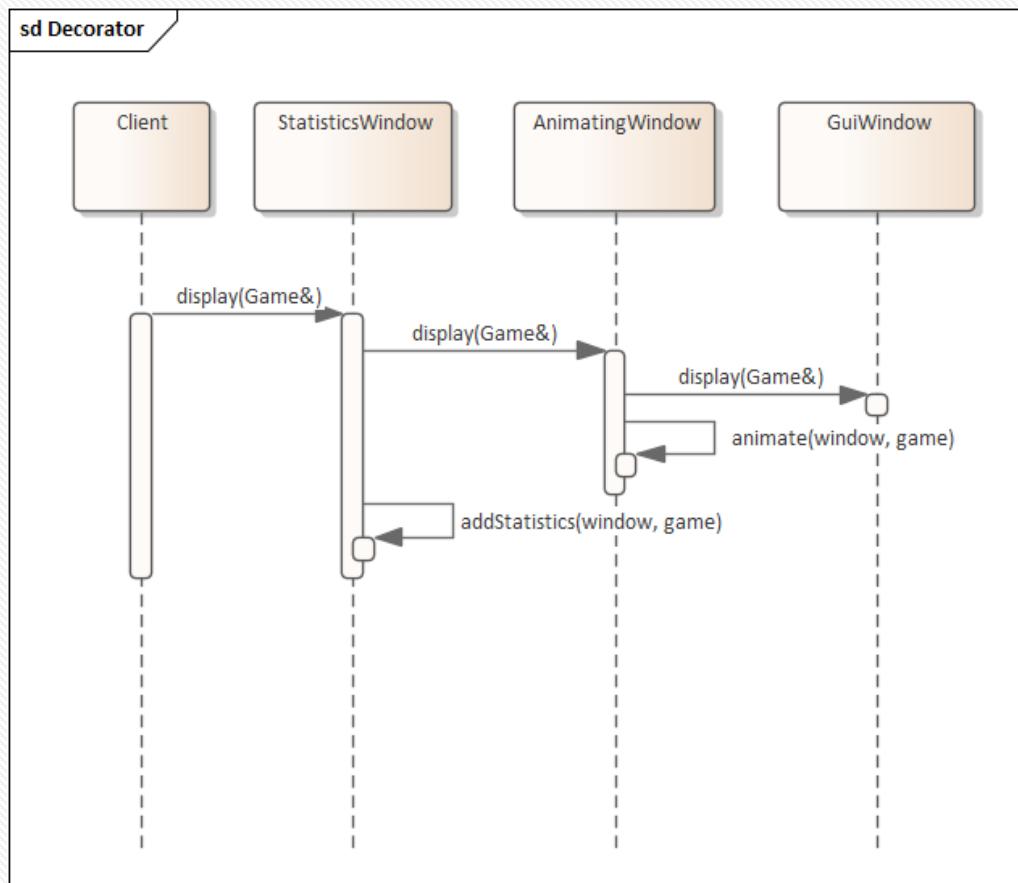


# Decorator: Solution (Composition)

```
class GuiWindow {  
public:  
    virtual void display(Game& game);  
};  
  
class AnimatingWindow : public GuiWindow {  
    GuiWindow* window;  
public:  
    void display(Game& game) override {  
        window->display(window, game);  
        animate(game);  
    }  
};  
  
class StatisticsWindow : public GuiWindow {  
    GuiWindow* window;  
public:  
    void display(Game& game) override {  
        window->display(window, game);  
        addStatistics(game);  
    }  
};
```

# Decorator: Solution (Composition)

```
GuiWindow* animatingWindowWithStatistics =  
    new StatisticsWindow(new AnimatingWindow(new GuiWindow()));  
  
animatingWindowWithStatistics->display(game);
```



# Factory: Motivation

We want to be able to take our game online

- ▶ When security is needed, we need to attach a special cryptographic token for every request
- ▶ Computing a crypto token is time consuming so we will not do that if it is not needed.
- ▶ We need a way to create **different messages in different circumstances**

# Factory: Motivation

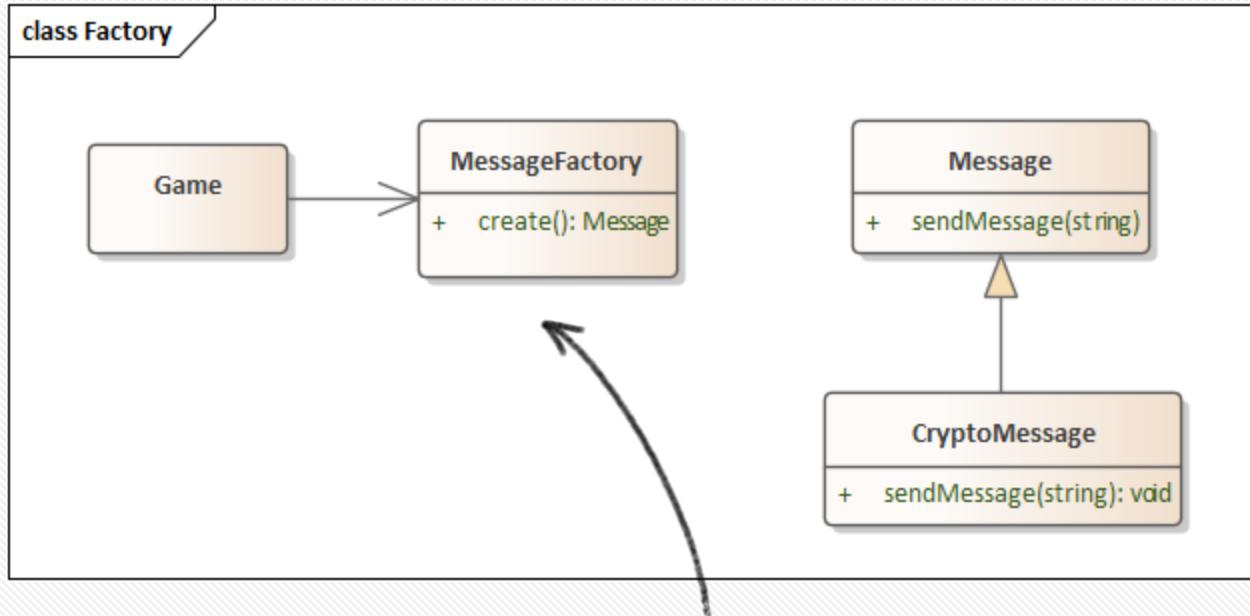
- ▶ A possible solution:

```
class Game {  
    bool m_isSecurityNeeded;  
public:  
  
    Game(bool isSecurityNeeded = false) {  
        m_isSecurityNeeded = isSecurityNeeded;  
    }  
  
    void sendMove(const GameMove& move) const {  
        Message *message;  
        if (!m_isSecurityNeeded)  
            message = new Message();  
        else  
            message = new CryptoMessage();  
  
        message->sendMessage(move.description());  
    }  
};
```

```
class Message {  
public:  
    virtual void sendMessage(const string &message);  
};  
  
class CryptoMessage : public Message {  
public:  
    void sendMessage(const string &message) override {  
        string encrypted = encrypt(message);  
        Message::sendMessage(encrypted);  
    }  
};
```

Acceptable solution, but sometimes we want to reduce game class size and want more flexibility

