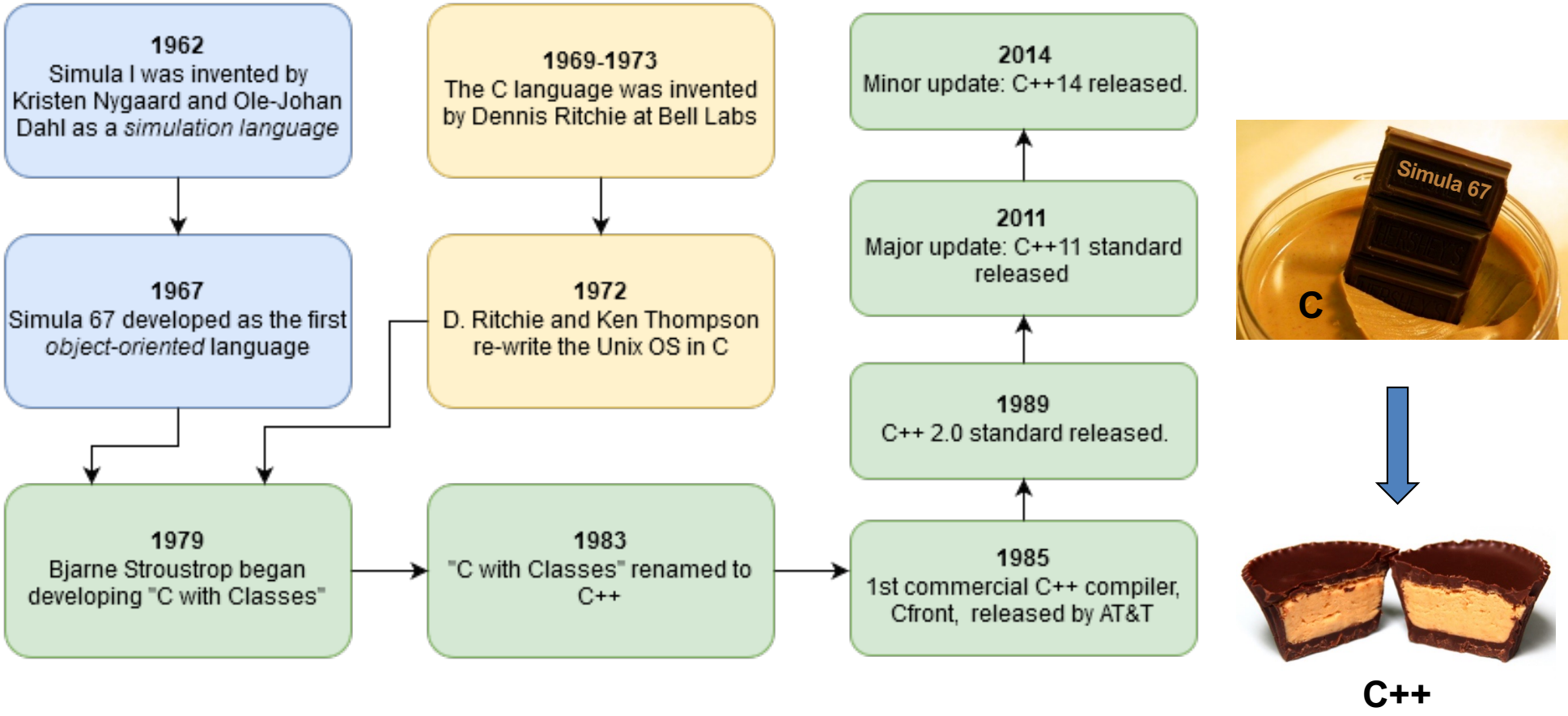
Data Structures and Algorithms

Lecture 6: C++ Programming

Very brief history of C++

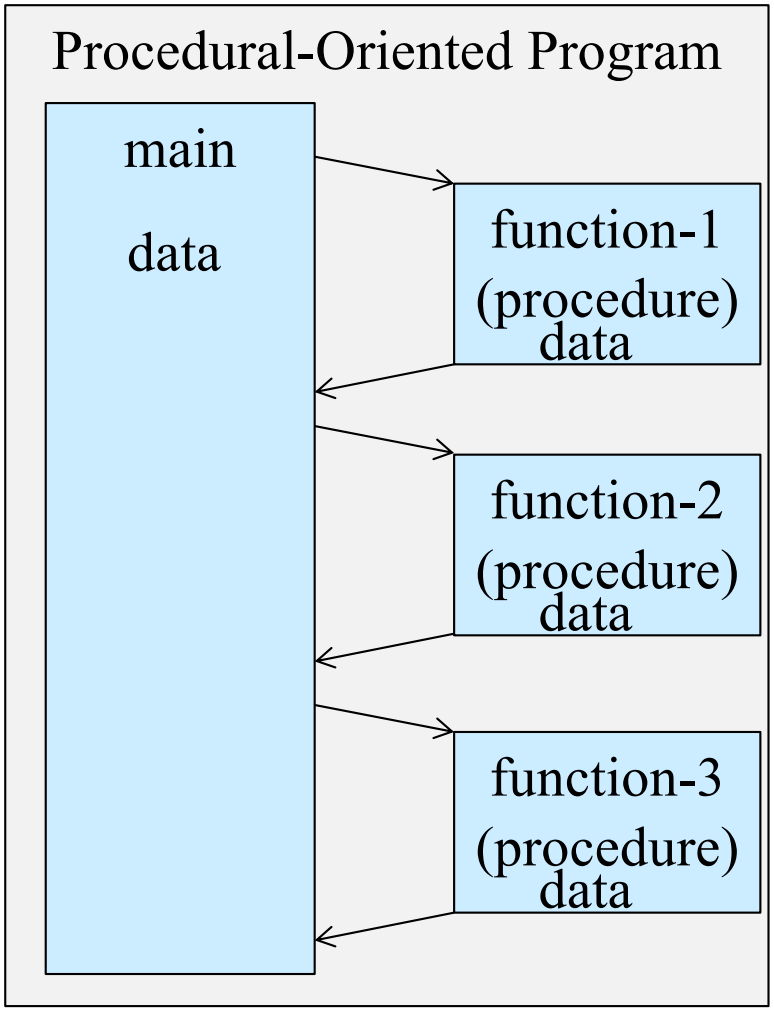


For details more check out [A History of C++: 1979−1991]

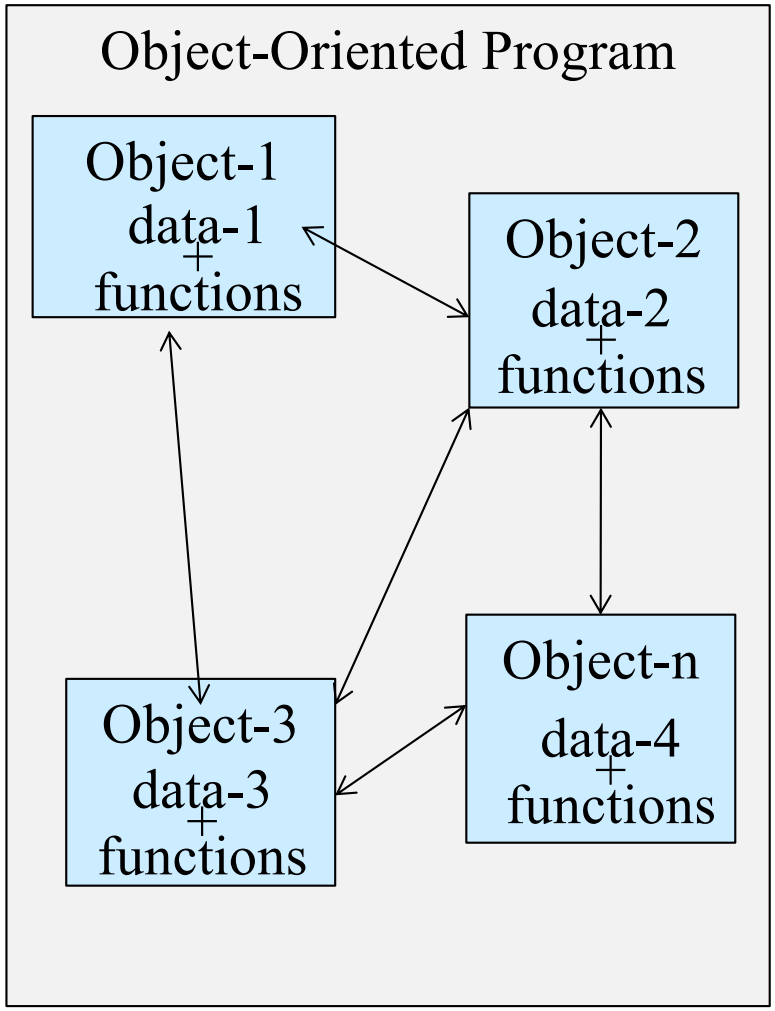
Brief Facts about C++

- Evolved from C
- Early 1980s: Bjarne Stroustrup (Bell Labs)
- Provides capabilities for Object-Oriented Programming (**OOP**)
  - Objects: resuable software components
    - Model items in real world
  - Object-oriented programs
    - Easy to understand, correct and modify
- C++ is a superset of C.
- Nowadays a language of its own!

Procedural-Oriented VS. Object-Oriented



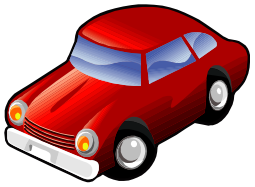
Modules interact by reading and writing state that is store in shared data structures.



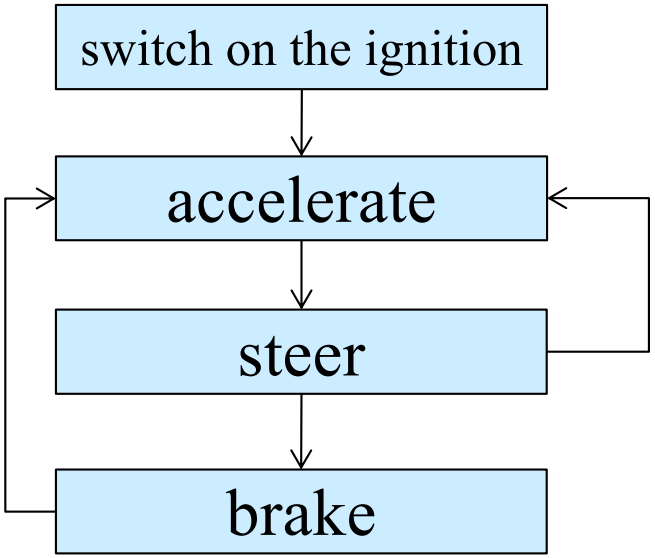
Modules in the form of objects interact by sending messages to each other.

Example: PO VS. OO

CAR

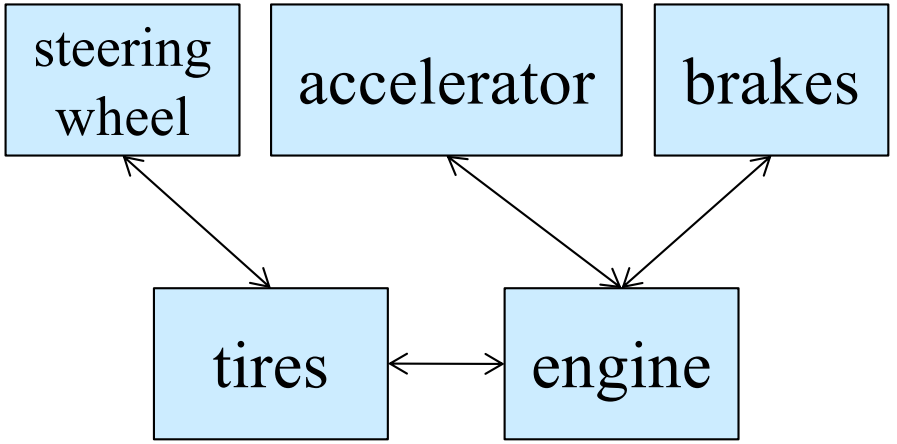


Procedure-oriented View of car operation



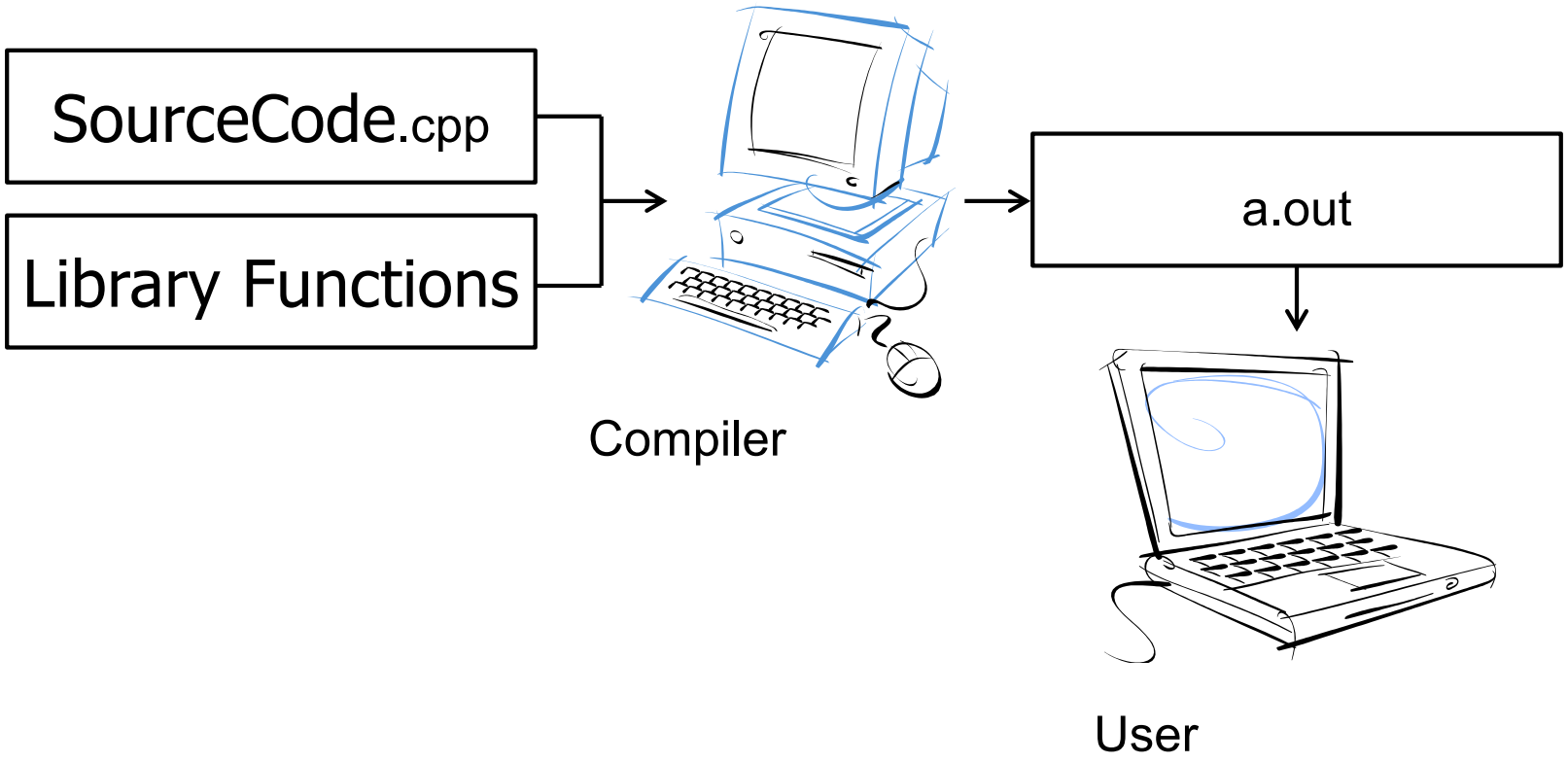
Car = a sequence of functions (procedures)

