

**COP 3503**

**Programming Fundamentals  
for CIS Majors II**

Jonathan C.L. Kavalan, Ph.D.  
CISE Department  
University of Florida

# ACM/IEEE-CS Computer Science Curricula (CS2013)

- (1) AL – Algorithms and Complexity
- (2) AR – Architecture and Organization
- (3) CN – Computational Science
- (4) DS – Discrete Structures
- (5) GV – Graphics and Visualization
- (6) HCI – Human-Computer Interaction
- (7) IAS – Information Assurance and Security

# ACM/IEEE-CS Computer Science Curricula (CS2013)

(8) IM – Information Management

(9) IS – Intelligent Systems

(10) NC – Networking and Communications

(11) OS – Operating Systems

(12) PBD – Platform-based Development

(13) PD – Parallel and Distributed Computing

(14) PL – Programming Languages

# ACM/IEEE-CS Computer Science Curricula (CS2013)

(15) SDF – Software Development Fundamentals

(16) SE – Software Engineering

(17) SF – System Fundamentals

(18) SP – Social Issues and Professional Practice

# About the Course

This course serves as an introduction to the  
“fundamentals” of computer science

based on C++ (and the underlying C)

We will examine the programming paradigm  
known as “object-oriented programming.”

data structures, algorithms, systems and  
applications

# What is Computer Science?

Computer science is ***not*** just coding.

Instead, computer science is about the underlying **principles** that allow for the design and implementation of **efficient** programs.

These issues exist, no matter which programming language is used.

# What is Computer Science?

There are a lot of different ways to sort objects, or what we like to call *data*.

One of the more intuitive methods is known as an *insertion sort*.

You keep the sorted part of your data separate from the unsorted part, placing each newly sorted object into the correct location into the currently sorted section.

# What is Computer Science?

There are a lot of different ways to sort objects, or what we like to call *data*.

Another sorting method is called *Quicksort*.

This method works by picking an approximate “median” for the data you want to sort, then throwing all the rest of the data to either the “high” side or the “low” side.

Once that’s complete, you sort the “high” side and the “low” side in the same manner.



# What is Computer Science?

There are a lot of different ways to sort objects, or what we like to call *data*.

We will likely examine these in much greater detail later.

Note that both techniques accomplish the same goal in different ways.

# Why Study Computer Science?

In computer science, it's encouraged to learn many different ways of handling the same problem.

At the same time, it teaches you how to evaluate the pros and cons of each.

Some techniques are superior to others for special circumstances.

# Origins of C (and C++)

- C is a by-product of UNIX, developed at Bell Lab by Ken Thompson, Dennis Ritchie and others.
- Thompson designed a small language named “B”.
- “B” was based on BCPL, a system programming language developed in the mid-1960s.

# Origins of C (2)

- By 1971, Ritchie began to develop an extended version of “B” language.
- He called the new language “NB” (New B) at first.
- As the NB language began to diverge more from B, he changed its name to C.
- The C language was stable enough by 1973 that UNIX could be re-written in C.

# The Year of 1973

- PC/Internet are not invented (totally) yet!!
- Computers mostly mean “super computers”, “main-frame computers” and “minicomputers” (e.g., DEC PDP-7)



# Standardizations of C

*C Became popular during the 1980s, both for UNIX programming and for developing applications for personal computers.*

- *K&R C*: Described in Kernighan and Ritchie, *The C Programming Language*, Prentice-Hall, 1978; The de-facto standard
- *ANSI C (ISO C, C89)*: ANSI standard X3.159-1989 (completed in 1988; formally approved in December 1989); International standard ISO/IEC 9899:1990
- *C99*: International standard ISO/IEC 9899:1999; Incorporates changes from Amendment 1 (1995)

# Here came the C++

Bjarne Stroustrup, a Danish and British trained computer scientist, began his work (in AT&T) on **C++'s** predecessor "C with Classes" in 1979.

The motivation for creating a new language originated from Stroustrup's experience in programming for his Ph.D. thesis.

# C-based Languages

- **C++** includes all the features of C, but adds classes and other features to support object-oriented programming
  - 1983: C-with-classes redesigned into C++
  - 1985: C++ compilers made available
  - 1989: ANSI/ISO C++ standardization starts
  - 1999: ANSI/ISO C++ standard approved
  - Hence the c99 compiler you will use later



# Why C++ and OOP

Note that, ideally, a sorting method (i.e., a “way” of doing sorting) should work *regardless* of whatever is being sorted.

Words; Numbers; Cards; Dates; Times;

Think about your experiences on Amazon, eBay, Expedia about the sorting preferences (e.g., prices, auction/connection time)

# The need for abstraction

Abstraction involves determining the “least-common denominator” held in common by sets of data/object or functionality/method within a program.