# Factory : Solution



We decompose the problem of  
the creation to a new class

# Factory : Solution

## ► A better solution:

```
class Game {
    MessageFactory *m_factory;
public:
    Game(MessageFactory *factory):
        m_factory(factory) {}

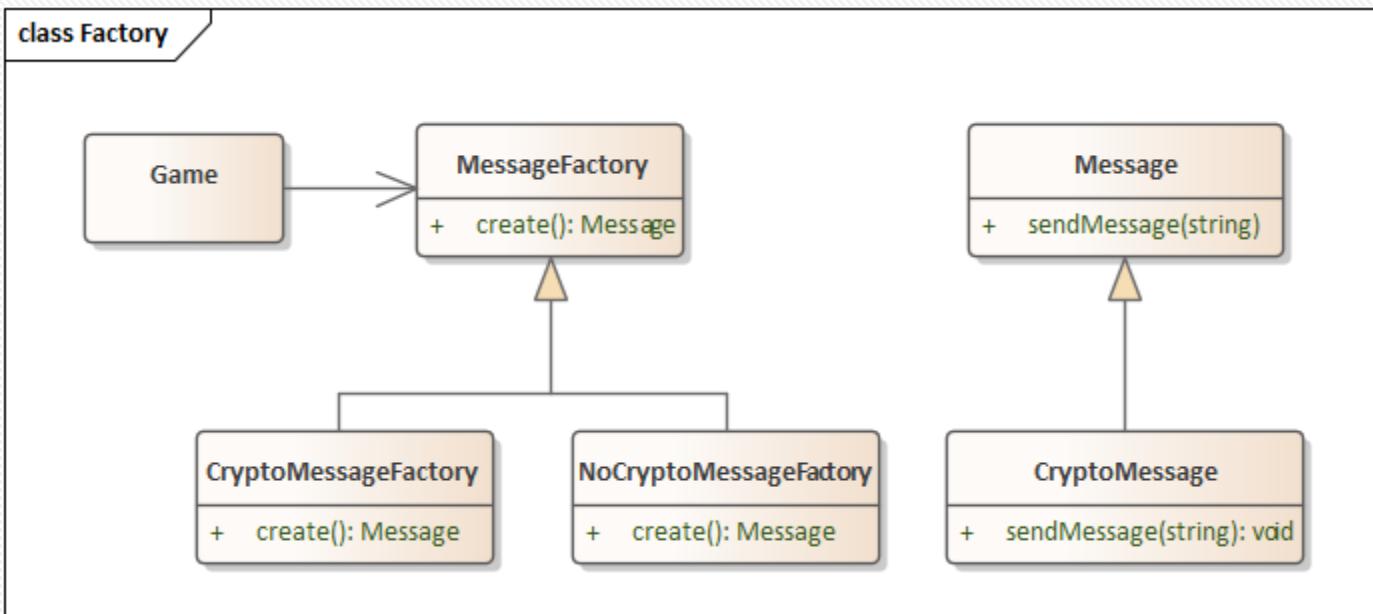
    void sendMove(const GameMove& move) const {
        Message *message = m_factory->create();
        message->sendMessage(move.description());
    }
};
```

```
class MessageFactory {
    bool m_isSecurityNeeded = false;
public:
    MessageFactory(bool isNeeded) {
        m_isSecurityNeeded = isNeeded;
    }

    Message *create() const {
        if (!m_isSecurityNeeded)
            message = new Message();
        else
            message = new CryptoMessage();
    }
}
```

What if we would like to use  
polymorphism instead ... ?

# Abstract Factory



# Abstract Factory

```
class Game {  
    MessageFactory *m_factory;  
public:  
    Game(MessageFactory *factory):  
        m_factory(factory) {}  
  
    void sendMove(const GameMove& move) const {  
        Message *message = m_factory->create();  
        message->sendMessage(move.description());  
    }  
};
```

```
..  
Game *game;  
if (isSecurityNeeded)  
    game = new game(new CryptoMessageFactory())  
else  
    game = new game(new NoCryptoMessageFactory())
```

```
class Factory {  
public:  
    virtual Message *create() const = 0;  
};  
  
class CryptoMessageFactory : public Factory {  
public:  
    CryptoMessage *create() const override {  
        return new CryptoMessage;  
    }  
};  
  
class NoCryptoMessageFactory : public Factory {  
public:  
    Message *create() const override {  
        return new Message;  
    }  
};
```

# **Singleton: Motivation**

- ▶ Sometimes we want a class that can make only one object.
- ▶ Usually used for a main class like System:
  - ◆ There is only one of them.
  - ◆ A lot of places in the system needs to get an easy access to it.

# **Singleton: Motivation**

In our example, we want to have only one MainWindow that every one can access.

- ▶ Prevent creating multiple main windows in our application
- ▶ Creating a MainWindow is very expensive
- ▶ Client can be confused and update the wrong window
- ▶ It's better to enforce this during compilation than at runtime, by disallowing multiple instances from being created in the first place

# Singleton: Solution

```
class MainWindow {  
    MainWindow() {}  
public:  
  
    static MainWindow& getInstance() {  
        static MainWindow instance;  
        return instance;  
    }  
  
    MainWindow(const MainWindow& other) = delete;  
    MainWindow& operator=(const MainWindow& other) = delete;  
  
    void display(Game& game);  
}
```

private constructor!

Guaranteed to be destroyed,  
Instantiated on first use

How to use:

```
MainWindow::getInstance().display(game);
```

# Singleton: Considerations

Although simple on the surface, the **singleton** design pattern has **far-reaching** consequences...

- ▶ A Singleton class cannot be **subclassed**
  - ◆ Nor should it be
  - ◆ Remember, if **A extends B** then **A is a B!**
- ▶ A **Singleton** is a **global object** accessible from **all** areas of the code → Inherit all global objects problems
- ▶ What happens if some time in the future, you find out you *do* need two windows (for example, tabs)?

# Singleton: Dependency Injection

```
class MainWindow {  
public:  
    virtual void display(Game& game) = 0;  
};
```

```
class MainWindowSingleton : public MainWindow {  
...  
};
```

Rest of code remains the same

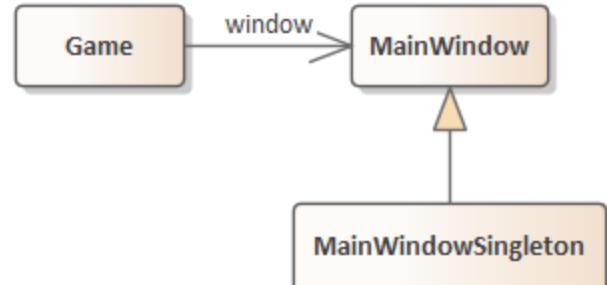
```
class Game {  
    MainWindow* window;  
public:  
    Game(MainWindow* window):  
        window(window) {}
```

```
    void start() {  
        window->display(*this);  
    }  
};
```

```
int main() {  
    Game game(MainWindowSingleton::getInstance());  
    game.start();  
}
```

Code is unaware  
of any globalness

class Singleton dependency injection example



Only the outer layer knows that it  
is actually a singleton

*“A design that doesn’t take change into account risks major redesign in the future”*

- Erich Gamma

# Software Testing

Testing Code and Creating Unit Tests

# Why Test Software?

- ▶ Need a way to be sure our program is giving us **the correct result**
  - ◆ It is impossible to reliably determine that a program is correct just **by trying it** out or **looking at its code**

although both are important!
- ▶ Software projects are developed by several programmers
  - ◆ Need a way to make sure a programmers' changes did not **degrade the performance of the full system**
- ▶ The end client will not be able to correct bugs on his own
  - ◆ It can take months from when a user detects a bug until the software is actually corrected
  - ◆ The cost of **a bug going into production** can be in the millions of \$

# Software Testing

- ▶ Fact: Software is becoming **increasingly complex**
  - ◆ Testing methods must be able to handle very large projects
- ▶ Investment in testing has dramatically increased over the years
  - ◆ Can reach **x1-x1.5 of the implementation effort**
- ▶ Some statistics for MS-Windows:

Date	Product	Dev. team	Test team	Lines of code
Jul-93	Win NT 3.1	200	140	4-5 million
Sep-94	Win NT 3.5	300	230	7-8 million
May-95	Win NT 3.51	450	325	9-10 million
Jul-96	Win NT 4.0	800	700	11-12 million
Dec-99	Win 2000 (NT 5.0)	1,400	<b>1,700</b>	29+ million
Oct-01	Win XP (NT 5.1)	1,800	<b>2,200</b>	40 million
Apr-03	Win Server 2003 (NT 5.2)	2,000	<b>2,400</b>	50 million

\**The Build Master*, Vincent Maraia (2005)

# Software Testing

## ▶ When should software be tested?

- ◆ When it's done
  - **Too late**, fixing bugs is cheaper if they are found **earlier**
  - When is it “done”?