Object-oriented View of car operation

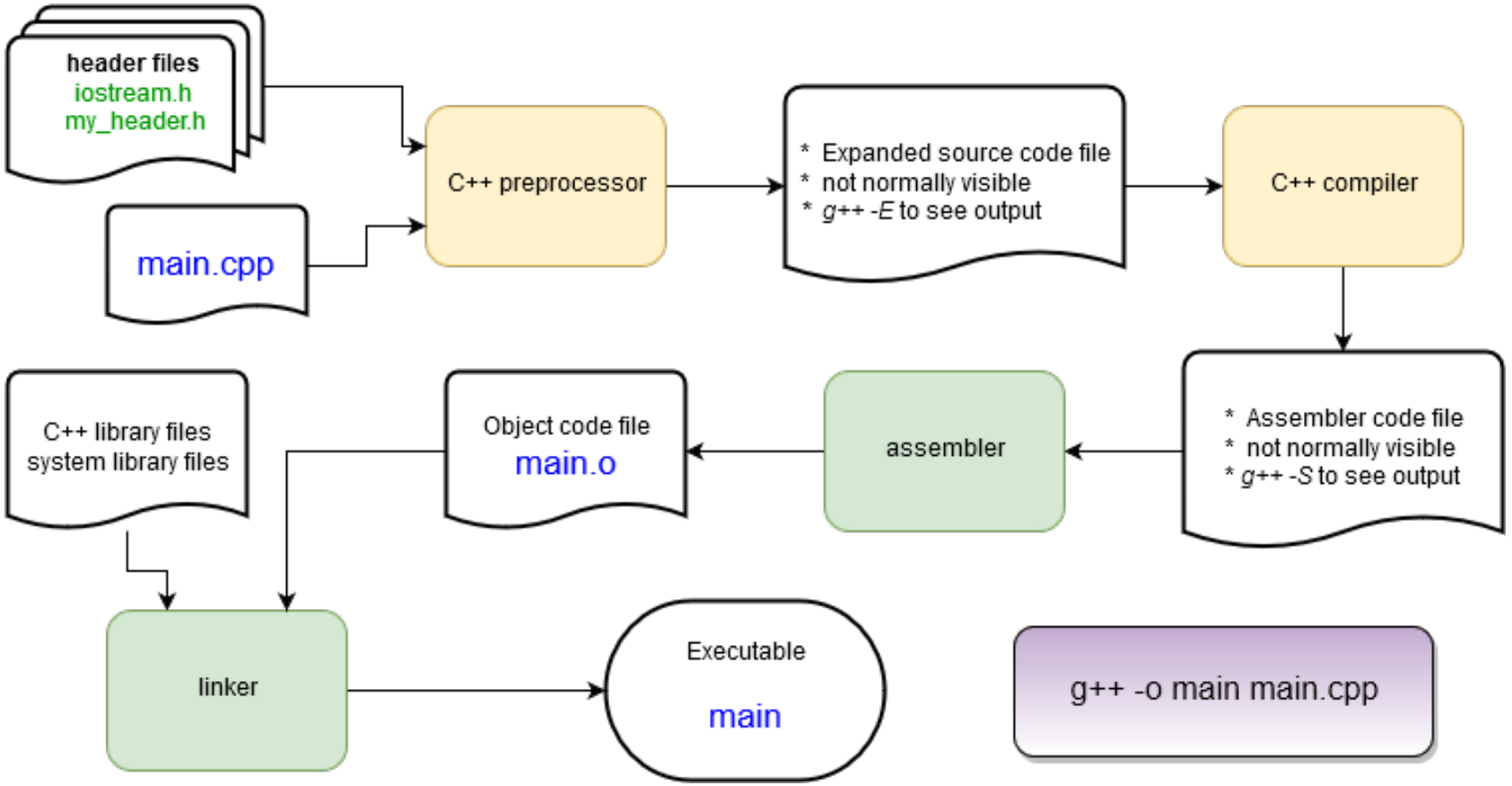


Car = interaction between components (objects)

The C++ Programming Model



The Compilation Process



Programming tools/compiler

- Windows
  - Dev-C++: https://www.bloodshed.net/ (**Easy to Go**)
  - VS Code: https://code.visualstudio.com/docs/setup/windows
- Mac OS
  - VS Code: https://code.visualstudio.com/docs/setup/mac
  - Xcode: https://developer.apple.com/xcode/

VS Code Setup Guide

- Part One: Installed VSCode IDE *successfully*!
- Part Two: Install a C++ Compiler
  - Windows
    - Follow the instructions at this link: https://code.visualstudio.com/docs/cpp/config-mingw
  - Mac OS
    - Follow the instructions at this link: https://code.visualstudio.com/docs/cpp/config-clang-mac
    - or this: **Step1**. Install Homebrew https://brew.sh/, and **Step2**. type "brew install gcc" in the terminal to install gcc.

Outline of Today's Lecture

- Basic Features of C++
- Class in C++
- Scope, Namespace, Casting, Control Flow
- Dynamic Memory Allocation
- Overloading, Polymorphism, Inline Function
- More on OOP and Class
  - Constructor and Destructor
  - Inheritance, Derivation, Overriding, Friend
- Template: Function and Class
- Exceptions
- File I/O

Basic features

Basic C++

- Inherit ALL C syntax
  - Primitive data types
    - Supported data types: `int, long, short, float, double, char, bool, and enum`
    - The size of data types is platform-dependent
  - Basic expression syntax
    - Defining the usual arithmetic and logical operations such as `+, -, /, %, *, &&, !,` and `||`
    - Defining bit-wise operations, such as `&, |,` and `~`
  - Basic statement syntax
    - `If-else, for, while,` and `do-while`

Basic C++ (cont)

- Add a new comment mark
  - `//` For a single line comment
  - `/*… */` for a group of line comment

- New data type
  - `Reference` data type "**&**". Much likes pointer.

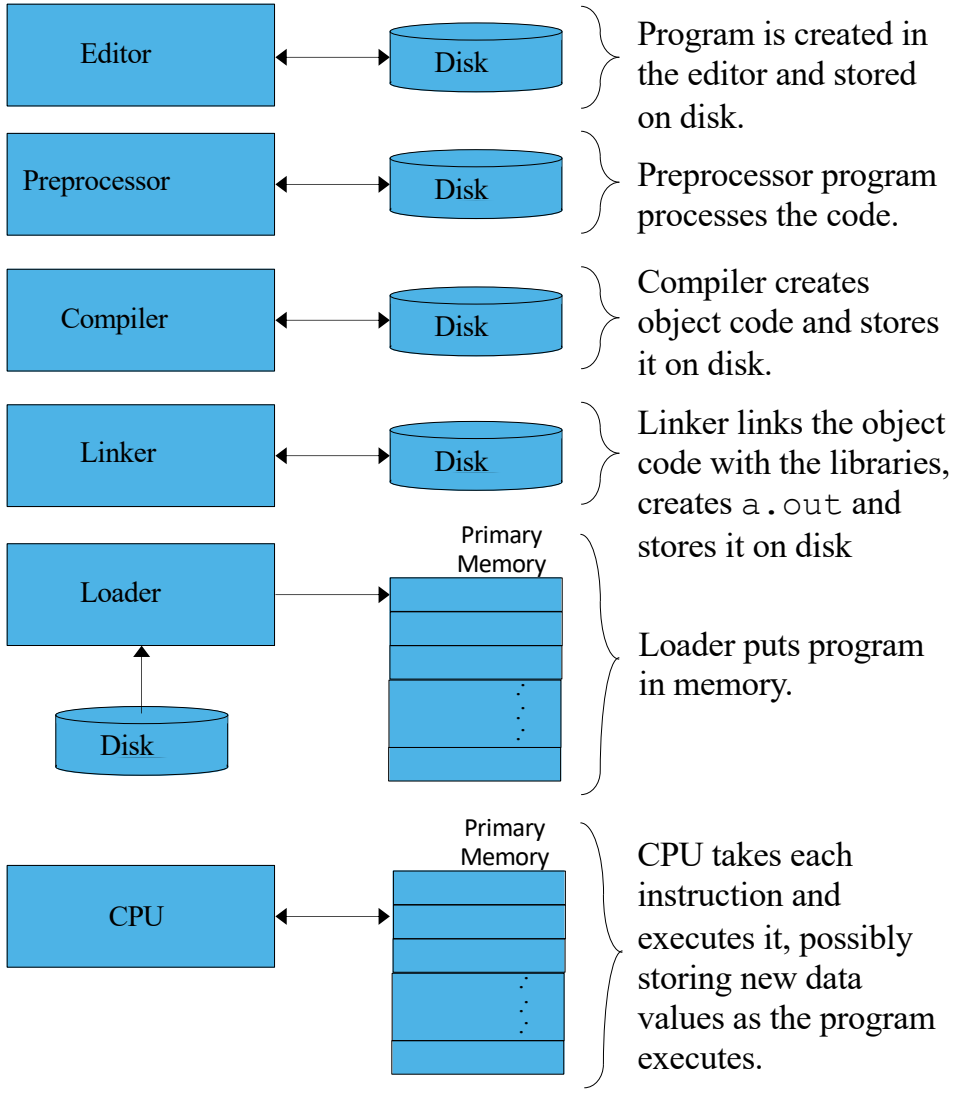
```
int ix;              // ix is "real" variable
int & rx = ix;  // rx is "alias" for ix
ix = 1;     /* also rx == 1 */
rx = 2;     /* also ix == 2 */
```

- *`const`* support for constant declaration, just likes C.

Basics of a Typical C++ Program

Phases of C++ Programs:

1. Edit
2. Preprocess
3. Compile
4. Link
5. Load
6. Execute



Main steps to create and run a C++ program

The steps are:

1. Create a new project.
2. Add a C++ source file to the project.
3. Enter your source code.
4. Include "lib_header_files.h" to the project. (optional)
5. Build an executable file.
6. Execute the program.

Example: A Simple C++ Program

- The infamous **Hello World** program!