- can be re-used
- can be shared
- can be extended

# Strengths of C/C++

**Portable:** usually does not requires a significant change when the C/C++ programs are ported to different machines (ranging from super-computer to embedded systems)

- C compilers are small and easy to be included in any application development environment

# Strengths of C/C++

- **Powerful:** has a large collections of data types/classes and operators/methods.
- **Flexible:** C imposes very few restrictions on the use of its features
- **Standard Library:** almost universal

# **Strengths of C/C++**

**Integrated well with all existing platforms:** including Microsoft C/C++, and popular UNIX variant (known as Linux and Android) and MacOS (and hence the iOS)

# Programming is never easy

- Some mistakes may not be detected by the compilers
- “No errors, No warnings” from compilers is just half-way done to finish the programming
- Run-time errors are hard to debug
- Infinite loop and program crash are possible outcomes

# Effective Use of C/C++

- Use software tools (debuggers, etc.) to make programs more reliable.
- Take advantages of existing code libraries.
- Adopt a sensible set of coding conventions.
- Avoid “tricks” and overly-complex codes.
- Stick to the standards (to maximize the portability).

# Software Engineering

The term *software engineering* refers to the study of software development on large scales.

Few programs these days are written by lone, individual programmers.

Instead, programs are often written by large teams who must coordinate their efforts.



# Software Engineering

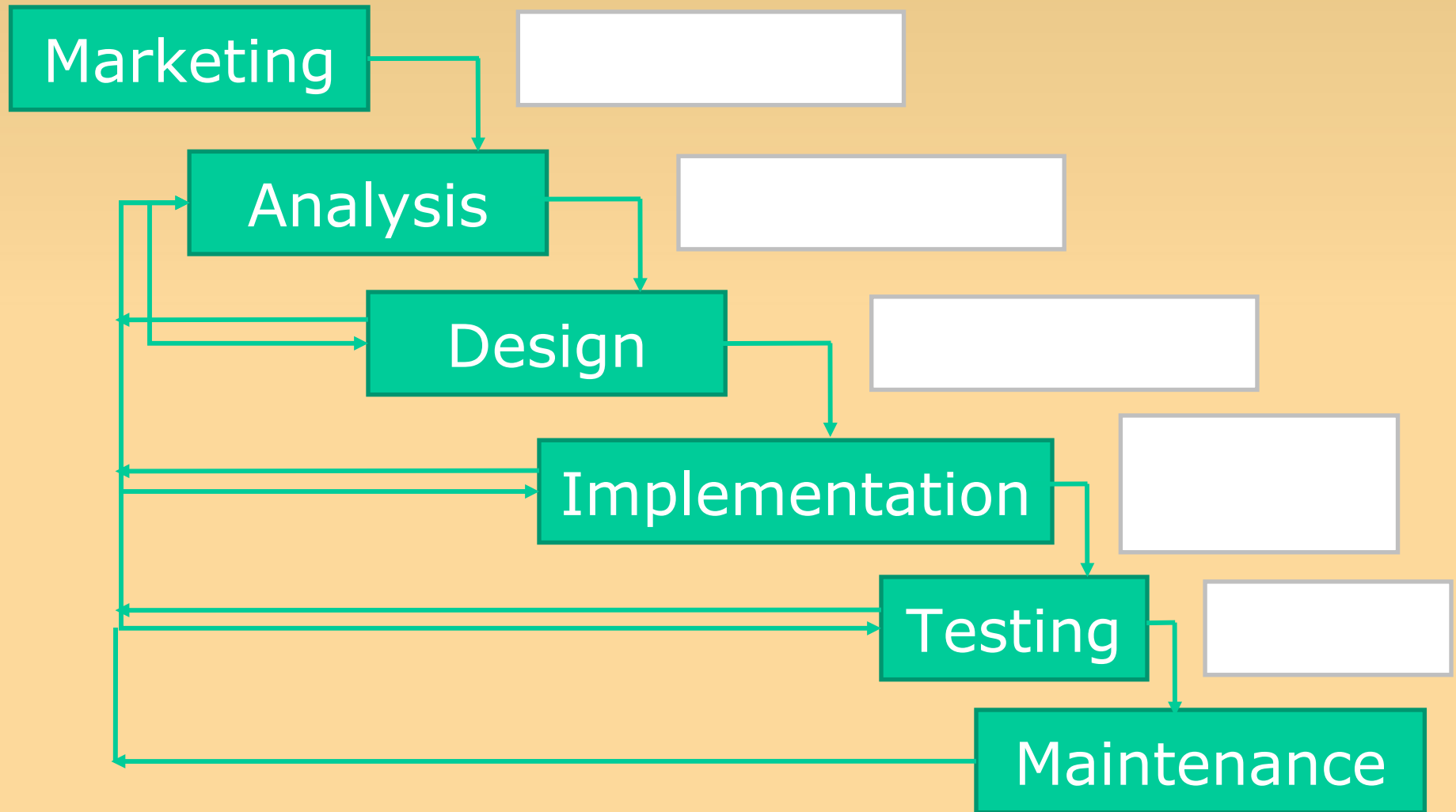
The way you approach *program design* matters.