Time detected	Cost to fix
Development	1x
System testing	10x
After release	10-25x

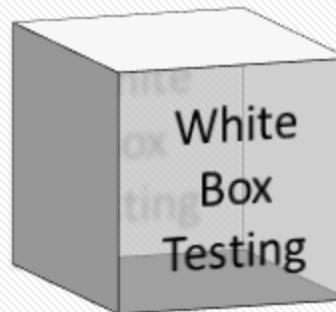
\*Code Complete (2nd ed.), Steve McConnell (2004)

- ◆ The code has to be **tested all the time**
  - **Saves time** when mistakes occur

# Testing Categories



vs.

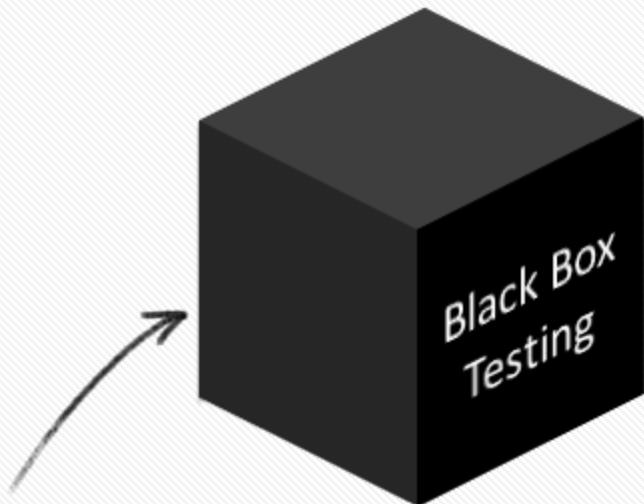


Static  
Testing

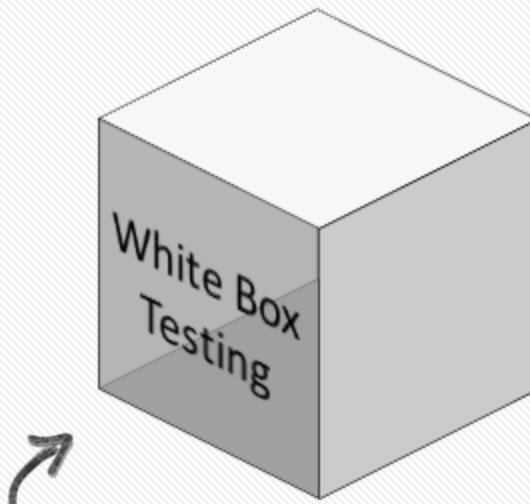
vs.

Dynamic  
Testing

# Black Box vs. White Box Testing



Testing without knowledge  
of the implementation



Testing while taking into  
account the  
implementation

When the two are mixed  
it's called gray-box testing

# Black Box (Behavioral) Testing

- ▶ Test **input-output relationships**

- ▶ **Advantages:**

- ◆ This is what the product is about
- ◆ Independent of implementation
- ◆ Easier to decide what to check

- ▶ **Disadvantages:**

- ◆ Identifying erroneous output may be hard
- ◆ Difficult to choose the right inputs to test
  - Most likely inputs?
  - Most problematic inputs?
- ◆ Hard to test a wide spectrum of system functionality



# White Box (Operational) Testing

- ▶ Test **how input becomes output**

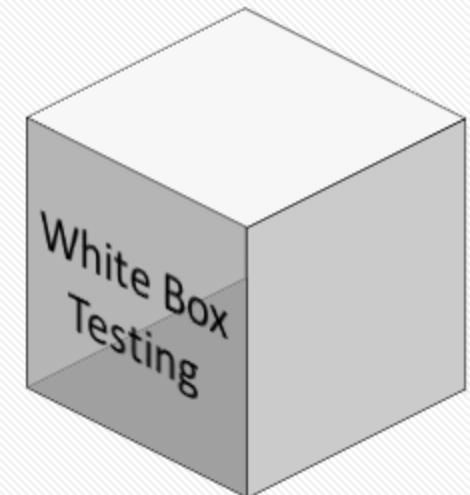
- ◆ Test the code itself

- ▶ **Advantages:**

- ◆ Easier to detect weaknesses of the program
  - ◆ The only way to check all execution paths and ensure good **code coverage**

- ▶ **Disadvantage:**

- ◆ Implementation dependent: more work to maintain
  - ◆ The product might have weaknesses which are hard to notice from within the code



# Simple Black Box Testing Example

- ▶ We'll create a simple black box test for a sorting function:

```
void sort(vector<int>& array);
```

sort.h

```
void printArray(const vector<int>& array) {
    for (int i : array) {
        cout << i << ", ";
    }
    cout << endl;
}
```

```
#define TEST_SIZE 100

void testSort() {
    vector<int> array(TEST_SIZE);
    cout << "Enter " << TEST_SIZE << " integers: ";
    for (int i = 0; i < TEST_SIZE; ++i) {
        cin >> array[i];
    }
    sort(array);
    printArray(array);
}
```

sort\_test.cpp

# Simple White Box Testing Example

- White box test code may (but does not have to) appear within the code itself:

```
void sort(vector<int>& array) {  
  
    assert(array.size() >= 0);  
  
    int num_swapped;  
    do {  
        num_swapped = 0;  
        for (int i = 0; i+1 < array.size(); ++i) {  
            if (array[i] > array[i+1]) {  
                swap(array[i], array[i+1]);  
                num_swapped++;  
            }  
        }  
        #ifdef TRACE  
        cout << "\nSwapped " << num_swapped << " numbers\n";  
        printArray(array);  
    #endif  
  
    } while (num_swapped > 0);  
}
```

sort.cpp

# Static vs. Dynamic Testing

## ▶ **Static** testing (white box only):

- ◆ Code reviews
  - Time-consuming, but **very effective**
- ◆ Using **code analysis** tools
  - Will look for suspicious code **without running it**  
(uninitialized objects, unused values, unreachable code, ...)
  - Limited capabilities, but constantly improving

## ▶ **Dynamic** testing

- ◆ Preprocessor-controlled code (white box)
  - Typically used to print or verify the status of internal objects
    - Example: assertions
- ◆ Writing **test drivers** (black or white box)
  - Code that runs the program (or parts of it) and checks its behavior

# Automatic Software Testing

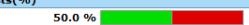
- ▶ How can we test our code?
- ▶ **Manually:** run the program or the test code and look for bugs
  - ◆ Simple to implement
  - ◆ Takes a lot of time = money
  - ◆ Easier to make mistakes
  - ◆ Things will have to be checked repeatedly
- ▶ **Automatically:** Write testing code that will automatically verify the correctness of the code
  - ◆ Requires writing more code
  - ◆ Once written, can be run many times