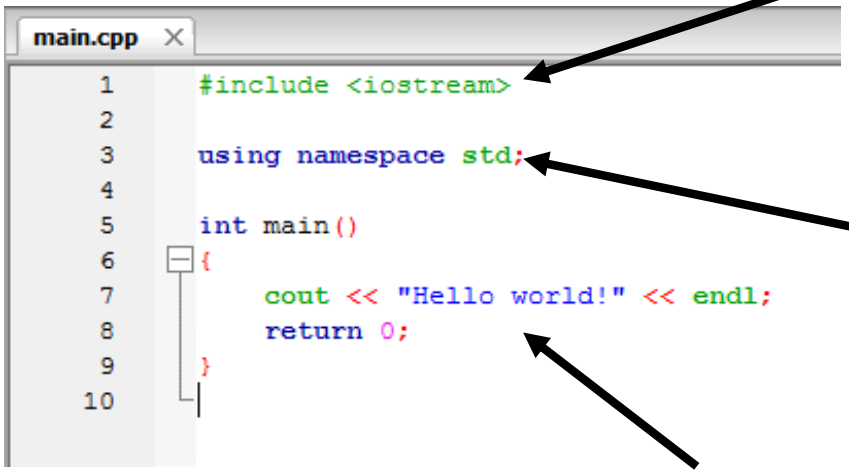
main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       cout << "Hello world!" << endl;
8       return 0;
9   }
10
```

The ***main*** routine – the start of **every** C++ program! It returns an integer value to the operating system and (in this case) takes no arguments: main()

The ***return*** statement returns an integer value to the operating system after completion. 0 means “no error”. C++ programs **must** return an integer value.

Example: A Simple C++ Program

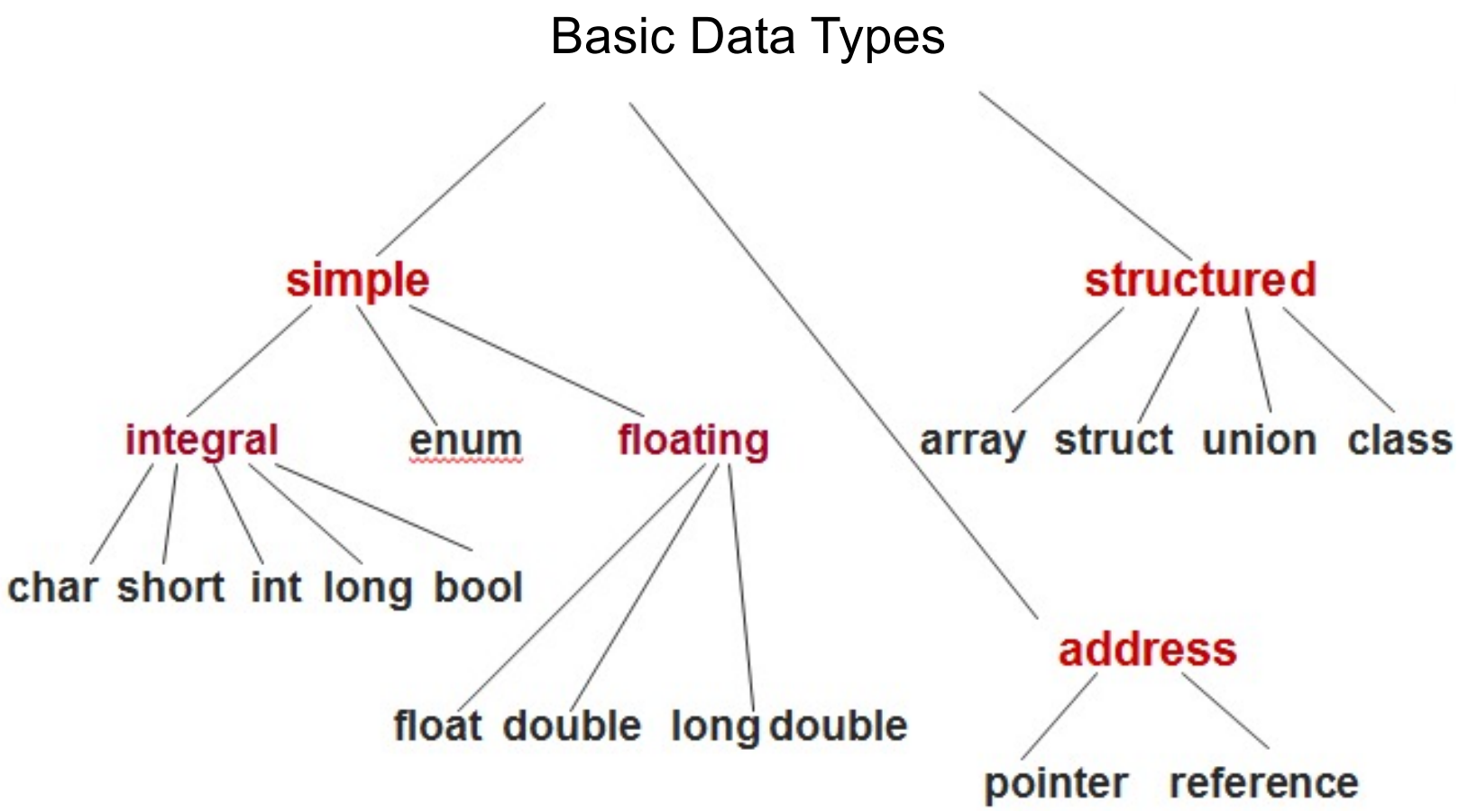


Load ***headers***; there are modules that include functions that you may use in your program; we will almost always need to include the header that defines cin and cout; the header is called <iostream.h>

Load a ***namespace*** called *std*. Namespaces are used to separate sections of code for programmer convenience.

- ***cout*** is the *object* that writes to the stdout device, i.e. the console window. It is part of the C++ standard library.
- **<<** is the C++ *insertion operator*. It is used to pass characters from the right to the object on the left.
- ***endl*** is the C++ newline character.

C++ Data Types


Basic Data Types
simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
address
float
double
long double
pointer
reference

Variable declaration

type variable-name;

Meaning: variable <variable-name> will be a variable of type <type>

Where type can be:

- `int` // integer
- `double` // real number
- `char` // character

Example:

```
int a, b, c;
double x;
int sum;
char my-character;
```

String

- C-style strings are implemented as an array of characters that ends with the null-character '\0'.
- C++ provides a string type as part of its "Standard Template Library" (STL).
  - Should include the header file "<string>"
  - STL: Collection of useful, standard classes and libraries in C++
- Full name of string type is "std::string"

```
#include <string>                  // Concatenated using + operator
using std::string;                 // Output using << operator
string s = "to be";
string t = "not " + s;             // t = "not to be"
string u = s + " or " + t;         // u = "to be or not to be"
if (s > t)                         // true: "to be" > "not to be"
   cout << u;                      // outputs "to be or not to be"
```

References

- An alternative name for an object (i.e., alias)
- The syntax "&" denotes a reference to an object
- It stores the memory location of other object.
- Cannot be `NULL`.
- Example:

```
string author = "Samuel Clemens";
string &penName = author;    // penName is an alias for author
penName = "Mark Twain";      // now author = "Mark Twain"
cout << author;              // outputs "Mark Twain"
```

Constants

- Adding the keyword `const` to a declaration
- The value of the associated object cannot be changed
- ex)

```
const double PI = 3.14159265;
const int CUT_OFF[] = {90, 80, 70, 60};
const int N_DAYS = 7;
const int N_HOURS = 24*N_DAYS; // using a constant expression
int counter[N_HOURS];          // constant used for array size
```

- Replace “#define” in C for the definition of constants

Typedef

- Define a new type name with keyword `typedef`
- Example

```
typedef char* BufferPtr;    // type BufferPtr is a pointer to char
typedef double Coordinate;  // type Coordinate is a double

BufferPtr p;                // p is a pointer to char
Coordinate x, y;            // x and y are of type double
```

Input statements

cin >> variable-name;

Meaning: read the value of the variable called <variable-name> from the user

Example:

```
cin >> a;
cin >> b >> c;
cin >> x;
cin >> my-character;
```

Output statements

cout << variable-name;

Meaning: print the value of variable <variable-name> to the user

cout << "any message ";

Meaning: print the message within quotes to the user

cout << endl;

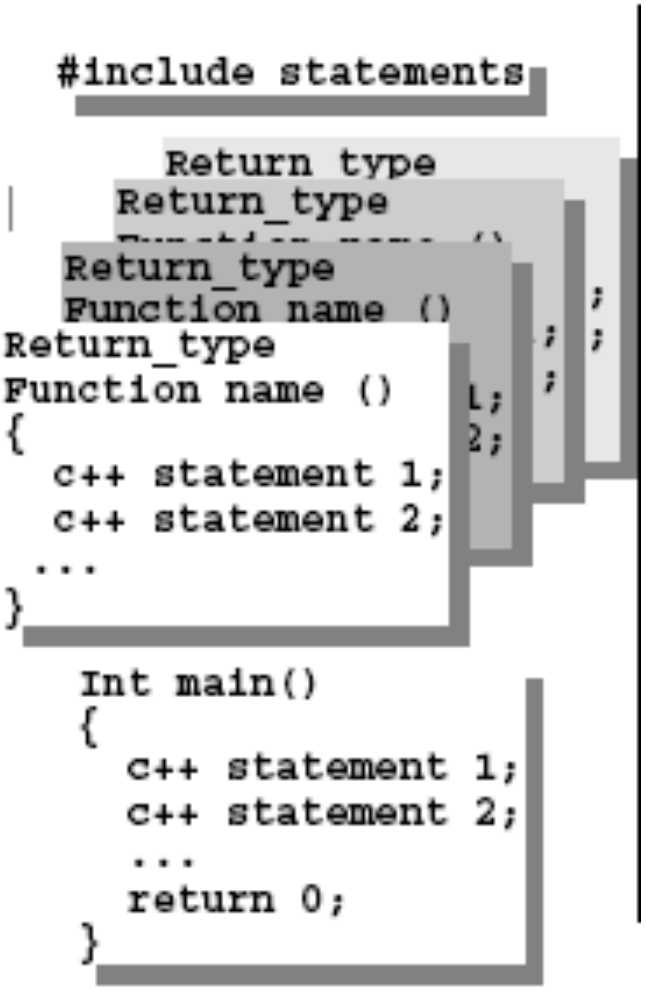
Meaning: print a new line

Example:

```
cout << a;
cout << b << c;
cout << “This is my character: “ << my-character << “ etc.”
<< endl;
```

Functions

- *functions* are abstractions that help you to reuse ideas and codes
  - make the code clearer, more logical and comprehensible



Functions

- *function prototyping*: a description of the types of arguments when declaring and defining a function
  - `void funct(float x, float y, float z);`
  - or having no arguments, `void funct(void)`

Example: Functions

- Return values
- Example →

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

char cfunc(int i) {
  if(i == 0)
    return 'a';
  if(i == 1)
    return 'g';
  if(i == 5)
    return 'z';
  return 'c';
}

int main() {
  cout << "type an integer: ";
  int val;
  cin >> val;
  cout << cfunc(val) << endl;
} ///:~
```

Parameter Passing

- Different ways to pass parameters into a function
  - Pass-by-value
  - Pass-by-address
  - Pass-by-reference

- Parameters are passed by value to a function
  - a copy of the parameters, and does NOT affect outside the function.

Pass-by-value

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

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

```
x = 47
a = 47
a = 5
x = 47
```

Pass-by-address

- A `pointer` is passed instead of a value.
- Pointer acts as an alias to an outside object.
- Any changes to the alias in the function DOES affect “outside” object.

Pass-by-address

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

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

Pass-by-reference

- C++ provide another way to pass an address into a function – *reference*
- Similar to pass-by-address
- Any changes to the objects in the function DOES affect "outside" objects.
- Note, Pass-by-constant-reference will NOT allow change the objects inside the function.

Pass-by-reference

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

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Looks like pass-by-value,
        // is actually pass by reference
  cout << "x = " << x << endl;
} ///:~
```

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

Class in C++

Class

- A tool for creating new types
- Conveniently used as if the built-in type, but user-defined
- Derived classes and templates – related classes are organized in a specific way according to their relationships
- Note: Class is an abstraction of a group of objects, while an object is an instance of the class

Class

Class Designer
Design and implement classes
To be used by other programmers

Objectives:
Efficient algorithms
Convenient coding

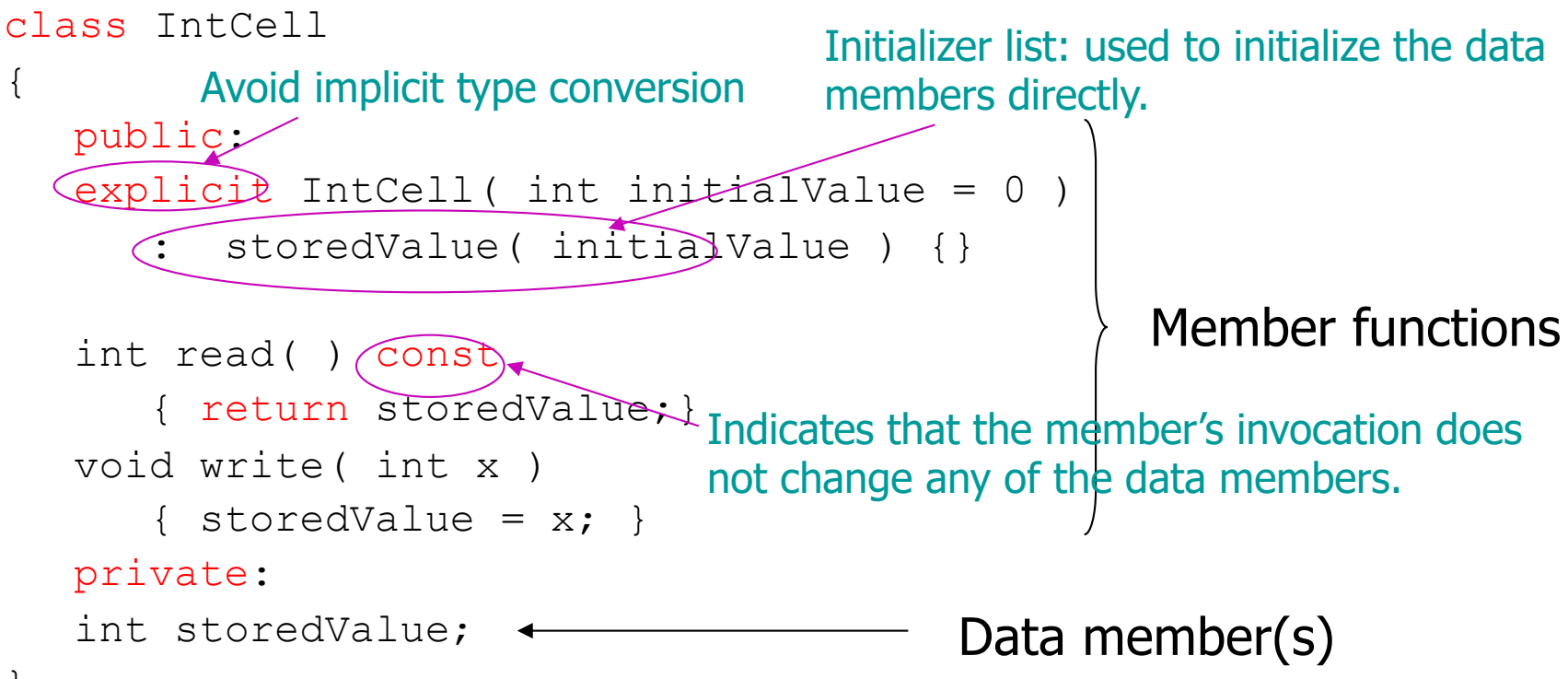
Class User
Programmers use the classes designed by class designer

Objectives:
Use of the public operations, no need to know internal implementations;

Hope the set of interface is large to solve the problems, but small enough to comprehen

Example: Class Definitions

- A C++ class consists of ***data members*** and *methods (**member functions**)*.



Class - Encapsulation

- Two labels: `public` and `private`
  - Determine visibility of class members
  - A member that is *public* may be accessed by any method in any class
  - A member that is *private* may only be accessed by methods in its class

- Information hiding
  - Data members are declared *private,* thus restricting access to internal details of the class
  - Methods intended for general use are made *public*

Class - Interface and Implementation

- In C++, it is more common to separate the class *interface* from its *implementation*.

- The ***interface*** lists the class and its members (data and functions).
- The ***implementation*** provides implementations of the functions.

Class – Member Functions

- Functions declared within a class definition
- Invoked only for a specific variable of the appropriate type