Programs are quite often large, complex structures.

Many modern video games have teams of over **100** people working on them for over a year.

If the program is being written from scratch, make that **five** years.

# The Waterfall Model



# Software Engineering

The way you approach *program design* matters.

For such large projects, it is important to plan your program's structure largely in advance.

It is very common for one programmer to use code that another one wrote, sight unseen.

Any team member must know what will be available to each component, as well as what form any important data will be in.

# Software Engineering

The way you approach *program design* matters.

It is important to constantly test your program components.

The longer you wait to find errors – and you ***will*** have errors – the harder it is to trace their source.

Time is money – the more time you waste on errors, the less time left for adding the features that will make your product shine.

# Testing: Why?

The first flight of the ESA Ariane 5: \$1 billion in damages.

At its core, the error was triggered by an arithmetic overflow exception... which was the result of improper code reuse.

The rocket self-destructed a mere 37 seconds after launch.

# Testing: Why?

Other known software catastrophes:

- 1999 - Mars Climate Orbiter - wrong units

- 1999 - Mars Polar Lander – engine cutoff too early, SW bug

- 2004 - Mars Rover – too many files opened

- 2006 - Mars Global Surveyor – battery failure due to SW error

# Necessary Skills

Prior programming experience

Not necessarily in Java or C/C++, but it helps greatly.

Willingness to work in groups

Ability to tolerate frustration

Rule #1 of programming: You will make errors.

Ability to see both the big picture and local, detailed specifics.

# What we hope

Eventually, you will have the ability to ...

- (1) see things abstractly and apply OOP
- (2) balance both performance and functional requirements of the software systems
- (3) continue the interest on computer science and join the software industry !!
  - 77 of world's top 100 software companies are head-quartered in USA (IDC Data 2009) ...



# **Object Orientation**

Why We Do It

# Typical “Early” Code

---

- Often, when programming is taught, the majority of the focus is on learning to use basic data types, programming logic, and functions.
- Much of the program is often thrown into one main method, which might call one or two other functions.

# Typical “Early” Code

---

- It's readily apparent that any program has (at least) two fundamental component categories that the user must define and manage.
  - Data – the information received, output, and maintained by the program
  - Functions/Methods – the programming logic that manipulates data as needed.



# Typical “Early” Data

---

- For (nearly) any program to serve a useful purpose, it will need to meaningfully store and use some type of data.
  - What are some of the basic data types that you’ve used to this point in programming?

# Typical “Early” Data

---

- int; long; char; float; double; bool; string; and array [];
  - Combination of these types can represent more complex types
  - Two basic types: integers and float
  - Integers are whole numbers
  - Integers can be signed or unsigned

# C++ vs. C languages

---

```
#include <iostream>
```

```
using namespace std;
```

```
class cl { int i; // private by default  
public: int get_i(); int put_i(int j); }
```

```
int cl::get_i() {return i;}  
int cl::put_i(int j) {i = j;}
```

```
int main()  
{  
    cl s;  
    s.put_i(10);  
    cout << s.get_i() << endl;  
    return 0;  
}
```

```
#include <iostream>
```

```
int main()  
{  
    int i, j=10;  
  
    i=j;  
    printf ("%d \n");  
    return 0;  
}
```



# A Rough Exercise

---

- Suppose we wanted to write a program for playing a card game of some sort.
  - Like with Hearts or Spades, the full deck is dealt to four players.
- Disregarding the rules of the game... how would we *manage the cards*?

# A Rough Exercise

---

- Cards are important information / data to keep track of for a card game.
- What manipulates cards, and how would this have to be coded?
  - Shuffling
  - Dealing
  - Each player has a separate hand...



# A Rough Exercise

---

- The following program illustrates both two-dimensional arrays and constant arrays.
- The program deals a random hand from a standard deck of playing cards.
- Each card in a standard deck has a *suit* (clubs, diamonds, hearts, or spades) and a *rank* (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace).

# A Rough Exercise

---

- The user will specify how many cards should be in the hand:

Enter number of cards in hand: 5

Your hand: 7c 2s 5d as 2h

- Problems to be solved:

- \* How do we pick cards randomly from the deck?

- \* How do we avoid picking the same card twice?