# Automatic Software Testing

- ▶ We prefer to invest in **automatic testing** when creating software
- ▶ If **good automatic tests are available**, we can know that the code is correct with the **touch of a button**
  - ◆ With some probability, tests can always miss some things
- ▶ Allows effective **regression testing** – making sure things don't break after a change
- ▶ Manual testing still has to be done
  - ◆ But we can minimize the time spent on it

# Automatic Software Testing

- ▶ Modern development environments can be configured to periodically (or whenever a change occurs) do **automatically** :
  - ◆ A full **build** of the project to make sure it compiles
  - ◆ A re-run of all (or part of) the **tests** to make sure no **regressions** have occurred
  - ◆ A generation of a detailed **test report**
- ▶ Not in this course... ☺

BUILD FAILED				
Project:	MySampleTests			
Date of build:	2012-08-23 09:04:53			
Running time:	00:01:11			
Integration Request:	Build (ForceBuild) triggered from GIBSON			
Modifications since last build (0)				
AutomationTest Report - gibson@GIBSON 2012-08-23 09:05:19				
Duration:				
StartTime:	23-08-2012 09:05:22			
EndTime:	23-08-2012 09:06:04			
Overall Result:	●			
Project	Passed Tests	Failed Tests	Passed tests(%)	
MySampleTests	1	1	50.0 %	 
Order	TestName	Duration	TestResult	
1	IpsumLoremDolor	00:00:14.6397736		
2	Google	00:00:27.4361008		
Test Name	Start Time	End Time	Duration	Result
IpsumLoremDolor	23-08-2012 09:05:22	23-08-2012 09:05:36	00:00:14.6397736	
Steps				
1	Navigate to : 'C:\Users\gibson\Documents\My Web Sites\Lorem ipsum dolor.html'			
2	[CheckFontSize] : Check Font Size			
3	[RetrieveTableText] : Retrieve table text			
4	[GetText] : GetParagraphText			
5	Verify 'ColorAndBackground:Color' style 'Exact' '#0000FF' on 'Text1Span'			
6	[IpsumLoremDolor_CodedStep] : @~"Verify 'ColorAndBackground:Color' style 'Exact' '#0000FF' on 'Text1Span'"			
Test Name	Start Time	End Time	Duration	Result
Google	23-08-2012 09:05:36	23-08-2012 09:06:04	00:00:27.4361008	
Steps				
1	Navigate to : 'http://www.google.com/'			
2	Enter text 'Telerik' in 'QueryInputField'			
3	Click 'GbqfsaSpan'			
4	Wait for 'TextContent' 'Contains' 'Telerik' on 'TelerikEmTag'			
Color	Meaning of color			
	Tests/Steps are passed			
	Tests/Steps are failed			
	Steps did not run			

# Automatic Testing of sort()

- ▶ We'll write tests that automatically determine whether they succeeded or failed

- ◆ In case of failure, we will supply the reason

test\_utils.h/.cpp

```
bool equalArrays(const vector<int>& arr1, const vector<int>& arr2) {  
    if (arr1.size() != arr2.size()) {  
        return false;  
    }  
    for (int i = 0; i < arr1.size(); ++i) {  
        if (arr1[i] != arr2[i])  
            return false;  
    }  
    return true;  
}
```

```
void printArrayDiff(const vector<int>& result, const vector<int>& expected) {  
    cout << "expected: ";  
    printArray(expected);  
    cout << "but received: ";  
    printArray(result);  
}
```

# Automatic Testing of sort()

- ▶ We'll write tests that automatically determine whether they succeeded or failed
  - ◆ In case of failure, we will supply the reason

```
void testSort() {  
    vector<int> array = { 3, 4, 2, 1, 5 };  
    vector<int> sorted = { 1, 2, 3, 4, 5 };  
    sort(array);  
  
    if (equalArrays(array, sorted)) {  
        cout << "Array sorted successfully!" << endl;  
    }  
    else {  
        cout << "Sort failed:" << endl;  
        printArrayDiff(array, sorted);  
    }  
}
```

sort\_test.cpp

```
int main() {  
    testSort();  
    return 0;  
}
```

# Automatic Testing

- ▶ Now that we can write tests, what **should** be tested?
  - ◆ **One input** is definitely not enough
  - ◆ We should **test boundary cases**:
    - What happens if the array size is **0** or **1**?
    - What happens if a number appears in the array **more than once**?
  - ◆ We should try **different** values for the function **arguments**
    - Generate random examples and compare to a different sorting routine which is already known to work

# sort\_test.cpp (1/4)

```
#include <iostream>
#include "test_utils.h"
#include "sort.h"

void testEqualArrays(const vector<int>& arr1, const vector<int>& arr2,
                     const char* testName) {
    if (equalArrays(arr1, arr2)) {
        cout << testName << ": PASSED" << endl;
    }
    else {
        cout << testName << ": FAILED" << endl;
        printArrayDiff(arr1, arr2);
    }
}
```

# sort\_test.cpp (2/4)

```
void testSort() {
    vector<int> array = { 3, 4, 2, 1, 5 };
    vector<int> sorted = { 1, 2, 3, 4, 5 };

    sort(array);
    testEqualArrays(array, sorted, "testSort");
}

void testSortWithDuplicates() {
    vector<int> array = { 3, 4, 3, 3, 4 };
    vector<int> sorted = { 3, 3, 3, 4, 4 };

    sort(array);
    testEqualArrays(array, sorted, "testSortWithDuplicates");
}
```

# sort\_test.cpp (3/4)

```
void testSortEmpty() {
    vector<int> array(0);
    vector<int> sorted(0);

    sort(array);
    testEqualArrays(array, sorted, "testSortEmpty");
}

void testSortSingleton() {
    vector<int> array = { 3 };
    vector<int> sorted = { 3 };

    sort(array);
    testEqualArrays(array, sorted, "testSortSingleton");
}
```

# sort\_test.cpp (4/4)

```
int main() {
    testSort();
    testSortWithDuplicates();
    testSortEmpty();
    testSortSingleton();

    return 0;
}
```

# Testing Large Software

- ▶ How can we test a larger system?
  - ◆ Must test the software at multiple levels

## High level

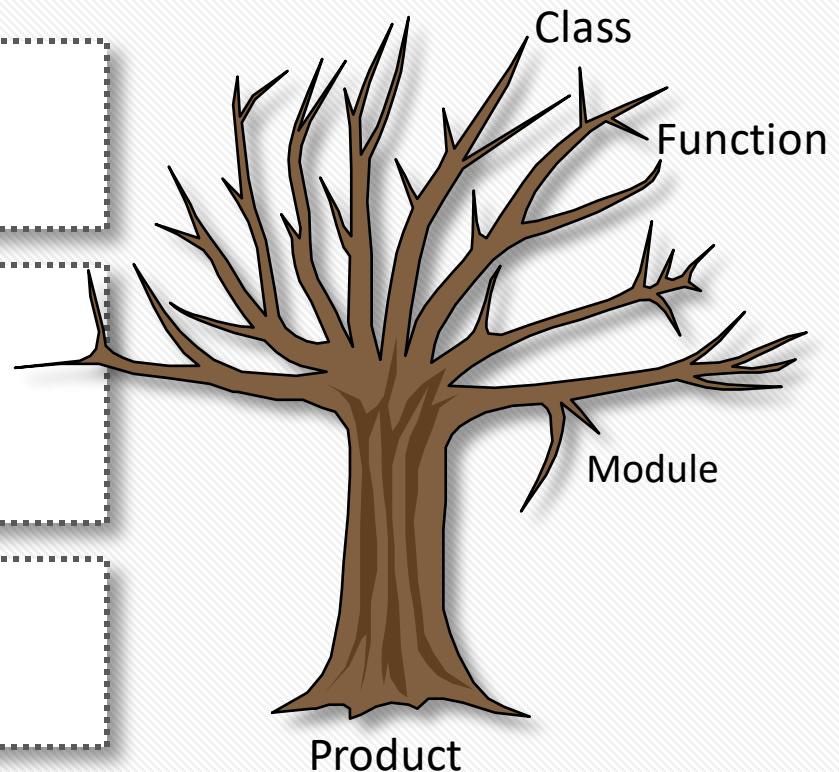
- System tests

## Intermediate level

- Sub-system tests
- Integration tests

## Low level

- Module and class tests



# Testing Large Software

## ▶ System tests

- ◆ Test the **overall system** in a specific usage scenario
- ◆ Writing automatic system tests is usually **very challenging**