```
class Date{
    int d, m, y;

public:
    void init(int dd, int mm, int yy);  // initalize

    void add_year(int n);   // add n years
    void add_month(int n);  // add n months
    void add_day(int n);    // add n days
}
```

```
int main()
{
    Date today;
    today.init(1, 9, 2005);
    today.add_day(7);
}
```

Class – Constructor

- A special function for the initialization of class objects
- It has the same name as the class itself
- Default or user-defined constructors

Class - Constructor

```
class Date{
     int d, m, y;

public:
     //..
     Date(int, int, int);     // day, month, year
     Date(int, int);          // day, month
     Date(int);               // day
     Date();
     Date(const char*)
}


int main()
{
     Date now;
     Date today(1);
     Date sept("Sept 1, 2005");
}
```

Constructors

Class – Access Control

- Three keywords/categories: `public`, `private`, and `protected`
- ***public*** means all member declarations that follow are available to everyone
- The ***private*** keyword, means that no one can access that member except designer, the creator of the type, inside function members of that type

Class – Access Control

- ***Protected*** acts just like **Private**, except that it allow the inherited class to gain access.
- Example

```
class X {
public:
    void interfaceFunc();
protected:
    void protectedFunc();
private:
    void privateFunc();
};
```

Scope, Namespace, Casting, Control Flow

Local and Global Variables

- Block
  - Enclosed statements in {...} define a **`block`**
  - Can be nested within other block
- Local variables are declared within a block and are only accessible from within the block
- Global variables are declared outside of any block and are accessible from everywhere
- Local variable hides any global variables of the same name

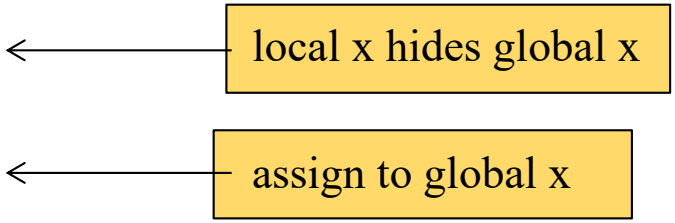
Local and Global Variables

- ex)

```
const int cat = 1;              // global cat


int main () {
      const int cat = 2;        // this cat is local to main
      cout << cat;              // outputs 2 (local cat)
      return EXIT_SUCCESS;
}
int dog = cat;                  // dog = 1 (from the global cat)
```

Scope Resolution Operator (::)

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

int  x;

int main()
{
      int  x;
      x = 1;
      ::x = 2;

      cout << "local x = " << x << endl;
      cout << "global x = " << ::x << endl;

      return 0;
}
```




```
result>
local x = 1
global x = 2
```

Namespaces: Motivation

- Two companies A and B are working together to build a game software "Snake"

- A uses a global variable
  - `struct Tree {};`
- B uses a global variable
  - `int Tree;`

- Compile? *Failure*!
- Solution
  - A: struct **Atree** {}; B: int **BTree**; → dirty, time consuming, inconvenient

- Let's define some "name space"
- Very convenient in making "large" software

Namespaces

- A mechanism that allows a group of related names to be defined in one place
- Access an object x in namespace group using the notation `group::x`, which is called its fully qualified name
- ex)

```
namespace myglobals {
      int cat;
      string dog = "bow wow";
}
myglobals::cat = 1;
```

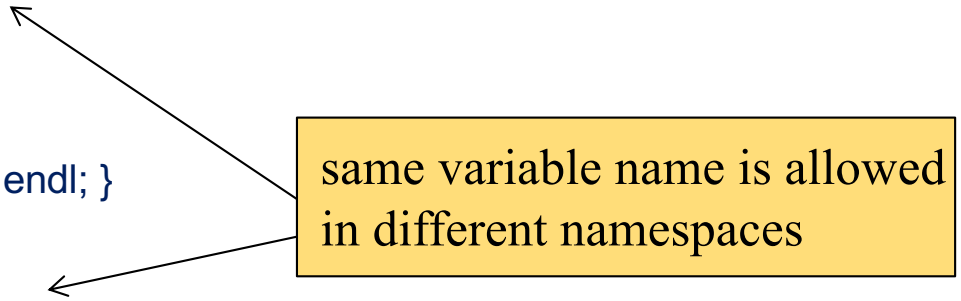
The `Using` Statement

- `Using` statement makes some or all of the names from the namespace accessible, without explicitly providing the specifier
- ex)

```
using std::string;              // makes just std::string accessible
using std::cout;                // makes just std::cout accessible


using namespace myglobals;  // makes all of myglobals accessible
```

Example: Namespace

```
#include <iostream>
namespace IntSpace{
    int data;
    void add(int n){ data += n; }
    void print(){ std::cout << data << std::endl; }
}
namespace DoubleSpace{
    double data;
    void add(double n){ data += n; }
    void print(){ std::cout << data << std::endl; }
}
int main()
{
    IntSpace::data = 3;
    DoubleSpace::data = 2.5;
    IntSpace::add(2);
    DoubleSpace::add(3.2);
    IntSpace::print();
    DoubleSpace::print();
    return 0;
}
```



result>

5

5.7

Type Casting

```
int cat = 14;
double dog = (double) cat;     // traditional C-style cast
double pig = double(cat);      // C++ functional cast

int i1 = 18;
int i2 = 16;
doubled v1 = i1 / i2;                          // dv1 = 1.0
doubled v2 = double(i1) / double(i2);          // dv2 = 1.125
doubled v3 = double( i1 / i2);                 // dv3 = 1.0
```

Static Casting (to give "warning")

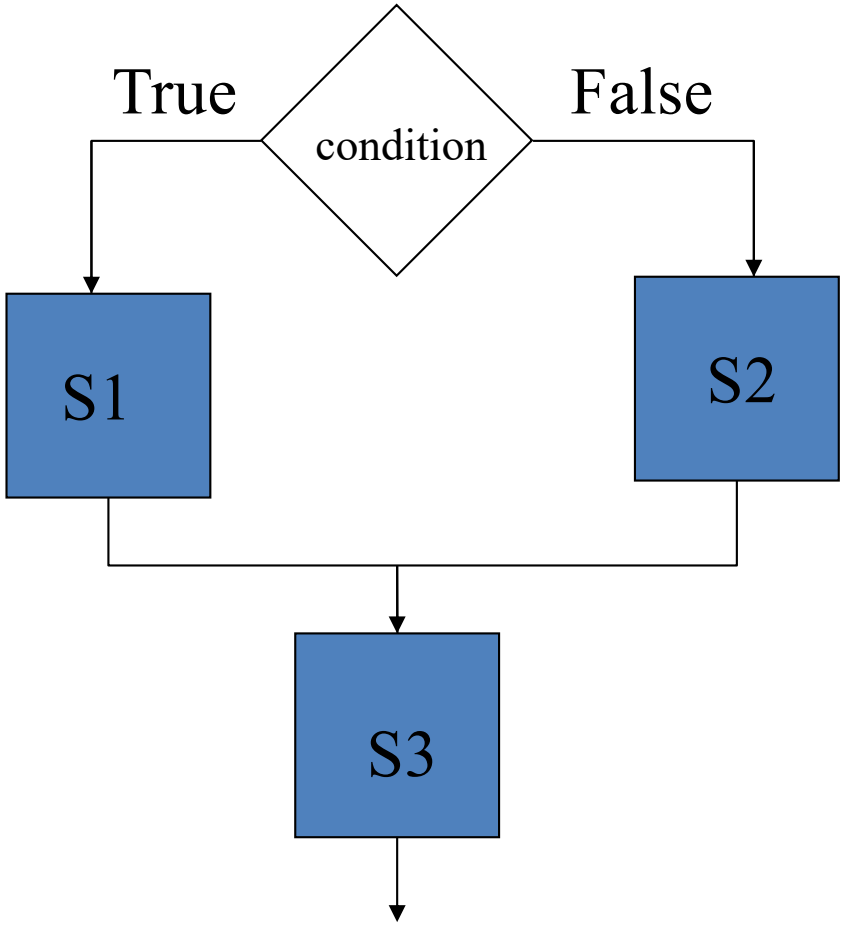
```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = static_cast<int>(d1);   // i1 = 3
int i2 = static_cast<int>(d2);   // i2 = 3
```

Implicit Casting

```
int i = 3;
double d = 4.8;
double d3 = i / d; // d3 = 0.625 = double(i) / d
int i3 = d3;       // i3 = 0 = int(d3)

// Warning! Assignment may lose information
```

Control Flow: If statement

```
if (condition) {
    S1;
}
else {
    S2;
}
S3;
```



Control Flow: Boolean conditions

- Comparison operators

  == equal

  != not equal

  < less than

  > greater than

  <= less than or equal

  >= greater than or equal

- Boolean operators

  && and

  || or

  ! not

Control Flow: Condition Examples

Assume we declared the following variables:

```
int a = 2, b=5, c=10;
```

Here are some examples of boolean conditions we can use:

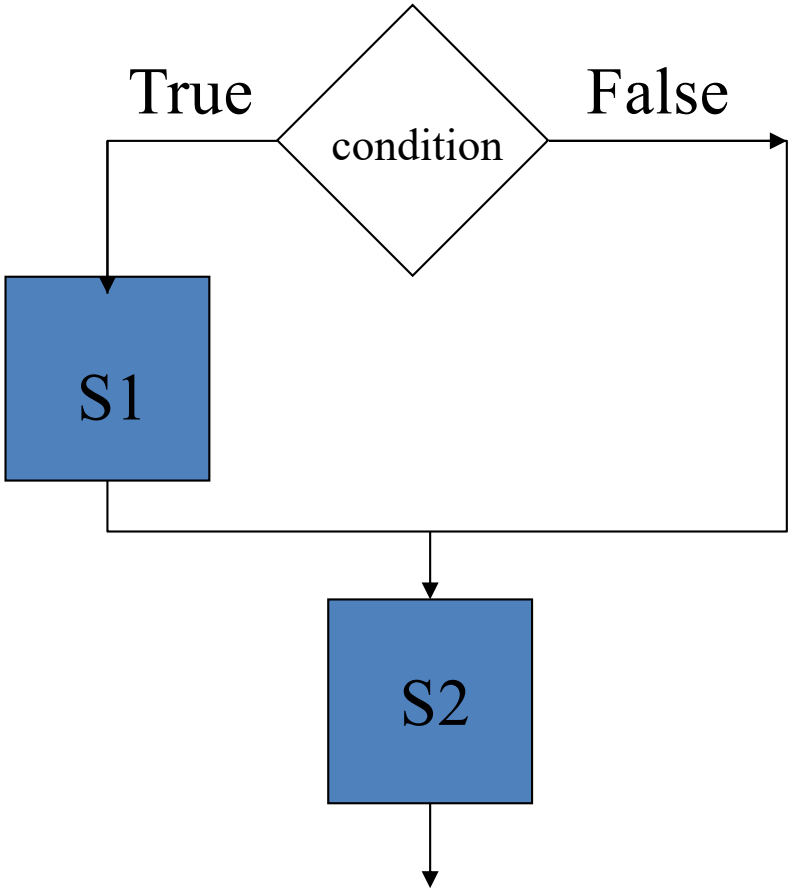
- `if (a == b) …`
- `if (a != b) …`
- `if (a <= b+c) …`
- `if(a <= b) && (b <= c) …`
- `if !((a < b) && (b<c)) …`

Control Flow: If example

```
#include <iostream.h>

void main() {
int a,b,c;
cin >> a >> b >> c;

if (a <=b) {
     cout << “min is “ << a << endl;
     }
else {
     cout << “ min is “ << b << endl;
}
cout << “happy now?” << endl;
}
```

Control Flow: While statement

```
while (condition) {
    S1;
}
S2;
```



Control Flow: While example

```
//read 100 numbers from the user and output their sum
#include <iostream.h>

void main() {
int i, sum, x;
sum=0;
i=1;
while (i <= 100) {
    cin >> x;
    sum = sum + x;
    i = i+1;
}
cout << "sum is " << sum << endl;
}
```

More Control Flows

- Do-While statement

```
do {
   loop_body_statement
}
while (<condition_exp>)
```

- Switch statement

```
switch (command){
        case '1': {};
        case '2': {};
        ...
}
```

- For loop

```
for ([<initialization>]; [<condition_exp>]; [<increment>])
        <body_statement>
```

Dynamic Memory Allocation

Memory Allocation

- Stack memory allocation
  - e.g., Non-static local variables
  - Such memory allocations are placed in a system memory area called the *stack*.
- Static memory allocation
  - e.g., static local or global variables
  - Static memory allocation happens before the program starts, and persists through the entire life time of the program.

Memory Allocation

- Dynamic memory allocation
  - It allows the program determine how much memory it needs at run time, and allocate exactly the right amount of storage.

- Note: the program has the responsibility to free the dynamic memory it allocated.

Dynamic Memory Allocation

- Create objects dynamically in the 'free store'.
- The operator '`new`' dynamically allocates the memory from the free store and returns a pointer to this object.
- The operator '`delete`' destorys the object and returns its space to the free store.