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13

int main(void)
{
    bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
    int num_cards, rank, suit;
    const char rank_code[] =
{'2','3','4','5','6','7','8',

'9','t','j','q','k','a'};
    const char suit_code[] = {'c','d','h','s'};
```



```
srand((unsigned) time(NULL));

printf("Enter number of cards in hand: ");
scanf("%d", &num_cards);

printf("Your hand:");
while (num_cards > 0) {
    suit = rand() % NUM_SUITS;    /* picks a random suit */
    rank = rand() % NUM_RANKS;    /* picks a random rank */
    if (!in_hand[suit][rank]) {
        in_hand[suit][rank] = true;
        num_cards--;
        printf(" %c%c", rank_code[rank], suit_code[suit]);
    }
}
printf("\n");

return 0;
}
```

# A Rough Exercise

---

- Now, consider the complexity of what we've put forth.
- There were many servers for competitive card-game playing
  - Imagine having to code like this for *thousands of simultaneous games*
  - How would *that* work?

# Motivation

---

- One of the most evident problems that arises in novice programming is a lack of scalability.
  - This is often fine for initial learning – simplicity leaves much less room for confusion.
  - The more interesting question – why is the typical novice programming style not scalable?



# Motivation

---

- Two key things to note in novice-style coding:
  - Note how we're organizing data.
  - Note how we're accessing data in the various functions of our proposed programs.

# Motivation

---

- Two key things to note in novice-style coding:
  - Note how we're organizing data.
  - How is the data grouped together?
  - Do these groupings help clarify things?
  - Are we limited to a fixed size/count of data?
  - Note how we're accessing data...



# Motivation

---

- Two key things to note in novice-style coding:
  - Note how we're organizing data.
  - Note how we're accessing data...
  - Do we have to copy-paste code to multiple points of our program, with slight modifications each time?
  - Do we have to assume all code copies operate perfectly for any of our code to work correctly?

# Object Orientation

---

- The coding style of object-orientation provides one popular solution to these concerns.
  - Data are organized to represent distinct *objects* of the scenario being modeled.
  - The card deck
  - Each player's hand
  - Each individual card
  - This is done by defining *custom* data types.

# Object Orientation

---

- The coding style of object-orientation provides one popular solution to these concerns.
  - When these conceptual “objects” of the program are modeled as custom data types, we may then manipulate them through functions designed to operate upon those custom types.
  - `CardHand[] CardDeck::dealHands`
    - `(int numHands, int numCards)`



# Object Orientation

---

- The coding style of object-orientation provides one popular solution to these concerns.
  - Additionally, we may provide some functionality that will be seen as *inherent* to these custom data types.
  - These allow accessing and manipulating attributes of our program's objects.
  - `void CardDeck::shuffle();`

# Object Orientation

---

- We don't think about it like this, but such functions already exist for our basic data types...

- $1 + 1$

- $3.14159 * 2.71828$

- From Java:

- “Hello ” + “World”

- `System.out.println(“The answer is ” + 42);`

- As written, these do not translate directly into C++.

- In C++, `cout << “The answer is ” << 42 << endl;`

# Object Orientation

---

- Programming then becomes about recognizing the distinct “objects” that need to exist within the system and coding them appropriately.
  - This includes needed interactions among objects.



# **C++ Code Structure**

Cooperating with the Compiler

# C / C++ Compilation

---

- In Java (and many other modern languages), the compiler is designed to make multiple passes over code files during compilation.
  - In doing this, the compiler first finds all objects, variables, and functions of interest that are available before beginning the actual computation.



# C / C++ Compilation

---

- In C++, you are required to manually “declare” any object, variable, or function *within a code file* **before** using it.

- Note: “declaring” vs “defining.”

- “declare” – “function X exists.”

- “define” – “this is what function X does.”

- If you try to use something before it’s declared, a compiler error will result.

# C / C++ Compilation

---

- To simplify the process of declaring relevant code objects, C++ has two core file types.
  - Header files: `"*.h"`
  - Contains relevant declarations
  - Source files: `"*.cpp"` (`"*.c"` in C.)
  - Contains code definitions
  - Source ***and*** header files then `#include` other header files with needed definitions.

# C++ Resources

---

- Like Java, C++ has a substantial amount of pre-coded resources for use in programs.
  - This being said, Java's built-in collection is far more extensive than C++'s.
  - These are also utilized by use of `#include`, as opposed to Java's `import`.
  - However, built-in resources are included through `<angle brackets>` rather than "quotes."



# C++ Resources

---

- Very common imports:

- `<string>`

- Includes the `std::string` class, a C++ counterpart to Java's `String`. This is *not* a fundamental type in C++.

- `<iostream>`

- Includes the `std::cout` and `std::cin` output and input streams.

- As used in class, these are the console output and console input structures, like `System.out` and `System.in` from Java.

# **Object Orientation**

A Crash Course Intro

# What is an Object?

---

- An object, in the context of object-oriented programming, is the association of a *state* with a set of *behaviors*.
  - State: its fields, or “member variables”
  - Behaviors: its associated methods, or “member functions.”



# A First Object

---

```
using namespace std;
```

```
// a very basic C++ object
```

```
class Person
```

```
{
```

```
    public:
```

```
        string name;
```

```
        int age;
```

```
}
```

# A First Object

---

```
using namespace std;  
  