## ▶ Integration tests

- ◆ Test the **interactions** between modules
- ◆ Particularly important when modules are developed by different people / teams

## ▶ Module (unit) tests

- ◆ Test a **single unit** in the code – typically a class
- ◆ Simple tests, and when they fail, it's **easy to locate the bug**
- ◆ They **do not test interactions** between modules or overall behavior

# Writing Unit Tests

- ▶ Test **one module** – in our case **a single class**
- ▶ Divide the unit test into **several tests**
  - ◆ A good rule of thumb:  
Write a test for every function in the interface
- ▶ In each test:
  1. Call the module with some input
    - ◆ Make sure to test error handling, not just normal behavior
  2. Check that the returned values are correct

# Example: A Unit Test for Set

set\_test.cpp

```
int main() {
    testSetConstruct();
    testSetCopy();
    testSetAssign();
    testSetAdd();
    testSetRemove();
    testSetContains();
    testSetGetSize();
    testSetUnite();
    testSetIntersect();
    testSetFilter();
    testSetIsEmpty();
    testSetOutput();
    testSetIterator();
    testSetIteratorEmptySet();
    testSetIteratorOneElement();
    return 0;
}
```

(we'll just look at  
a two of these  
here)

# Example: A Unit Test for Set

```
void testSetCopy() {  
    Set<int> empty;  
    Set<int> emptyCopy = empty;  
    testEqualSets(empty, emptyCopy, "testSetCopy");  
  
    Set<string> set;  
    set.add("joey");  
    set.add("chandler");  
    set.add("monica")  
    Set copy = set;  
    testEqualSets(set, copy, "testSetCopy");  
    set.add("ross");  
    testNotEqualSets(set, copy, "testSetCopy");  
}
```

set\_test.cpp

same idea as  
testEqualVectors

make sure copies are  
independent

# Example: A Unit Test for Set

```
void testSetGetSize() {  
  
    Set<string> empty;  
    Set<string> colors;  
    colors.add("orange");  
    colors.add("red");  
    colors.add("magenta");  
  
    verifyEqual(empty.getSize(), 0, "testSetGetSize");  
    verifyEqual(colors.getSize(), 3, "testSetGetSize");  
  
    Set<string> colorsCopy = colors;  
    colors.remove("orange");  
    verifyEqual(colors.getSize(), 2, "testSetGetSize");  
    verifyEqual(colorsCopy.getSize(), 3, "testSetGetSize");  
}
```

set\_test.cpp

# Writing Testable Code

- ▶ The real challenge is writing code that can be easily tested
  - ◆ Write modules with **little or no dependencies**
    - ◆ Isolate class members with **private** and **protected**
    - ◆ Do not use modifiable **global objects**
  - ◆ Write **short functions** that perform a single task
  - ◆ Minimize **side-effects** of functions
  - ◆ Reduce **code duplication** through helper functions and class inheritance

*"I find that writing unit tests actually increases my programming speed"*

- Martin Fowler, in "UML Distilled"

# More C++ Features

A collection of useful C++ features

# Contents

- **Automatic type inference**
- Constructor inheritance and delegation
- Lambda expressions
- In short: a few smaller features

# Automatic type inference

- Consider these typical uses of STL:

```
std::vector<int>::iterator iter =  
    std::find(ids.begin(), ids.end(), 315085121);
```

```
std::pair<std::string, std::string> firstAndLastNames =  
    std::minmax(names.begin(), names.end());
```

```
std::chrono::time_point<std::chrono::system_clock> time =  
    std::chrono::system_clock::now();
```

- Those type names can get complicated quickly...

# Automatic type inference

- Consider these typical uses of STL:

```
std::vector<int>::iterator iter =  
    std::find(ids.begin(), ids.end(), 315085121);
```

```
std::pair<std::string, std::string> firstAndLastNames =  
    std::minmax(names.begin(), names.end());
```

```
std::chrono::time_point<std::chrono::system_clock> time =  
    std::chrono::system_clock::now();
```



The compiler **knows** the function return types, why not just have it deduce them for us?

# Automatic type inference

- Consider these typical uses of STL:

```
auto iter =  
    std::find(ids.begin(), ids.end(), 315085121);
```

```
auto firstAndLastNames =  
    std::minmax(names.begin(), names.end());
```

```
auto time =  
    std::chrono::system_clock::now();
```



The compiler **knows** the function return types, why not just have it deduce them for us?

# Automatic type inference

- The **auto** keyword declares a variable to the type of its initialization:

```
auto iter = std::find(ids.begin(), ids.end(), 315085121);
auto firstAndLastNames = std::minmax(names.begin(), names.end());
auto time = std::chrono::system_clock::now();
```



Hover over the variable  
to see its deduced type

Visual Studio

```
auto time = std::chrono::system_clock::now();
std::chrono::time_point<std::chrono::system_clock> time
```

- Const and reference modifiers** can be added to refine the variable type:

```
vector<string> names;
vector<int> ids;
auto& namesRef = names;      // namesRef is a vector<string>&
const auto& idsRef = ids;    // idsRef is a const vector<int>&
```

# When **not** to use auto

- Do **not** use auto when we actually **care about the type** of the variable



```
auto height = person.getHeight();
auto center = height / 2;
scanf("%lf", &height);
```



Does this function  
return an **int**? A  
**double**??  
Potential bugs



```
double height = person.getHeight();
double center = height / 2;
scanf("%lf", &height);
```

- Do **not** use auto when it **harms readability**



```
auto title = table.data.elements[i].title.text;
```



```
FormattedText* title = ... // ok, got it
```



To follow this, we need  
to read a long nested  
line

# When to **use** auto

- Use **auto** when the actual type is not important, only **what it can do**:

```
std::map<string, string> addressBook;

for (auto it = addressBook.begin();
     it != addressBook.end();
     ++it)
{
    cout << "Name = " << it->first <<
          "Address = " << it->second << endl;
}
```

it has operators  
++, !=, and ->. We  
do not care about  
its actual type.

std::map<string, string>::iterator

# When to **use** **auto**

- Use **auto** when the **type is clear from the context**, and writing it just adds clutter:

```
Matrix<double> A, B;  
auto C = inv(A)*B;
```

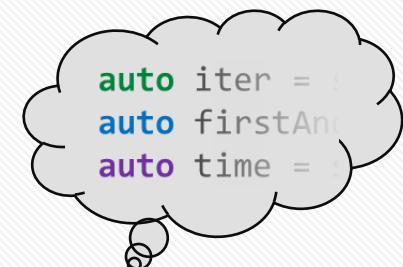


Also a matrix

- Use **auto** to avoid redundant type specification:

```
auto button = new UIButton("OK", COLOR_GRAY);  
auto imageData = (uint8_t*)image.data();  
auto I = Matrix<float>::identity(10,10);
```

Static method



All the examples we saw earlier are also fine

# When to **use** auto

- In some cases, using **auto** may be the **only choice**:

```
template <class Sequence>
void printMax(const Sequence& seq, int n)
{
    auto maxval = seq[0];
    for (int i=0; i<n; ++i)
    {
        maxval = seq[i] > maxval ? seq[i] : maxval;
    }
    cout << maxval;
}
```



Defines `maxval` to the type of the sequence

```
template <class T, class U>
void multiply(T x, U y)
{
    auto res = x*y;
}
```



Defines `res` to be the larger type of `T` and `U`

# Notes about auto

- Give special care to **variable names** when using auto

 `auto x = table.queryCell(3,5,&cell);` ↗ What is x??