Dynamic Memory Allocation

- Example

```
Passenger *p;
//...
p = new Passenger;          //  p points to the new Passenger
p->name = "Pocahontas";  //  set the structure members
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = "NONE";
//...
delete p;                          //  destroy the object p points
```

Overloading, Polymorphism, Inline Function

Function Overloading

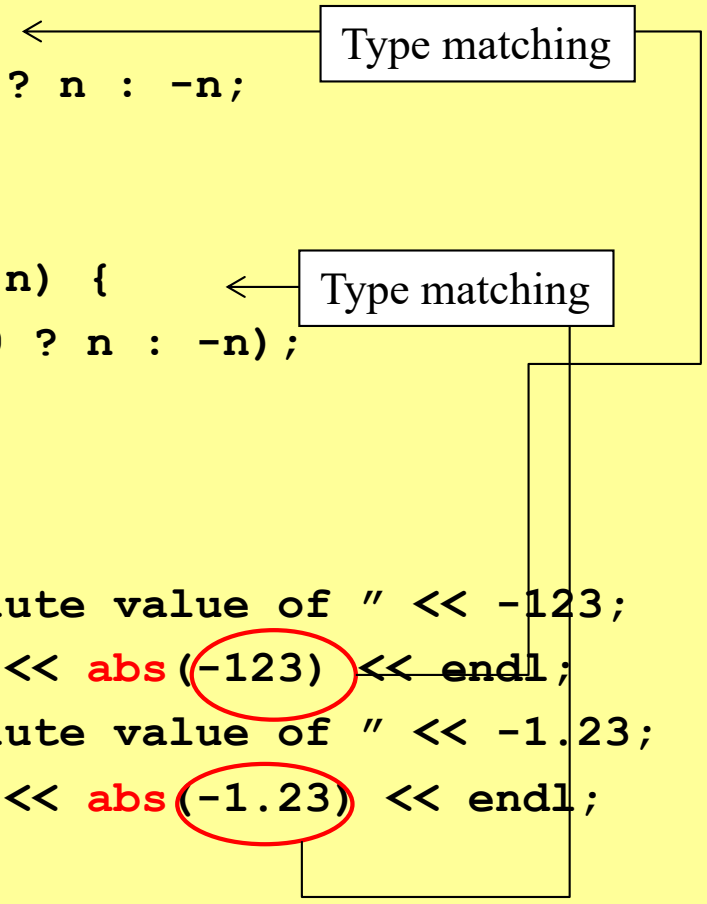
```
#include<iostream>
using namespace std;

int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```

In C, you can't use the same name for different functions

C++ allows multiple functions with the same name: the right function is determined at runtime based on argument types

Function Overloading

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

int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```



In C, you can't use the same name for multiple function definitions

C++ allows multiple functions with the same name as long as argument types are different: the right function is determined at runtime based on argument types

C++ Operator overloading

- User can overload operators for a user-defined class or types
  - e.g., string s1("ab"); string s2("cd"); string s = s1+s2;
  - define an operator as a function to overload an existing one
  - operator followed by an operator symbol to be defined.
    - define an operator + ➔ operator+
    - define an operator ++ ➔ operator++
    - define an operator << ➔ operator <<
  - To avoid confusion with built-in definition of overload operators, all operands in the basic types (int, long, float) are not allowed

Example : Operator Overloading

```
#include <iostream>
using namespace std;
enum Day { sun, mon, tue, wed, thu, fri,sat };

Day& operator++(Day& d)
{
  return d = (sat == d) ? sun : Day(d+1);
}
void print(Day d){
  switch(d){
   case sun : cout << "sun\n"; break;
   case mon : cout << "mon\n"; break;
   case tue : cout << "tue\n"; break;
   case wed : cout << "wed\n"; break;
   case thu : cout << "thu\n"; break;
   case fri : cout << "fri\n"; break;
   case sat : cout << "sat\n"; break;
  }
}
```

Operator overloading

```
int main()
{
  Day d = tue;
  cout << "current : ";
  print(d);
  for(int i = 0; i < 6; i++){
      ++d;
  }
  cout << "after 6 days : ";
  print(d);
  return 0;
}
```

use of overloaded operator

Result >

current : tue

after 6 days : mon

Polymorphism

- Allow values of different data types to be handled using *a uniform interface*.
- One function name, various data types
  - Function overloading
- Merit
  - improve code readability
- Ex.

C	abs ( )	labs ( )	fabs ( )
	int	long int	floating point
C++	abs ( )		
	int	long int	floating point

Inline Functions

C (Macro functions)

```
#include <stdio.h>
#define square(i)  i*i
#define square2(i) ((i)*(i))
#define  pr(i)  printf("value
= %d\n", (i))

main( ) {
  int i = 1, j = 1, k;
  k = square(i+1); pr(k);
  k = square2(j+1); pr(k);
  k = 100/square(2); pr(k);
  k = 100/square2(2); pr(k);
}
```

i+1*i+1

100/2*2

wrong answer

value = 4

value = 100 // wrong answer

value = 25

Side effect of macro functions

C++ (Inline functions)

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

inline int square(int i) { return i*i; }
inline void pr(int i) { cout << "value = "
<< i << endl; }

main( ) {
  int i = 1, j = 1, k;
    k = square(i+1); pr(k);
    k = 100/square(2); pr(k);
}
```

Function body is expanded at the point of function call during compile-time.

Similar to macro function

Result >

value = 4

value = 25

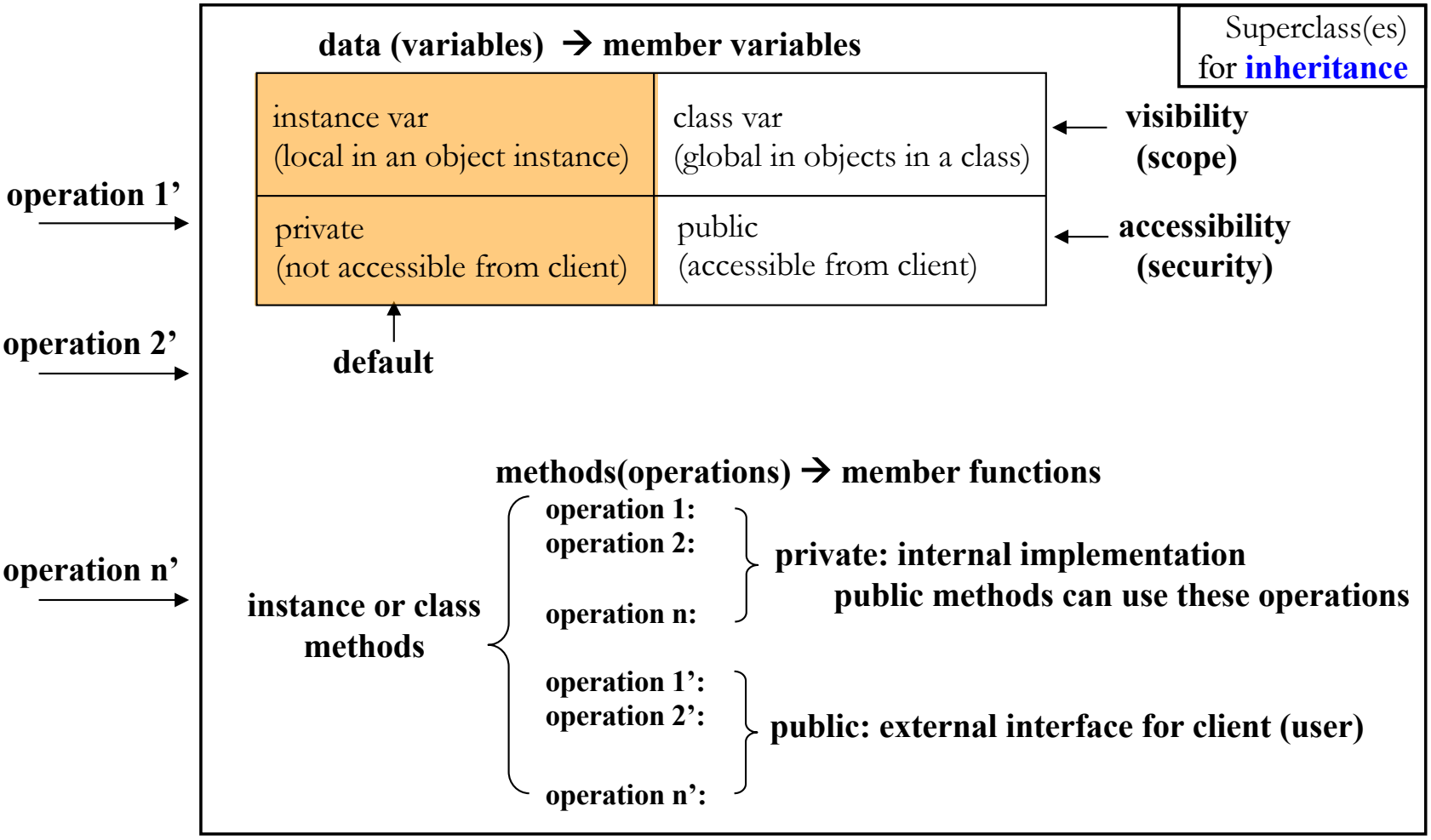
No side effect

More on OOP and Class

More on OOP and Class

-- Constructor and Destructor

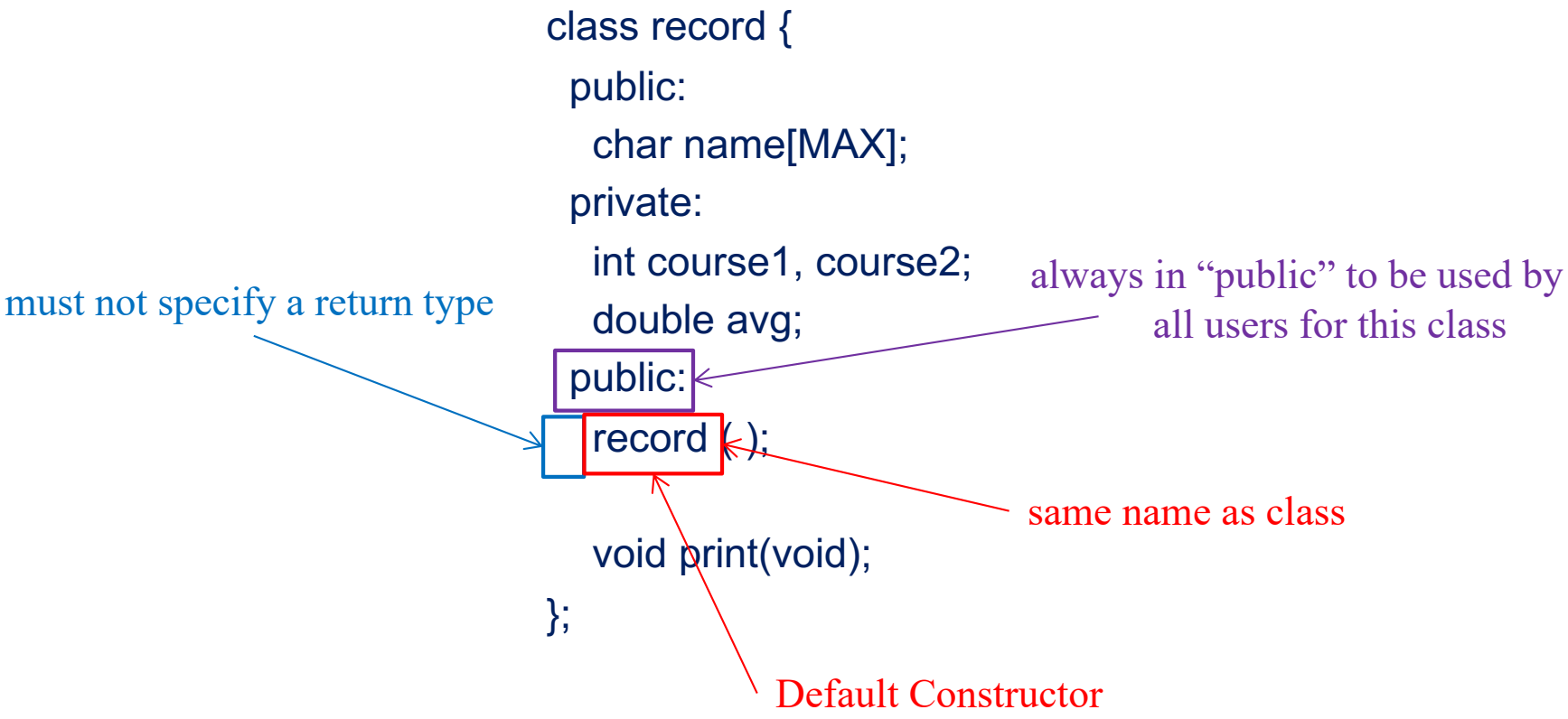
Class Structure in General Form



Constructors

- A ***constructor*** is a special method that describes how an instance of the class (called *object*) is constructed.
  - which will be called whenever an instance of the class is created.
- C++ provides a *default constructor* for each class.
  - The default constructor has no parameters.
- But we can define multiple constructors ***w/o parameters*** for the same class, and may even redefine the default constructor.

Default Constructor with No Argument



Constructors with Arguments

```
#include<iostream>
using namespace std;
#define MAX 10

class record {
  public:
    char name[MAX];
  private:
    int course1, course2;
    double avg;
  public:
    record();
    record(char*, int);
    record(char*, int, int);
    void print(void);
};
```

```
record::record() {
  strcpy(name, "");
  course1 = course2 = 100;
  avg = 100;
}

record::record(char *str, int score) {
  strcpy(name, str);
  course1 = course2 = score;
  avg = score;
}

record::record(char *str, int score1, int score2) {
  strcpy(name, str);
  course1 = score1; course2 = score2;
  avg = ((double) (course1 + course2)) / 2.0;
}
```

```
void record::print(void) { … }

int main( ) {
  record myRecord;
  record yourRecord = record("KIM", 80, 100);
  record hisRecord("LEE", 70);

  myRecord.print( );
  yourRecord.print( );
  hisRecord.print( );

  return 0;
}
```

overloading

shorthand notation

same as
record hisRecord = record(“LEE”, 70);

Destructor

- A ***destructor*** is called whenever an object goes out of scope or is subjected to a `delete`.
  - Typically, the destructor is used to free up any resources that were allocated during the use of the object.

- C++ provides a *default destructor* for each class
  - The default simply applies the destructor on each data member.
  - But we can redefine the destructor of a class.

- A C++ class can have ONLY one destructor.