// a very basic C++ object  
class Person  
{  
    public:  
        string name;  
        int age;  
}
```

Note – this is **not** a properly-designed class according to object-orientation principles.



# Analysis

---

# Analysis

---

- Object-orientation is all about recognizing the different “actors” at work in the system being modeled.
  - First, the different data available within the system are organized appropriately.
  - Secondly, functionalities relating to the state of data and its management are bound to that data.

# Analysis

---

- Object-orientation is all about recognizing the different “actors” at work in the system being modeled.
  - These objects (“actors”) may then interact with other objects through well-formed, bounded relationships.



# Analysis

---

- For now, let's examine how we should look at individual objects – the “actors” in a program.
  - Each object should be composed of a set of related data that represents some logical unit within the program.
  - In this case, this would be our “Person” class.

# Analysis

---

1.Inputs: what does our object need in order to be properly formed?

- Both from outside, and for internal representation?

2.Outputs: what parts of our object are needed by the outside world?

- What might some other part of the program request from it?

# Analysis

---

1. Constraints: should our object have limitations imposed on it, beyond those implied by the language we're using?

- Some of our internal state variables (fields) may allow values which make no sense in the context of what our object represents.



# Analysis

---

1. Assumptions: Are we assuming something in our construction of the class which might have to change later?

- We wish to minimize these (in the long run at least) as much as possible.

# A First Object

---

```
// a very basic C++ object
class Person
{
    public:
        string name;
        int age;
}
```

- What is bad about the design of our current “Person” class?



# Encapsulation

---

- *Encapsulation* refers to the idea that an object should protect and manage its own state information.
  - In a way, each object should behave like its own entity.
  - Data security is enforced by the object definition itself.
  - This allows a programmer to make ensure that the data being represented is always in a consistent form.

# Encapsulation

---

- Generally speaking, objects should never make their fields **public**.
  - A **public** field can be accessed *and* modified at any time from code that has a reference to the object, which can invalidate its internal state.

# Encapsulation

---

- Note that encapsulation is motivated by the desire to enforce *constraints* on our object.
  - How can we make sure our object is always in a proper, well-formed state if we can't limit how others modify it?



# Encapsulation

---

- In object-oriented languages, objects may set their fields to be inaccessible outside of the class.
  - To do this, one may use the access modifier `private`.
  - This restricts access to the “`private`” field or method to ***only*** code in the class in which said field or method is defined.

# A First Object

---

```
// a very basic C++ object
class Person
{
    public:
        string name;
        int age;
}
```

- So, instead of marking the fields as public...

# A First Object

---

```
// a very basic C++ object
class Person
{
    private:
        string name;
        int age;
}
```

- We want to mark them as private.



# A First Object

---

```
// a very basic C++ object
class Person
{
    private:
        string name;
        int age;
}
```

- This creates a new problem, though.
  - How can we initialize our object?



# Initialization

---

- By default, when no constructor exists for a class, C++ creates a “default” constructor with no internal code.
  - Note: this “default” constructor will initialize *nothing* within the class.
  - Java’s default constructor acts differently, setting values to zeros and nulls.
- Typically, we will need to create our own constructors to ensure the class is properly initialized.

# A First Object

---

```
// a very basic C++ object
class Person
{
    public:
        Person(string name, int age);

    private:
        string name;
        int age;
}
```

# A First Object

---

```
Person::Person(string name, int age)
{
    this->name = name;
    this->age  = age;
}
```

- Something interesting here: note the use of “->”.
  - “this” is a *pointer*, and it refers to the *instance* of the class upon which the constructor/function has been called.



# Initialization

---

- Once one constructor has been coded for a class, the default constructor no longer exists unless it is manually coded.
- A fully constructed and initialized class object can be called an *instance* of its class.

# A First Object

---

```
// a very basic C++ object
class Person
{
    public:
        Person(string name, int age);

    private:
        string name;
        int age;
}
```

- We still have another problem.
  - How can we actually make use of the class's data?



# Encapsulation

---

- Since we've set the class fields to **private**, it is necessary to implement some way of accessing its information
  - one that does not expose the fields.
  - The solution? *Accessor* methods.

# A First Object

---

```
string Person::getName()  
{  
    return this->name;  
}
```

```
int Person::getAge()  
{  
    return this->age;  
}
```

# A First Object

---

```
string Person::getName()  
{  
    return this->name;  
}
```

- Suppose we had a "Person p". The line "p.getName()" would return the value for "name" from the object represented by "p".



# Encapsulation

---

- First, note that these accessor methods will be set to **public** – otherwise, they won't be of use to code outside of the class.
- Secondly, these methods retrieve the data without allowing it to be changed.
  - In Java**, String's implementation does not allow its internal data to be changed. C++ differs on this point.

# Encapsulation

---

- What if we need to be able to change one or more of the fields of a class instance?
  - The (first) solution: *mutator* methods.
  - These provide an interface through which outside code may *safely* change the object's state.



# A First Object

---