 `auto status = table.queryCell(3,5,&cell);` ↗ With **good variable names**, most code can use **auto**

- Don't forget to use **reference modifiers** where needed

```
class DrawingCanvas {  
public:  
    vector<Shape*>& GetShapes();  
    Shape& SelectedShape();  
};  
  
DrawingCanvas canvas;
```

`auto& shapes = canvas.GetShapes();`

Without &, causes a **copy** of the vector

`auto& currShape = canvas.SelectedShape();`

Without &, **will not compile** (Shape is abstract)

# Return type deduction

- Back to our last examples. What if we wanted to **return** maxval or res?

```
template <class T, class U>
auto multiply(T x, U y)
{
    auto res = x*y;
    return res;
}
```

```
void main()
{
    cout << multiply(5,4);
}
```

Determines the return type  
from the **return** statement

- There is also automatic deduction for **function return types**
  - Requires that all **return** statements return the *same type*
  - Requires that the **entire function code be known** when it is called
    - So mostly applicable to template functions

# Contents

- Automatic type inference
- **Constructor inheritance and delegation**
- Lambda expressions
- In short: a few smaller C++ features

# Advanced constructor features

- Let's consider the following **Webcam** class:

```
class Webcam
{
public:
    Webcam(int width, int height, double exposure = AUTO_EXPOSURE)
    {
        Webcam::Profile prof(width, height);
        if (exposure == AUTO_EXPOSURE) {
            prof.setAutoExposure();
        }
        else {
            prof.setExposure(exposure);
        }
        m_camHandle.init(prof);
        if (!m_camHandle.initialized())
            throw ConnectionError();
        m_buffer.resize(width*height);
        m_camHandle.setImageBuf(&m_buffer[0]);
        // ... and so on ...
    }
}
```

# Advanced constructor features

- Now, suppose that in version 2 we decide to add this:

```
class Webcam
{
public:
    // set height automatically from width, everything else is the same
    Webcam(int width, double exposure = AUTO_EXPOSURE);
}
```

- How do we implement the new constructor?
  - Option 1: **duplicate the code**
  - Option 2: **create a common init() function**
  - Option 3: **use constructor delegation**

# Constructor delegation

- One constructor can call another before it begins its own code:

```
class Webcam
{
public:
    // set height automatically from width, everything else is the same
    Webcam(int width, double exposure = AUTO_EXPOSURE) :
        Webcam(width, getHeight(width), exposure)
    {}
}
```

- The object is considered **fully constructed** after the first constructor is done. So, if the second constructor throws an exception, **the destructor is called!**

# Advanced constructor features

- Now, let's consider this class that **inherits from Webcam**:

```
class SmartphoneCam : public Webcam
{
public:

    SmartphoneCam(int width, int height, double exposure = AUTO_EXPOSURE)
        : Webcam(width, height, exposure) {}

    SmartphoneCam(int width, double exposure = AUTO_EXPOSURE)
        : Webcam(width, exposure) {}

    // And so on for all the constructors...
}
```



That's a lot of code! Also, what if we **add constructors** to Webcam in the future?

# Constructor inheritance

- We can tell the compiler to **inherit all base class constructors**:

```
class SmartphoneCam : public Webcam
{
public:
    using Webcam::Webcam;    // inherit Webcam's constructors
}
```

- An inherited constructor has the **same parameters** (including defaults) and performs the **same action** as in the base class.
- Any new members in the derived class are **default-initialized**

Too bad we can't change  
that... Or can we?



# In-class member initialization

- We can provide **default values** for any non-static class member:

```
class SmartphoneCam : public Webcam
{
    bool m_isUserFacing = true;
    FlashMode m_flashMode = FlashMode::Off;

public:
    using Webcam::Webcam; // inherit Webcam's constructors
}
```

- The default value is automatically assigned to the class member when it is created (unless the constructor **explicitly specifies a different value** in its initialization list)

# Contents

- Automatic type inference
- Constructor inheritance and delegation
- **Lambda expressions**
- In short: a few smaller C++ features

# Passing operations to functions

- Functions sometimes require **an operation** as their parameter
  - In the past, this used to be done using a **template**

```
template <class Iter, class Condition>
int std::count_if(Iter begin, Iter end, Condition cond)
{
    int count = 0;
    for (Iter it = begin; it != end; ++it)
    {
        if (cond(*it)) ++count;
    }
    return count;
}
```



cond is some object with an  
operator()

# Passing operations to functions

- Our template will accept both **standard functions** and **function objects** as the operation:

function

```
template <class T>
bool isPositive(const T& val)
{
    return val > 0;
}
```

```
int numPositive = std::count_if(
    v.begin(), v.end(),
    isPositive<int>
);
```

function object

```
template <class T> class GreaterThan
{
    const T thresh;
public:
    GreaterThan(const T& t) : thresh(t) {}
    bool operator()(const T& val) {
        return val > thresh;
    }
}
```

```
int numPositive = std::count_if(
    v.begin(), v.end(), GreaterThan<int>(0)
);
```

# Passing operations to functions

- However, in both cases we have to write **tedious additional code** in order to call `count_if()`
  - Possibly in a completely different place in the program
- **Lambda expressions** simply let us write:

## Lambda

```
int numPositive = std::count_if(  
    v.begin(), v.end(), [] (int val) { return val > 0; }  
);
```



No additional code!

# Lambda expressions

- The basic syntax for a lambda expression is:

```
[ ] (<parameters>) -> <return-type> { <code> }
```

- This produces a **temporary nameless function** equivalent to:

```
<return-type> unknown_name(<parameters>)
{
    <code>
}
```

- The lambda has a simple pointer-to-function type

```
<return-type> (*fptr)(<parameters>)
```

# Using a lambda expression

- A lambda can be assigned to a variable

```
auto gaussian = [] (double x) -> double {
    return 1.0/sqrt(2.0*M_PI)*exp(-x*x/2.0);
}
...
gaussian();
```

gaussian has type  
double(\*)(double), but  
easier to just use **auto**

- However, its main use is as a **function argument**

```
std::vector<int> v;
// ...
int numPositive = std::count_if(
    v.begin(), v.end(), [] (int n) -> bool { return n > 0; }
);
```

# Return value deduction

- A lambda that contains **only a return statement** can automatically deduce its return type

```
int numPositive = std::count_if(  
    v.begin(), v.end(), [] (int n) -> bool { return n > 0; }  
);
```

Redundant!

```
int numPositive = std::count_if(  
    v.begin(), v.end(), [] (int n) { return n > 0; }  
);
```

# Capturing variables in lambdas

- What about this case?

```
int thresh;  
cin >> thresh;  
  
int aboveThresh = std::count_if(  
    v.begin(), v.end(), [] (int n) { return n > thresh; }  
);
```

compilation error, thresh is **not**  
**in scope** within the lambda code

Just like a  
normal function

- But, unlike a normal function, a lambda expression can **capture local variables** from the enclosing scope

# Capturing variables in lambdas

- The [ ] in the lambda expression are actually there to **capture variables** from the enclosing scope

```
[<capture-list>] (<parameters>) -> <return-type> { <code> }
```

- You may capture a local variable by value

```
int numPositive = std::count_if(  
    v.begin(), v.end(), [thresh] (int n) { return n > thresh; }  
);
```