Example: Destructors

```
class record {
 public:
   char name[MAX];
 private:
   int course1, course2;
   double avg;
 public:
   record ( ) { ... }
   ~record ( ) {
     ...
   }
   void print(void);
};
```

```
int main( ) {
  record myRecord;
  …
  return 0;
}
```

record::~record( ) invoked for myRecord

always in "public"

must not specify a return type

Destructor

the tag name of the class prefixed with a tilde ("~")

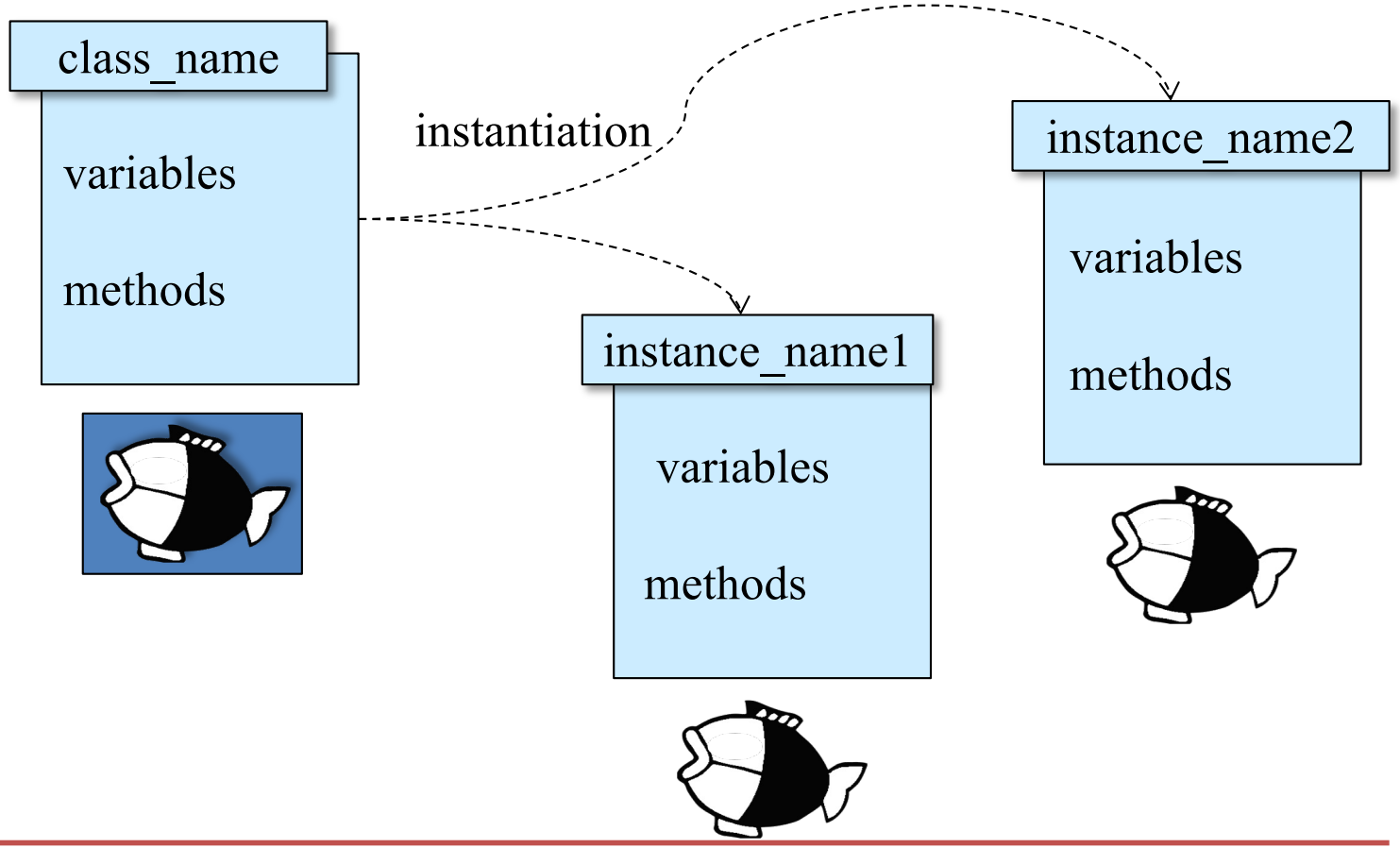
More on OOP and Class

-- Inheritance, Derivation, Overriding, Friend

Recall: Class Declaration

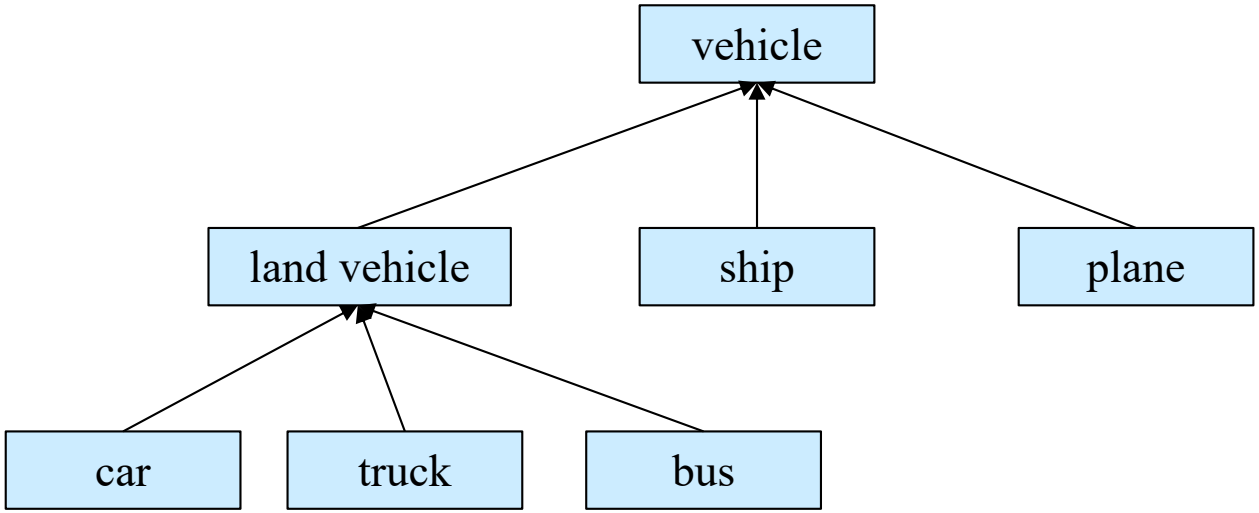
class_name *instance_name1, instance_name2;*

C.f. struct *tag_name struct_variable, … ;*



Inheritance (1/2)

- **Subclassing**: define a class based on another class
  - Another class = parent class (or superclass)
  - New class = child class (subclass)
  - Hierarchical classification in a tree form
  - Another way of "polymorphism"

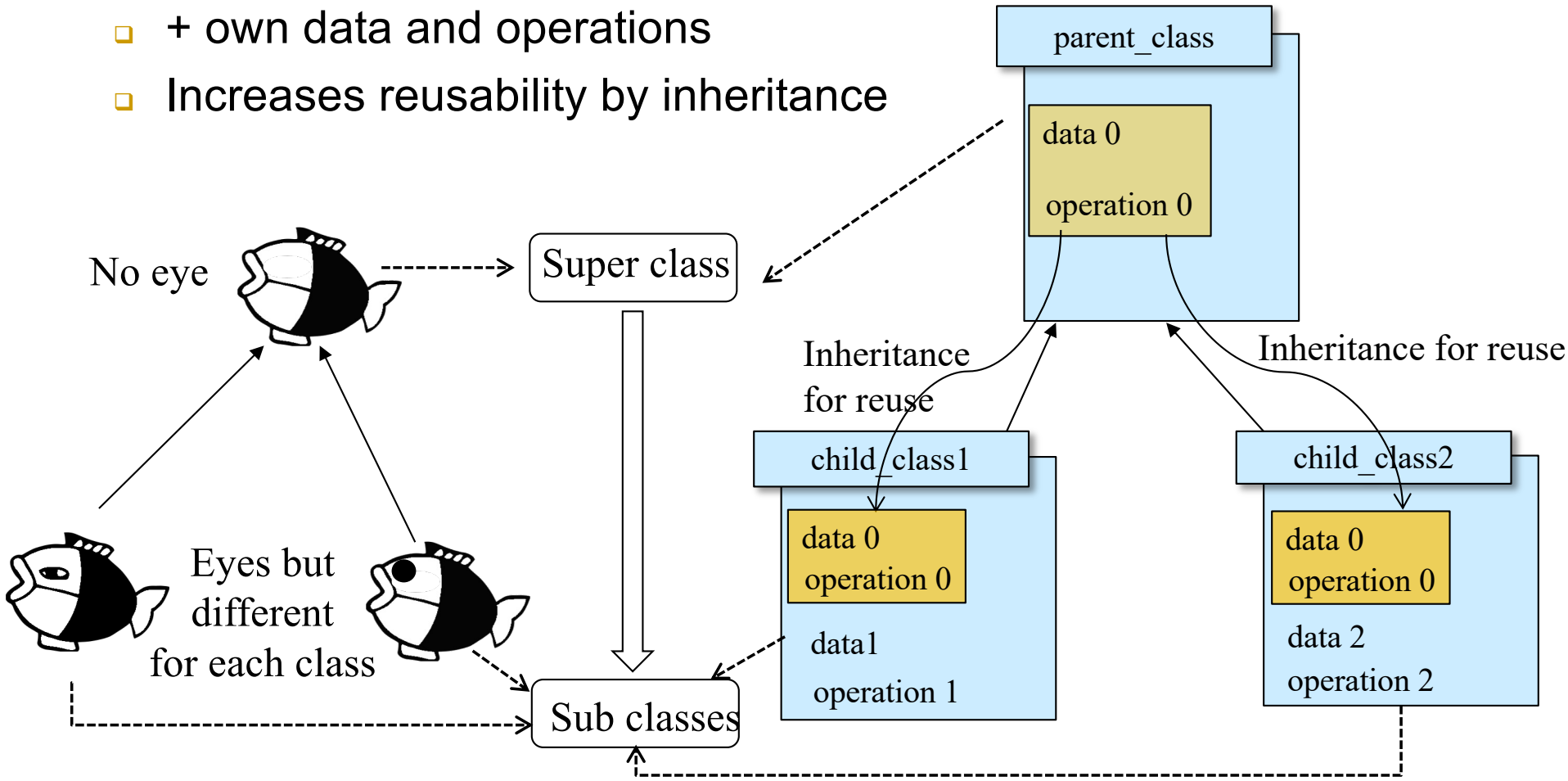


Superclass → subclass

- overrides information in superclass
- refines information in superclass to detailed one
- adds more information to one in superclass

Inheritance (2/2)

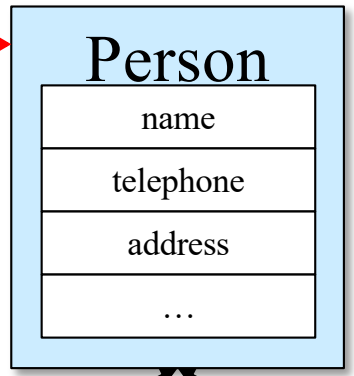
- Inheritance
  - Inherits data (attributes) and operations (behaviors) from parent
  - + own data and operations
  - Increases reusability by inheritance



Inheritance: Mechanism for Reuse

Base class
(Parent, Superclass)

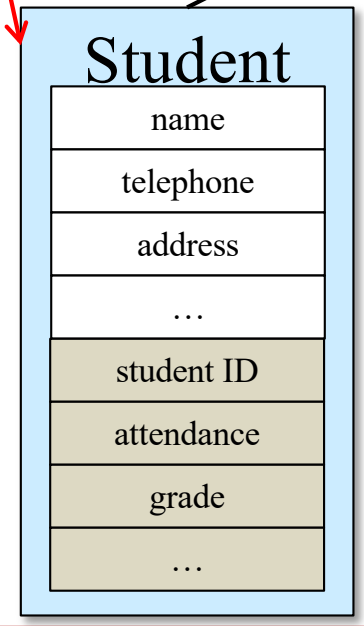
Derived class
(Child, subclass)

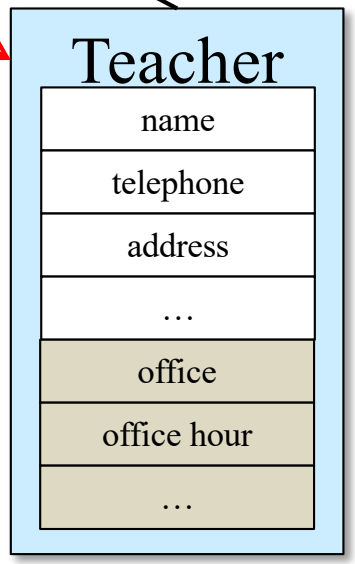



```
class Person {
    char *name, *telephone, *address;
    ...
};
```