```
void Person::setName(string name)
{
    this->name = name;
}
```

```
void Person::setAge(int age)
{
    this->age = age;
}
```

# Encapsulation

---

- Is this necessarily the correct solution, though?
  - It depends on the purpose for our class.
- Note that we allow both the “name” of our “Person” and his/her “age” to change, freely.

# Encapsulation

---

- Should we allow a “Person” to change his/her name?
  - It does happen in the real world, but for simplicity, let us suppose that we do not wish to allow people to change names.
  - In such a case, we should remove the setName() method.



# Encapsulation

```
void Person::setName(string name)
{
    this->name = name;
}
```

```
void Person::setAge(int age)
{
    this->age = age;
}
```

# Encapsulation

---

- However, we shouldn't stop here. If we wish to make sure that a person may *never* have their name changed, can we make sure that even code from within the class may not change it?
  - Yes: use the `const` keyword.
  - In Java: `"final"`.



# Encapsulation

---

```
class Person
{
    private:
        const string name;
        int age;
}
```

- When a field is marked as `const`, it can only be initialized in a special part of the constructor.

# Encapsulation

---

```
Person::Person(string name, int age)
:name(name)
{
    //this->name = name;
    /* This line would be
       a compiler error! */

    this->age = age;
}
```

# Encapsulation

---

```
Person::Person(string name, int age)
: name(name)
{
    //this->name = name;
    /* This is a comment
    a comment
    this->age = age;
}

```



This is the only valid way to initialize a **const** variable.



# Encapsulation

---

- Should we allow a “Person” to change his/her age?
  - Last time I checked, everyone ages.
  - However, note that a person’s age *cannot* change freely.
  - Nobody ages in reverse.
  - A person can only add one year to their age, well, every year.



# Encapsulation

```
void Person::setName(String name)
{
    this->name = name;
}
```

```
void Person::setAge(int age)
{
    this->age = age;
}
```

# Encapsulation

---

```
void Person::haveABirthday()  
{  
    this->age++;  
}
```

# Encapsulation

---

- At first, encapsulation may seem to be unnecessary.
  - It does add extra effort to using values that you *could* just directly access instead.
  - However, someone else might not know how to properly treat your object and may mess it up if you don't encapsulate.



# Encapsulation

---

- There are other benefits to encapsulation.
  - What if you later realize there's an even *better* way to implement your class?
  - You can provide the same methods for accessing object data while changing its internals as needed.



# Encapsulation

---

- Is our current implementation of age “the best”?
  - A possible alternative: track birthdays instead!
  - Birthdays only come once a year, after all, and at known, preset times.

# Encapsulation

---

- Disclaimer – C++ does not provide a simple way to calculate differences in dates.
  - As a result, know that the code coming up is representative of what *could* be done, *if* the appropriate class existed.

# A First Object

---

```
class Person
{
    private:
        const string name;
        const MyDate birthday;

        //...
}
```



# A First Object

---

```
public class Person
{
    //...

    public:
        Person(string name, MyDate bday)

        int getAge();
        string getName();
}
```



# A First Object

---

```
int Person::getAge()
{
    return MyDate.differenceInYears(
        MyDate.now(), birthday);
}
```