- Or by reference

This lambda can **modify** sum

```
int sum = 0;  
std::for_each(v.begin(), v.end(), [&sum] (int n) { sum+=n; });
```

# Capturing variables in lambdas

- You can capture multiple local variables

```
double a, b, c;  
// ...  
double result = integrate(0.0, 1.0,  
    [a,b,&c] (double x) { return a*x*x + b*x + c; }  
);
```

Just for the example

- Or you can **have the compiler** identify which variables are accessed, and let it capture them **automatically**

```
double result = integrate(0.0, 1.0,  
    [=] (double x) { return a*x*x + b*x + c; }  
);
```

[=] means auto-capture by value,  
[&] means auto-capture by reference

# Capturing variables in lambdas

- Finally, if you're within a class member function, you can **access the class members** by capturing this

```
class Polynomial
{
    double a, b, c;
public:
    double integrate(double xmin, double xmax) const
    {
        return numerics::integrate(xmin, xmax,
            [this] (double x) { return a*x*x + b*x + c; })
    }
};
```

[=] and [&]  
also capture  
this, plus  
local variables

- The member variables themselves are always captured **by reference** (or const reference if the method is const), as they are accessed through **this**

# Lambdas: Final remarks (1/2)

- Use lambdas for **simple and short operations**, which do not justify writing a full function
  - Remember that a lambda function is **not given a name**, so the code may become more difficult to interpret
  - Also, overuse of lambdas can lead to **code duplication**
- Used correctly, generic lambdas can **improve** readability

```
std::vector<std::shared_ptr<Shape>> v;  
  
int numRedShapes = std::count_if(  
    v.begin(), v.end(),  
    [] (shared_ptr<Shape> s) { return s->FillColor() == Red; }  
);
```

# Lambdas: Final remarks (2/2)

- When capturing by reference, **the captured object must still exist** when the lambda code is executed
  - A lambda does **not** extend the lifetime of captured references

```
void downloadUserPhoto(const char* userName, const char* outFileName)
{
    std::ofstream outFile(outFileName);

    auto query = server.createQuery("Name", userName, "Object", "Picture");
    server.processQueryAsync(query, [&] (const char* data, size_t n)
    {
        outFile.write(data, n); ←
    });
}
```

X by the time this code executes,  
outFile will be long gone!

# Contents

- Automatic type inference
- Constructor inheritance and delegation
- Lambda expressions
- **In short: a few smaller features**

# In short (1/3): Template aliases

- The following syntax which can be used as an **alternative to `typedef`**:

Equivalent to:

```
using real      = double;
using condfunc = bool(*)(int);
using point3d  = float[3];
```

```
typedef double real;
typedef bool(*condfunc)(int);
typedef float point3d[3];
```



this syntax makes the defined names more obvious

- These type definitions can be **templated**:

```
template <class T>
using compare_fcn = int(*)(T,T);
```

```
int(*)(int,int)
```

```
void sort(int a[], int n,
          compare_fcn<int> int_cmp);
```

Usage

# In short (2/3): Raw string literals

- C and C++ string literals are **very unfriendly** to path names:

```
const char* filename =  
    "c:\\users\\ron\\docs\\presentations\\cpp11\\cpp11.pptx";
```

- Raw string literals** treat all the characters within them **as-is**:

```
const char* filename =  
    R"(c:\\users\\ron\\docs\\presentations\\cpp11\\cpp11.pptx);
```



R" begins the raw string

)" ends  
it



```
std::string quote =  
    R"(\"Life is like a box of chocolates.\" -Forrest Gump's Mom)";
```

# In short (3/3): New integer types

- There is a **long long int** type
  - In the best tradition of C/C++, its exact size is undefined
  - But it should be **at least 64 bits** (and in reality is exactly that)
- The **<cstdint>** header file provides definitions for exact-width integral types

<code>int8_t</code>	<code>int16_t</code>	<code>int32_t</code>	<code>int64_t</code>
<code>uint8_t</code>	<code>uint16_t</code>	<code>uint32_t</code>	<code>uint64_t</code>

- Prefer using these over **short/int/long/long long** when you need a **specific sized integer**

# Online resources

- **Wikipedia**

[http://www.wikipedia.org/wiki/c++11](http://www.wikipedia.org/wiki/c%2B%2B11)

[http://www.wikipedia.org/wiki/c++14](http://www.wikipedia.org/wiki/c%2B%2B14)

- **Stroustrup's C++11 FAQ**

[http://www.stroustrup.com/C++11FAQ.html](http://www.stroustrup.com/C%2B%2B11FAQ.html)

*Note: not always updated with the final standard*

- **The C++ reference**

<http://www.cppreference.com>

- **cplusplus.com**

<http://www.cplusplus.com>

“C++ has indeed become too “expert friendly” at a time where the degree of effective formal education of the average software developer has declined. However, the solution is not to dumb down the programming languages but to use a variety of programming languages and educate more experts. There have to be languages for those experts to use — and C++ is one of those languages.”

- Bjarne Stroustrup

# Move Semantics

Optimizing moves and  
implementing unique\_ptr

# Why move semantics?

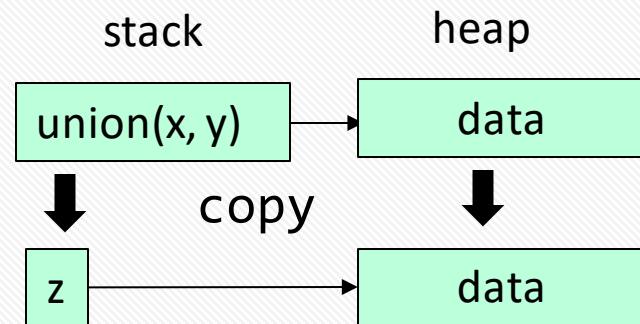
- ▶ Move semantics was introduced to solve two main problems:
  - ◆ Unnecessary copies
  - ◆ Ownership transfer, e.g. `unique_ptr` and file handles

# Unnecessary copies

- ▶ What happens when we run the following code?

```
Set<int> x, y;  
// code that populates x, y  
Set<int> z = Set<int>::union(x, y);
```

- ◆ union creates a new temporary set
- ◆ Copy constructor:
  - Allocates memory for z
  - Copies temporary set to z
- ◆ Destructor frees temporary set



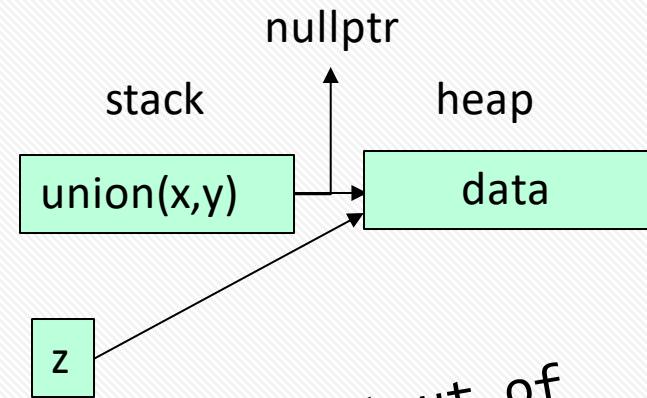
Unnecessary copy is inefficient!

# Unnecessary copies

## ▶ Dream solution:

- ◆ Copy constructor that “knows” whether its input is a temporary copy or not
- ◆ If the input is temporary, set is **moved** rather than **copied**

But how do we know that?