Class derivation

Class derivation



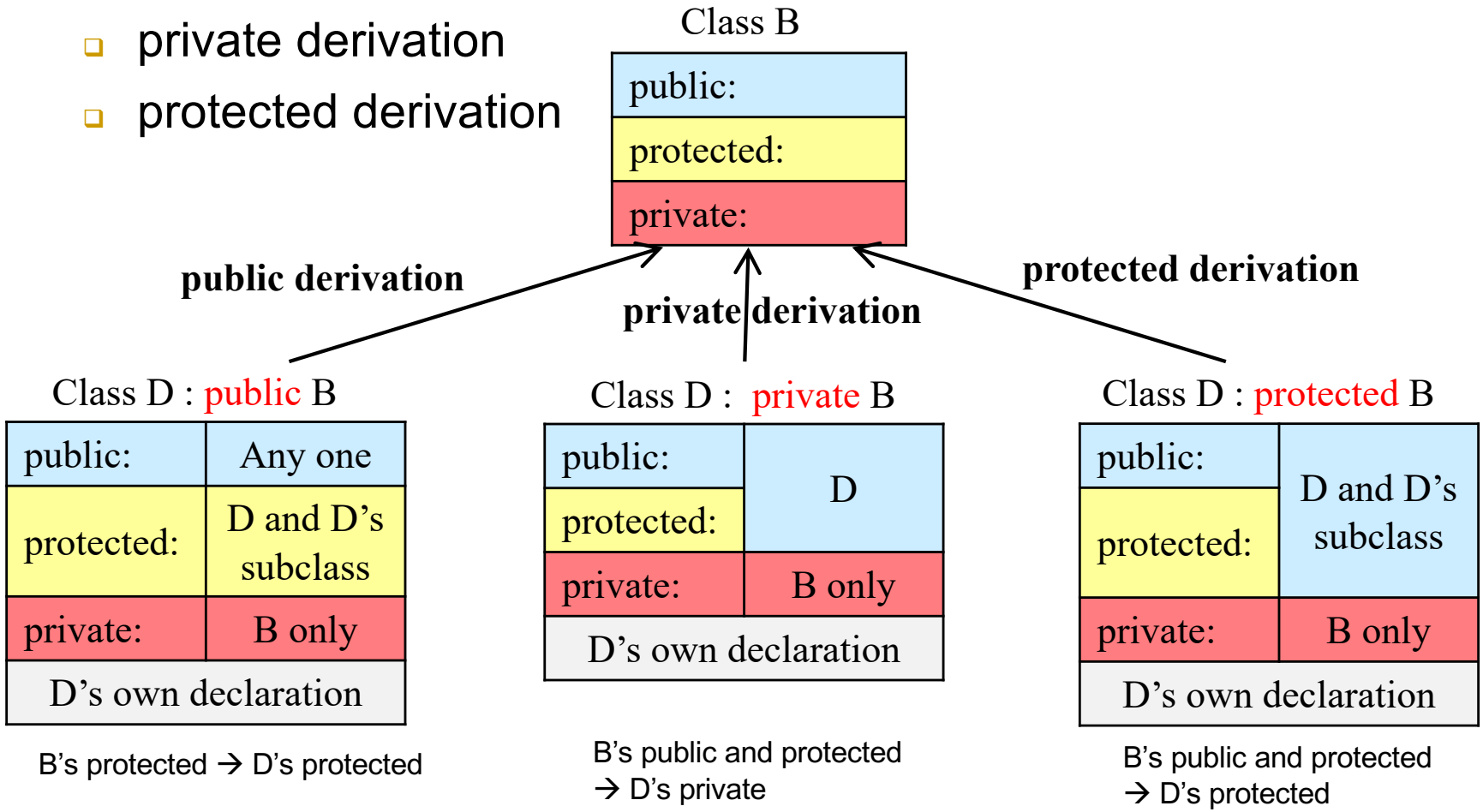



```
class Student : public Person {
    int studentID, attendance;
    char grade;
    ...
};
```

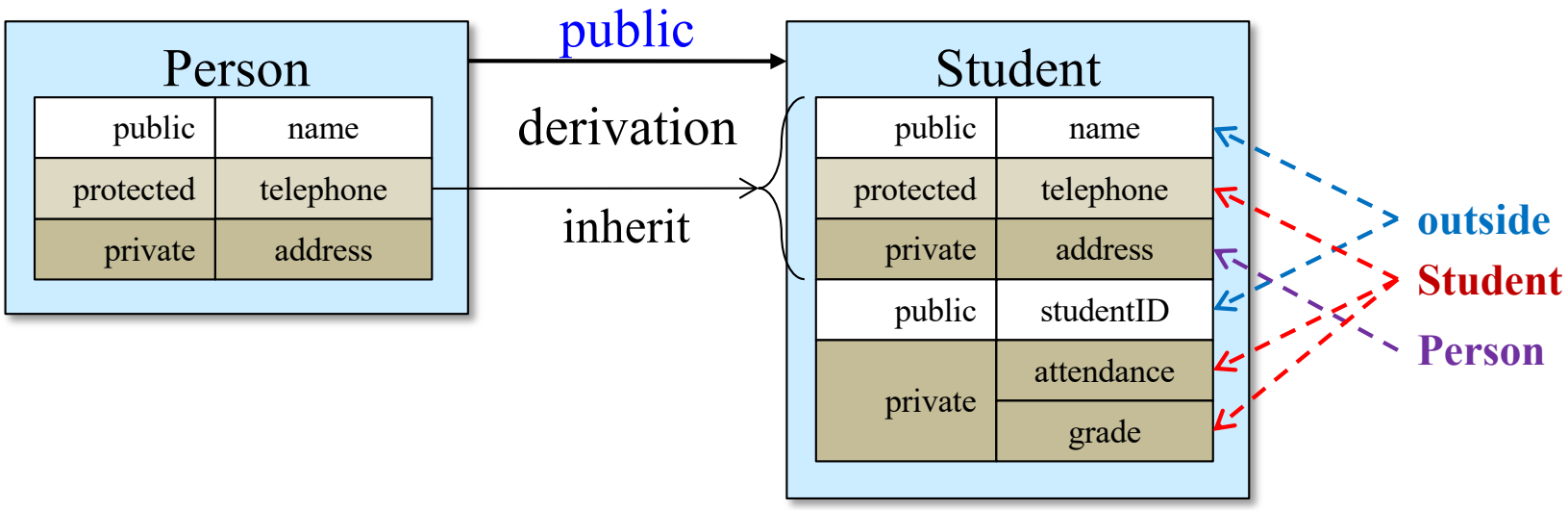
```
class Teacher : public Person {
    char *office, *officeHour;
    ...
};
```

Access to Base Classes

- Access control of a base class
  - public derivation
  - private derivation
  - protected derivation



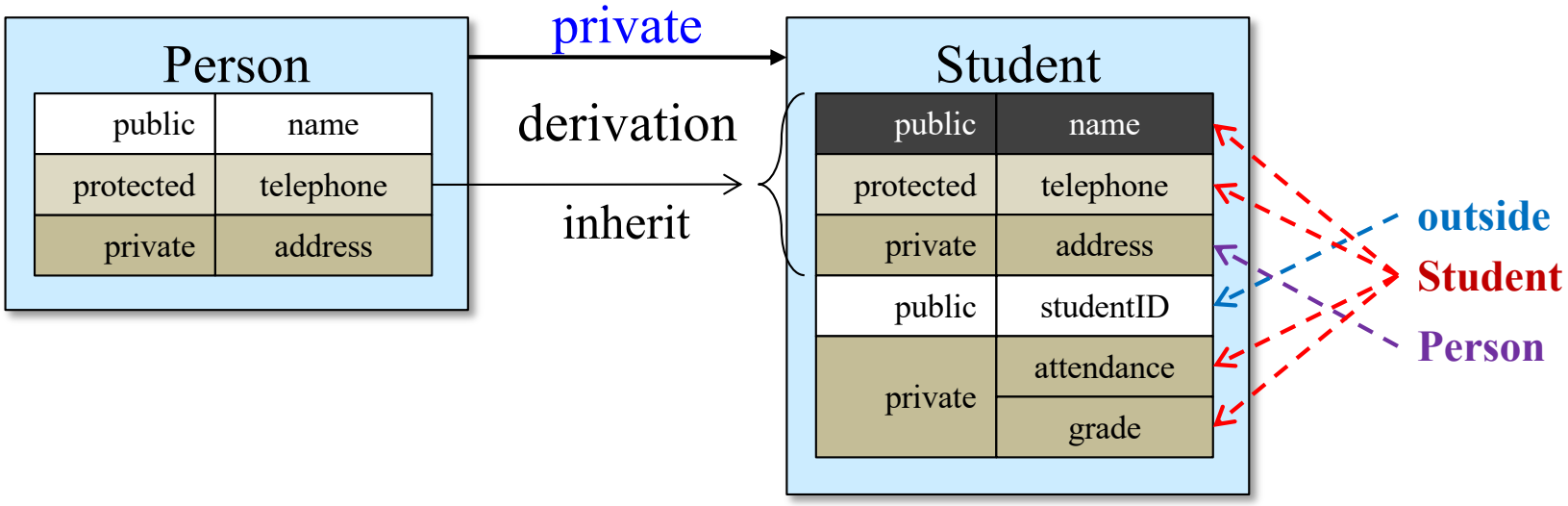
Public Derivation




```
class Person {
 public:
   char *name;
 protected:
   char *telephone;
 private:
   char *address;
};
```

```
class Student : public Person {
 public:
   int studentID;
 private:
   int attendance;
   char grade;
};
```

Private Derivation




```
class Person {
 public:
   char *name;
 protected:
   char *telephone;
 private:
   char *address;
};
```

```
class Student : private Person {
 public:
   int studentID;
 private:
   int attendance;
   char grade;
};
```

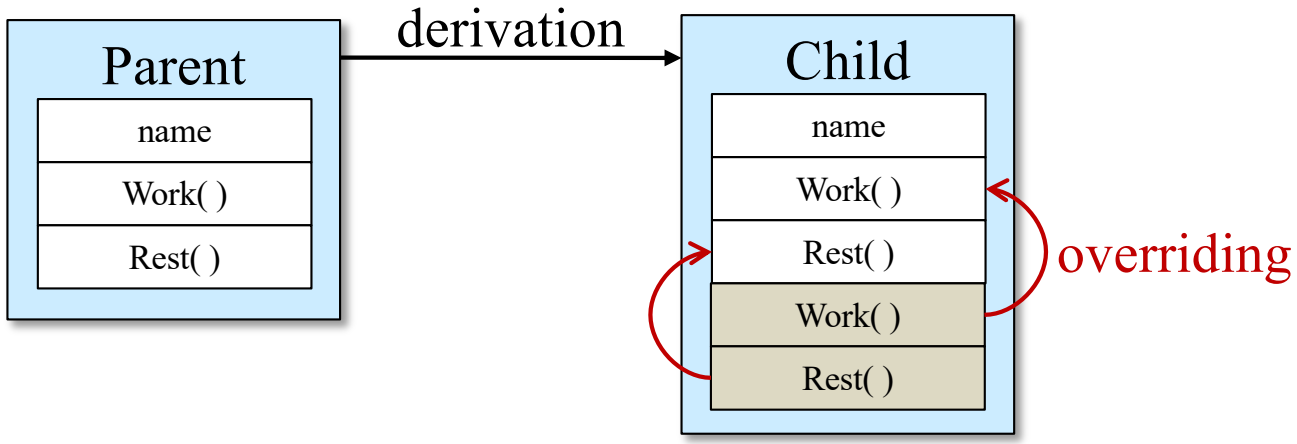
Example: Public Derivation

```
#include<iostream>
using namespace std;
class Parent {
  char *_lastname;
public:
  char *_name;
  char* lastname() { return _lastname; }
  char* name() { return _name; }
  Parent(char *name = "",
         char *lastname = "");
  ~Parent() { delete _name, _lastname; }
};

Parent::Parent(char *name, char *lastname) {
  _name = new char[strlen(name)+1];
    strcpy(_name, name);
  _lastname = new
            char[strlen(lastname)+1];
    strcpy(_lastname, lastname);
}
```

```
class Child : public Parent {
public:
  Child(char *name = "", char *lastname = "");
};

Child::Child(char *name, char *lastname) :
Parent(name, lastname)
{}

int main() {
  Child myRecord("GivenName", "FamilyName");
  cout << "Name : " << myRecord._name << endl;
  cout << "Last name : " << myRecord._lastname() << endl;

  return 0;
}
```

```
Name : GivenName
Last name : FamilyName
```

Overriding: From Subclass to Superclass




```
class Parent {
   ...
public:
   void Work ( ) { ... }
   void Rest ( ) { ... }
};
```

overriding

```
class Child : public Parent{
   ...
public:
   void Work ( ) { ... }
   void Rest ( ) { ... }
};
```

Example: Overriding (1/2)

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

class Parent {
public:
  void print( ) {
    cout << "I'm your father." << endl;
  }
};

class Child : public Parent {
public:
  void print( ) {
    cout << "I'm a Child class." << endl;
  }
};
```

overriding

```
int main() {
  Child child;
  child.print( );
  return 0;
}
```

```
result>
I'm a Child class.
```

Example: Overriding (2/2)

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

class Parent {
public:
  void print( ) {
    cout << "I'm the Parent class." << endl;
  }
};

class Child : public Parent {
public:
  void print(int i = 1) {
    for (int j = 0; j < i; j++)
      cout << "I'm a Child class." << endl;
  }
};
```


overriding

```
int main() {
  Child child;
  child.print( );
  child.print(3);
  return 0;
}
```

result>

```
I'm a Child class.
I'm a Child class.
I'm a Child class.
I'm a Child class.
```

Call Overridden Functions

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

class Parent {
public:
  void print( ) {
    cout << "I'm the Parent class."
<< endl;
  }
};

class Child : public Parent {
public:
  void print( ) {
    cout << "I'm a Child class." << endl;
  }
};
```

overriding