```
Person::Person(string name, MyDate bday)
:name(name), birthday(bday)
{
}
```

# Analysis

---

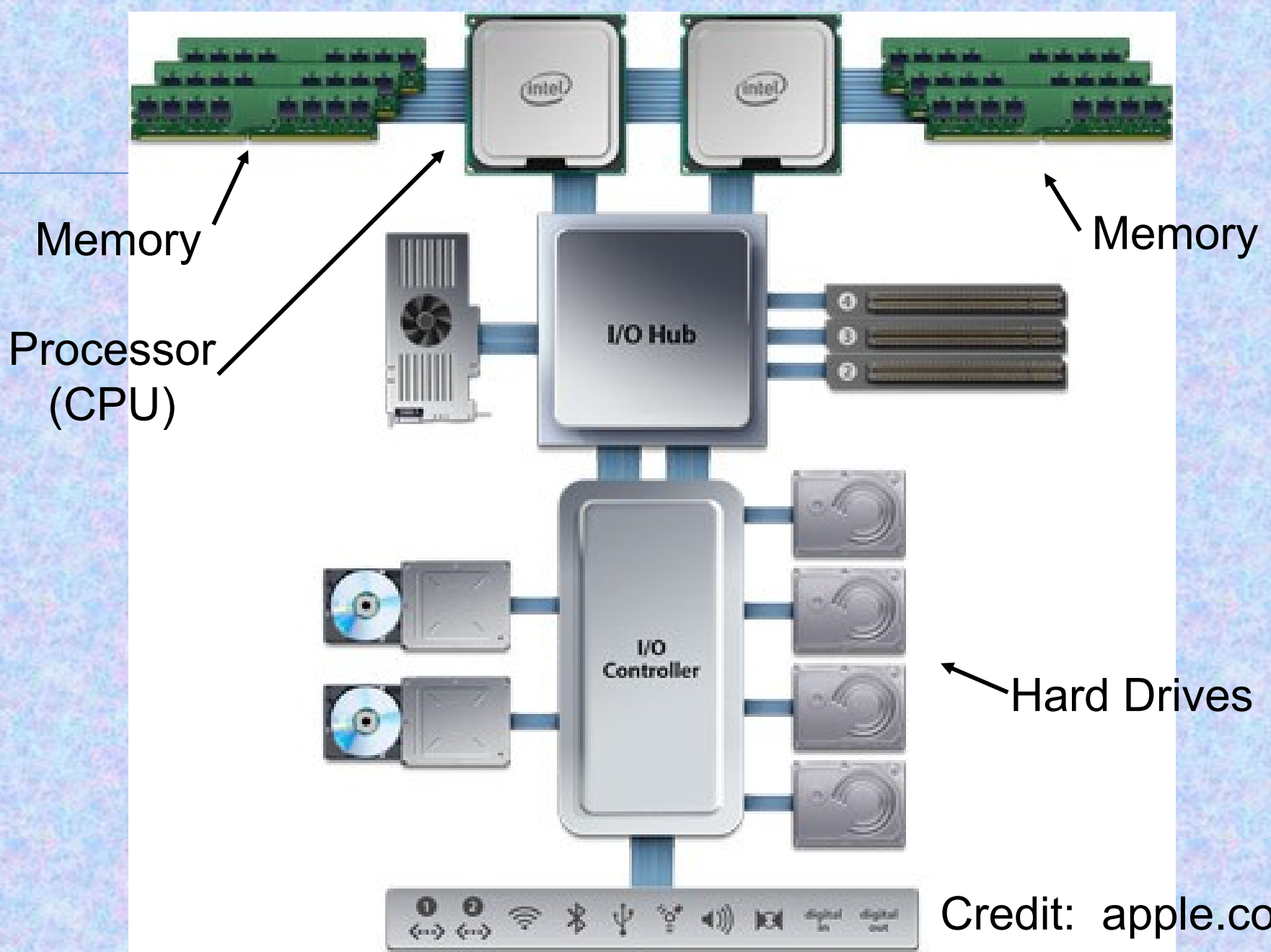
- Note that the “inputs” to an object are managed through its constructors and mutator methods.
- The “outputs” are managed through its accessor methods in such a way that the “constraints” are still enforced.

---

# Value Types vs. Reference Types

- Master the pointers in C++/C
- Importance of memory management





Credit: [apple.com](http://apple.com)



# CPU + Memory

---

- Note that the CPU
  - the “central *processing* unit” is separate from memory, where data are *stored*.



# CPU + Memory

---

- Any data being directly used by a program at any given moment is placed within the CPU on what is called a *register*.



# CPU + Memory

---

- While it's actually more complex than this...
  - Registers are *like* local variables
    - temporary placeholders for values.
  - Memory is *like* a giant, global array.





# CPU + Memory

---

- For now, we'll settle for this simplification.
  - More details will come in CDA 3101
  - Introduction to Computer Organization.





# Memory

---

- Memory is like a giant set of lockers, where each such locker can hold a set, limited amount of data.
- Each locker has a very precise number assigned to it – in computer terms, it has a unique memory *address*. (Like a mailbox.)

# Memory

---

The address of each “locker” is permanently assigned to it.

- You cannot move (or “give a new address to”) an already existing locker.
- Thus, to change the address of data being stored, you must move it from one “locker” to another.

# Memory

---

- It is possible to use multiple adjacent “lockers” to store a large data structure.
  - Another name for a “data structure” would be an *object*, in C++ terms.
  - In this sense, the idea of an “array” is *also* a “data structure.”



# Memory + Arrays

---

- An *array*, when actually utilized during execution, is a large, contiguous (undivided) block of memory.
- The array's starting location – its address within memory – is then stored for future *reference*.
  - All of its data can be found given this starting reference and indices.



# Memory + Arrays

---

The “first” (typically, index “0”) element of the array is stored directly at the starting address of the array.

- Each subsequent element is then stored at a constant offset from this address.

# Working with Data in C++

---

- Note: while the information on the next few slides is written in a C++ fashion, the underlying principles apply to most computer languages.
- Remember the learning experience from Java to C++/C. You will need to learn new languages in your CS careers.

# Working with Data in C++

---

- At its core, all data within a program are stored as a binary number.
  - These are the famous 0's and 1's.
  - Each 0 and 1 is known as a *bit*, or *binary digit*.
  - Eight of these make a *byte*, 1024 bytes make a *kilobyte*, and so forth.



# Working with Data in C++

---

- This “binary number” may be thought of as a ***value***.
- Data which are *directly* represented on a CPU, within a programming language, by a binary number is considered a ***value type***.



# Working with Data in C++

---

- *(Primitive) value types* within C++:
  - int
  - short
  - long
  - char
  - double
  - float
  - bool

This list is not exhaustive.

# Working with Data in C++

---

- Other values are instead handled *through their memory address*.
- Data which are *referenced* on a CPU, within a programming language, through its address is considered a ***reference type***.
  - Note that the *address itself*, while unseen by the programmer, is also a *value*.
    - We call this value... a **pointer**.

# Working with Data in Java

---

- *Reference types* within Java are **all** “classes”/“objects,” and vice-versa.
  - Only the primitive value types are treated “by value” in Java.
- Note that these objects, or classes, may be composed of multiple value types.
  - These values must be obtained *through* the whole object’s **reference**.



# Working with Data in C++

---

- In C++, the programmer may choose which way to handle data.
  - Objects and arrays may be handled by value (within a function) *or* through a pointer.
  - While primitive types default to “by value,” they may be handled by reference!



# Working with Data in C++

---

- By default, most types in C++ will be treated as if they were direct values.
  - The following code will handle both variables “by value.”