Since the output of the union is tmp, we can move it and “steal the pointer”

```
Set<int> x, y;  
// code that populates x, y  
Set<int> z = Set<int>::union(x, y); // move  
Set<int> w = x; // copy
```

# Value categories

- ▶ C++ distinguishes between:

- ◆ **lvalues** – values that can be assigned to
- ◆ **rvalues** – values that cannot be assigned to

```
int getValue() {  
    return 10;  
}  
  
int& getValue1() {  
    static int value = 10  
    return value;  
}
```

Diagram illustrating value categories:

- lvalues** (blue text):
  - int i = 10; → *lvalue* (blue arrow)
  - int a = i; → *lvalue* (blue arrow)
  - int i = getValue(); → *lvalue* (blue arrow)
  - getValue1() = 5; → *lvalue* (blue arrow)
- rvalues** (red text):
  - 10 = i; // error! → *rvalue* (red arrow)
  - getValue() = 5; // error! → *rvalue* (red arrow)
- returns rvalue** (red text):
  - int i = getValue(); → *returns rvalue* (red arrow)

# Value categories

## lvalues

- ◆ global variables

```
std::cout
```

- ◆ local variables

```
int i;  
Set<int> x;
```

- ◆ function parameters

```
f(x);
```

## rvalues

- ◆ literals

```
10
```

```
"C++"
```

- ◆ return values of functions and operators

```
height * width
```

```
Set<int>::union(x, y)
```

# Rvalue references

- ▶ References: bind only to lvalues

```
int i;  
int& ref1 = i;      // OK  
int& ref2 = 3;      // Fails: literal is rvalue  
int& ref3 = i + 3; // Fails: function output is rvalue
```

- ▶ Rvalue references: bind to both lvalues & rvalues

```
int i;           rvalue reference, not a reference to a reference!  
int&& ref1 = i;      // OK  
int&& ref2 = 3;      // OK: can bind to rvalue  
int&& ref3 = i + 3; // OK: can bind to rvalue
```

- ▶ Const references also bind to both, but don't allow modification

# Move constructor

```
template <class T>
class Set {
    T* data;
    int size;
    int maxSize;
public:
    Set(const Set&); // copy constructor
    Set(Set&&); // move constructor
    // many other methods
};
```

*argument  
not const!*

```
template<class T>
Set<T>::Set(const Set& other) :
    data(new T[max(other.getSize(), INITIAL_SIZE)]),
    size(other.getSize()),
    maxSize(max(other.getSize(), INITIAL_SIZE))
{
    for(int i = 0; i < size; i++) {
        data[i] = other.data[i];
    }
}
```

*simplified  
version*

copy constructor

```
Set<int> x, y;
// calls move constructor
Set<int> z = Set<int>::union(x, y);

// calls copy constructor
Set<int> w = x;
```

```
template<class T>
Set<T>::Set(Set&& other) :
    data(other.data),
    size(other.size),
    maxSize(other.maxSize)
{
    other.size = other.maxSize = 0;
    other.data = nullptr;
}
```

*other no longer  
controls original  
memory*

move constructor

# Move assignment operator

```
template <class T>
class Set {
    T* data;
    int size;
    int maxSize;
public:
    Set& operator=(const Set&); // copy assignment operator
    Set& operator=(Set&&);    // move assignment operator
    // many other methods
};
```

```
template<class T>
Set<T>& Set<T>::operator=(Set&& other)
{
    delete data;
    data = other.data;
    size = other.size;
    maxSize = other.maxSize;
    other.size = other.maxSize = 0;
    other.data = nullptr;
    return *this;
}
```

```
Set<int> x, y, z, w;

// calls move assignment operator
z = Set<int>::union(x, y);

// calls copy assignment operator
w = x;
```

more efficient and  
exception-safe

# The Big Five

- ▶ These five functions typically **go together**:
  - ◆ Copy constructor
  - ◆ **Move constructor**
  - ◆ Destructor
  - ◆ Copy assignment operator
  - ◆ **Move assignment operator**
- ▶ If you need to write **one**, you probably **need all five**

# Default move operations

- ▶ Default move constructor moves all fields
  - ◆ Supplied by compiler if **none** of the five is defined
  - ◆ Can be specified using **=default**
  - ◆ Otherwise, copy constructor is called
- ▶ Rules for move assignment operator are similar

# std::move

- ▶ Sometimes we want to force a move

```
vector<string> col1, col2;  
string str = "Hello"; // move, since "Hello" is an rvalue  
col1.push_back(str); // copy, since str is an lvalue  
col2.push_back(str); // copy; if str isn't used after this, it could be moved
```

- ▶ We can force the compiler to use move semantics by casting to rvalue

```
col2.push_back(std::static_cast<string&&>(str)); // invokes move version of push_back
```

- ▶ Instead, can use **std::move**

```
col2.push_back(std::move(str)); // equivalent form
```

# Implementing unique\_ptr

```
template<class T>
class unique_ptr {
    T* data;
public:
    explicit unique_ptr(T* ptr = nullptr);
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr(unique_ptr&&);
    unique_ptr& operator=(const unique_ptr&) = delete;
    unique_ptr& operator=(unique_ptr&&);
    ~unique_ptr();
    T& operator*() const;
    T* operator->() const;
};
```

```
template<class T>
unique_ptr::unique_ptr(unique_ptr&& other)
    : data(other.data) {
    other.data = nullptr;
}
```

```
template<class T>
unique_ptr<T>& unique_ptr<T>::operator=(unique_ptr&& other) {
    delete data;
    data = other.data;
    other.data = nullptr;
    return *this;
}
```

*other no longer in  
charge of original  
memory*

```
template<class T>
unique_ptr::unique_ptr(T* ptr)
    : data(ptr) {}
```

```
template<class T>
unique_ptr::~unique_ptr() {
    delete data;
}
```

```
template<class T>
T& unique_ptr::operator*() const {
    return *data;
}
```

```
template<class T>
T* unique_ptr::operator->() const {
    return data;
}
```

# std::move and unique\_ptr

```
unique_ptr<int> p2 = p1;    // compilation error: no copy c'tor  
unique_ptr<int> p3 = std::move(p1);    // OK! calls move c'tor
```

- ▶ **std::move** signals compiler to use move constructor or move assignment operator
- ▶ Implemented by a cast:

```
// equivalent form  
unique_ptr<int> p3 = std::static_cast<unique_ptr<int>&&>(p1);
```

# std::move

## ► std::move is needed:

- ◆ when passing ownership by assignment

```
unique_ptr<int> p3 = std::move(p1);
```

- ◆ when passing ownership by function argument

```
// some function
void process(unique_ptr<int> ptr);

unique_ptr<int> my_ptr;

process(my_ptr);           // fails, since cannot copy my_ptr to ptr
process(std::move(my_ptr)); // OK, initializes ptr using move c'tor
process(make_unique<int>(10)); // not needed here
```

# std::move

## ► std::move is not needed:

- ◆ when returning value from a function

```
unique_ptr<Employee> createEngineer(const char* name, int salary) {  
    unique_ptr<Engineer> eng(new Engineer(name));  
    eng->setSalary(salary);  
    return eng; // move elided  
}
```

- ◆ on the returned value

```
unique_ptr<Employee> emp = createEngineer("David", 10000);
```

# std::move

## ► Other applications of std::move:

### ◆ Move constructors

```
class Wrapper {  
    unique_ptr<Employee> employee;  
public:  
    Wrapper(Wrapper&& other) :  
        employee(std::move(other.employee)) {} // won't compile without std::move  
    since other.employee is lvalue  
};
```

### ◆ Move on last usage

```
void move_string_to_both(string str, vector<string> vec1, vector<string> vec2) {  
    vec1.push_back(str);           // makes a copy of str  
    vec2.push_back(std::move(str)); // str not used any more, so can be moved  
}
```

*“Move is an optimization of copy.”*

- Scott Meyers