```
int main() {
  Child child;
  child.print( );
  child.Parent::print( );
  return 0;
}
```

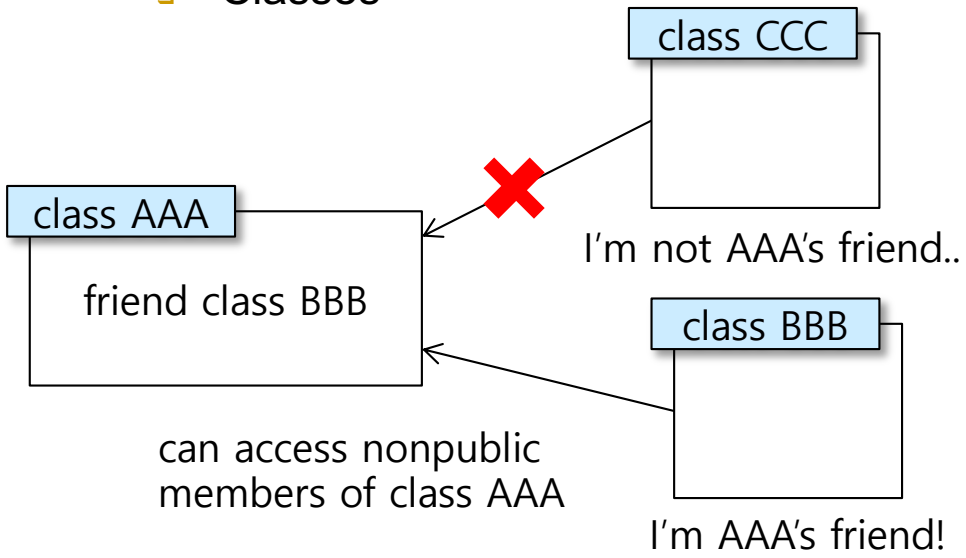
result>

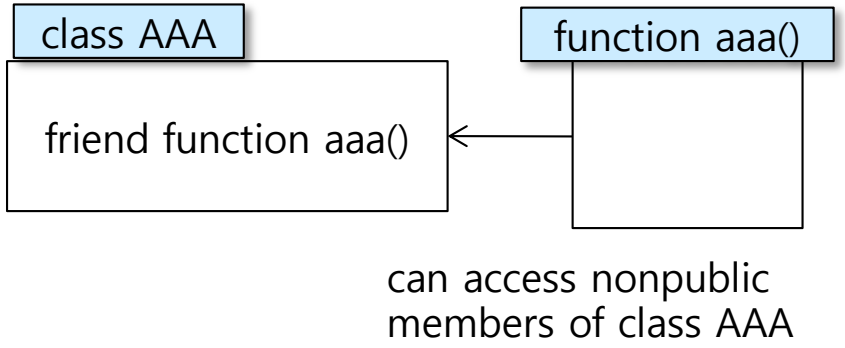
I'm a Child class.

I'm the Parent class.

Friends to a Class

- In some cases, information-hiding is too prohibitive.
  - Only public members of a class are accessible by non-members of the class
- "friend" keyword
  - To give nonmembers of a class access to the nonpublic members of the class
- Friend
  - Functions
  - Classes





Example: Friend Functions

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

class point {
   int x, y;
 public:
   point(int a = 0, int b = 0);
   void print();

   friend void set(point &pt, int a, int b);
};

point::point(int a, int b) {
  x = a; y = b;
}
```

not member function, but friend function

```
void point::print() {
  cout << x << ", " << y << endl;
}
```

call-by-reference

```
void set(point &pt, int a, int b) {
  pt.x = a; pt.y = b;
}

int main() {
  point p(1, 1);
  p.print();
  set(p, 2, 2);
  p.print();

  return 0;
}
```

not "p.set( );"

result>

1, 1

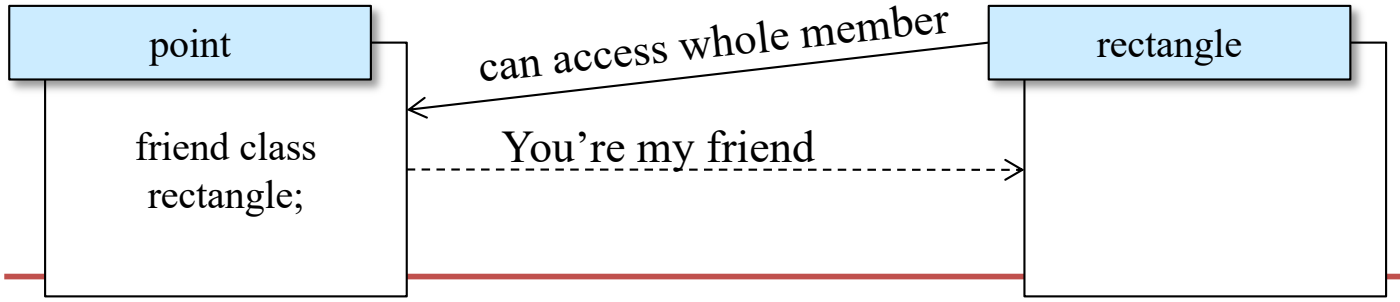
2, 2

Friend Class

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

class point {
   int x, y;
   friend class rectangle;
 public:
   void set(int a, int b);
};

void point::set(int a, int b) {
  x = a; y = b;
}
```

```
class rectangle {
   point leftTop, rightBottom;
 public:
   void setLT(point pt);
   void setRB(point pt);
   void print();
};

void rectangle::setLT(point pt) {
  leftTop.set(pt.x, pt.y);
}

void rectangle::setRB(point pt) {
  rightBottom.set(pt.x, pt.y);
}
```

```
void rectangle::print() {
  cout << "LT:" << leftTop.x;
  cout << ","      << leftTop.y << endl;
  cout << "RB:" << rightBottom.x;
  cout << ","      << rightBottom.y << endl;
}

int main() {
  rectangle sq;
  point lt, rb;
  lt.set(1, 1); sq.setLT(lt);
  rb.set(9, 9); sq.setRB(rb);
  sq.print();
  return 0;
}
```



Result >

LT: 1, 1

RB: 9, 9

Template: Function and Class

Function Template (1)

```
int integerMin(int a, int b)                    // returns the minimum of a and b
  { return (a < b ? a : b); }
```

- Useful, but what about min of two doubles?
  - C-style answer: double doubleMin(double a, double b)

- Function template is a mechanism that enables this
  - produces a generic function for an arbitrary type T.

```
template <typename T>
T genericMin(T a, T b) {                        // returns the minimum of a and b
  return (a < b ? a : b);
}
```

Function Template (2)

```
template <typename T>
T genericMin(T a, T b) {                    // returns the minimum of a and b
  return (a < b ? a : b);
}
```

```
cout << genericMin(3, 4) << ' '      // = genericMin<int>(3,4)
  << genericMin(1.1, 3.1) << ' '     // = genericMin<double>(1.1, 3.1)
  << genericMin('t', 'g') << endl;   // = genericMin<char>('t','g')
```

Function Overloading vs. Function Template

- Function overloading
  - Same function name, but different function prototypes
  - These functions do not have to have the same code
  - Does not help in code reuse, but helps in having a consistent name.
- Function template
  - Same code piece, which applies to only different types.

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

int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -
123;
    cout << " = " << abs(-123) <<
endl;
    cout << "absolute value of " << -
1.23;
    cout << " = " << abs(-1.23) <<
endl;
}
```

Class Template (1)

- We can also define a generic template class
- Example: BasicVector
  - Stores a vector of elements
  - Can access i-th element using [] just like an array

```
template <typename T>
class BasicVector {                    // a simple vector class
public:
  BasicVector(int capac = 10);         // constructor
  T& operator[](int i)                 // access element at index i
    { return a[i]; }
  // ... other public members omitted
private:
  T* a;                                // array storing the elements
  int capacity;                        // length of array a
};
```

Class Template (2)

- BasicVector
  - Constructor code?

```
template <typename T>                    // constructor
BasicVector<T>::BasicVector(int capac) {
  capacity = capac;
  a = new T[capacity];                   // allocate array storage
}
```

- How to use?

```
BasicVector<int>     iv(5);     iv[3] = 8;
BasicVector<double>  dv(20);    dv[14] = 2.5;
BasicVector<string>  sv(10);    sv[7] = "hello";
```

Class Template (3)

- The actual argument in the instantiation of a class template can itself be a templated type
- Example: Twodimensional array of int

```
BasicVector<BasicVector<int> > xv(5); // a vector of vectors
// ...
xv[2][8] = 15;
```

- BasicVector consisting of 5 elements, each of which is a BasicVector consisting of 10 integers
  - In other words, 5 by 10 matrix

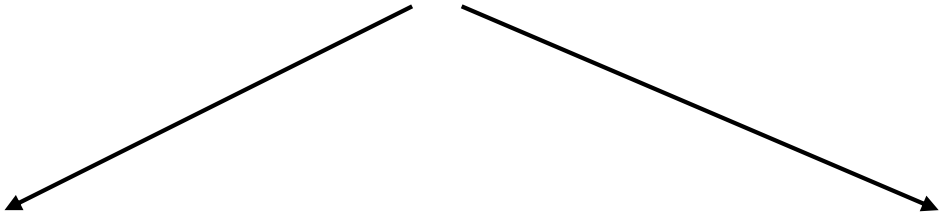
Exceptions

Exceptions: Intro

- Exception
  - Unexpected event, e.g., divide by zero
  - Can be user-defined, e.g., input of studentID > 1000
  - In C++, exception is said to be "thrown"
  - A thrown exception is said to be "caught" by other code (exception handler)
  - In C, we often check the value of a variable or the return value of a function, and if… else… handles exceptions
    - Dirty, inconvenient, hard to read

Exception: Also a class

```
class MathException {                          // generic math exception
public:
  MathException(const string& err)             // constructor
    : errMsg(err) { }
  string getError() { return errMsg; }         // access error message
private:
  string errMsg;                               // error message
};
```



```
class ZeroDivide : public MathException {
public:
  ZeroDivide(const string& err)
  : MathException(err) { }
};
```

```
class NegativeRoot : public MathException {
public:
  NegativeRoot(const string& err)
  : MathException(err) { }
};
```

Exception: Throwing and Catching

```
try {
  // ... application computations
  if (divisor == 0)                          // attempt to divide by 0?
    throw ZeroDivide("Divide by zero in Module X");
}
catch (ZeroDivide& zde) {
  // handle division by zero
}
catch (MathException& me) {
  // handle any math exception other than division by zero
}
```

ZeroDivide “is a” MathException? Yes

Exception Example (1)

```
#include <iostream>
using namespace std;
double division(int a, int b){
     if( b == 0 ) {
         throw "Division by zero condition!";
     }
     return (a/b);
}

int main () {
     int x = 50;   int y = 0;   double z = 0;
     try {
           z = division(x, y);
           cout << z << endl;
     } catch (const char* msg) {
           cerr << msg << endl;
     }
     return 0;
}
```

Exception Specification

- In declaring a function, we should also specify the exceptions it might throw
  - Lets users know what to expect

```
void calculator() throw(ZeroDivide, NegativeRoot) {
    // function body ...
}
```

The function calculator (and any other functions it calls) can throw two exceptions or exceptions derived from these types

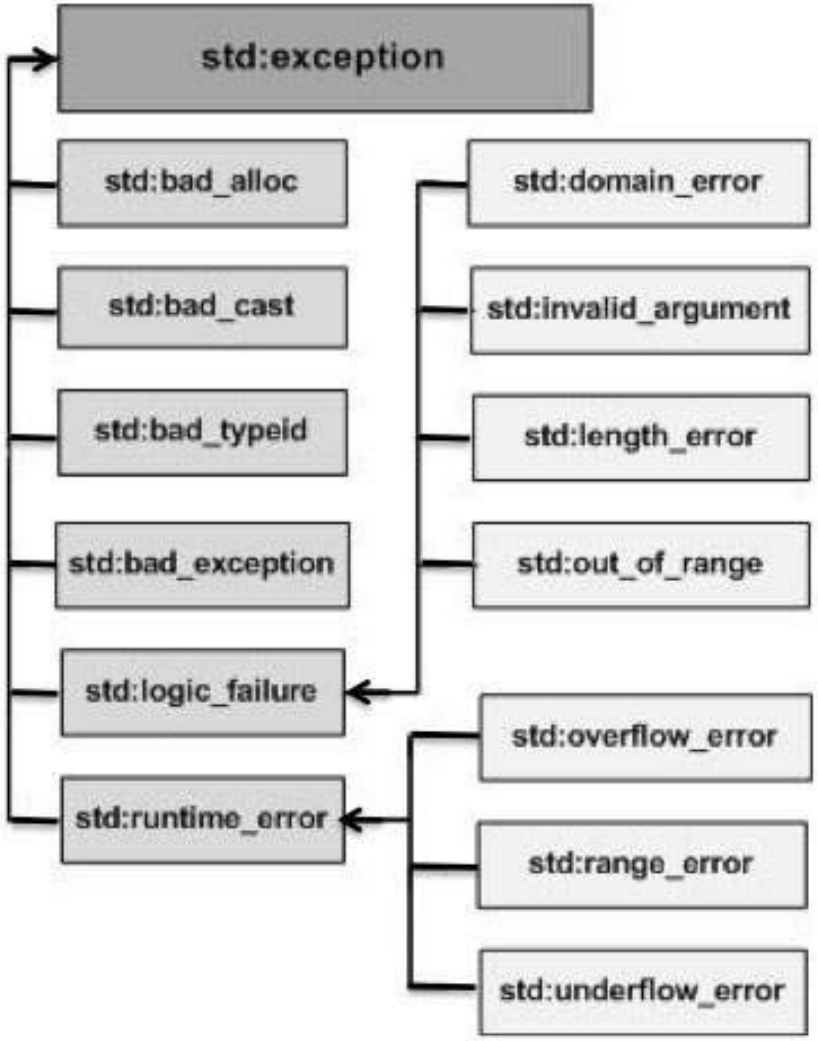
- Exceptions can be "passed through"

```
void getReadyForClass() throw(ShoppingListTooSmallException,
                              OutOfMoneyException) {
   goShopping();  // I don't have to try or catch the exceptions
                  // which goShopping() might throw because
                  // getReadyForClass() will just pass these along.
   makeCookiesForTA();
}
```

Exception: Any Exception and No Exception

```
void func1();                    // can throw any exception
void func2() throw();            // can throw no exceptions
```

C++ Standard Exceptions



Exception Example (2)

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

class MyException : public exception {
    const char * what () const throw (){
        return "C++ Exception";
    }
};
int main()
{
    try {
       throw MyException();
    }catch(MyException& e){
       std::cout << "MyException caught" << std::endl;
       std::cout << e.what() << std::endl;
    } catch(std::exception& e){
       //Other errors
    }
}
```

File I/O

File I/O

- Declare the stream to be processed:

```
#include <fstream>
ifstream ins;// input stream
ofstream outs;// output stream
```

- Need to open the files

```
ins.open(inFile);
outs.open(outFile);
```

Files

- `#define` associates the name of the stream with the actual file name
- `fail()` function - returns nonzero if file fails to open
- Some more functions
  - `get`, `put`, `close` and `eof`
  - ...

Wrap Up

- You may not have a big problem in using the C++ programming language
- You may not have a big problem in doing the homework and project assignments
- However,
  - Be ready to debug your program
  - Be ready to search more things in Web
  - Be ready to meet "compilation errors"
- The online C++ Tutorials would be useful.
  - https://cplusplus.com/doc/tutorial/