```
int i = 4;  
Person p(“Harrison Ford”, 75);
```

- Note the form of the constructor call here – it’s for a “by value” class instance.

# Working with Data in C++

---

- By default, most types in C++ will be treated as if they were direct values.
  - Noteworthy exception: arrays are automatically pointers to the actual storage.
  - The following code is valid:

```
int i[] = {1, 2, 3};  
int* iArr = i; //Arrays ARE ptrs.
```

# Working with Data in C++

---

- Conversely, any type in C++ can be referred to via a pointer.
  - The following code will handle both variables “by reference.”

```
int *i = new int(4);  
Person *p =  
new Person(“Harrison Ford”, 75);
```

- The ‘\*’ symbol denotes that the variable stores a *pointer* to that type.



# Working with Data in C++

---

- If a pointer is not presently referring to any active object, it should **always** be set to “null” – the address zero (0).

```
int *i = 0;
```

```
Person *p = 0;
```

```
Person *q; // WARNING: q is not  
           // pointing to null!
```



# Working with Data in C++

---

- A pointer can be obtained for *any* value – including pointers!
  - This is done with the & operator.

```
Person p("Harrison Ford", 75);
```

```
    Person *pPtr = &p;
```

```
Person **pPtrPtr = &pPtr;
```

```
// Yes, pointers to pointers
```

```
// are completely legal.
```

# Function Calls

---

- In C++, each function may specify the manner by which its parameters are received.
  - The type declaration of the parameter determines whether the data is passed “by value” or “by reference.”
    - Value types are said to be passed “*call by value*”.
    - On the other hand, reference types are said to be passed “*call by reference*.”

# Call By Value

---

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

```
void main()  
{  
    int a = 2;  
    int b = 3;  
  
    swap(a, b);  
}
```

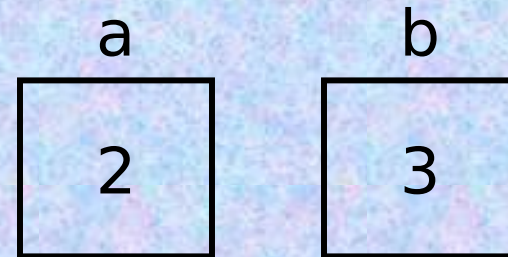


# Call By Value

---

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

```
void main()  
{  
    → int a = 2;  
      int b = 3;  
      swap(a, b);  
}
```

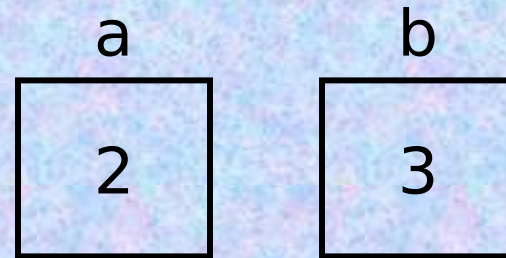
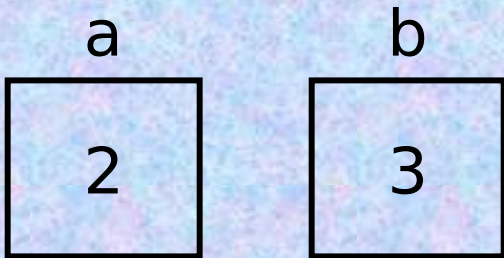




# Call By Value

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

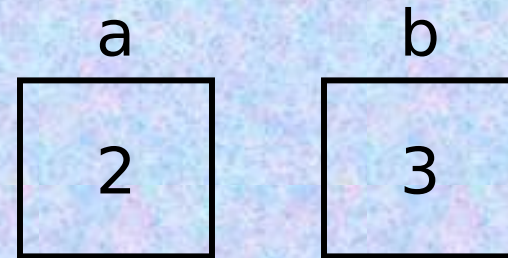
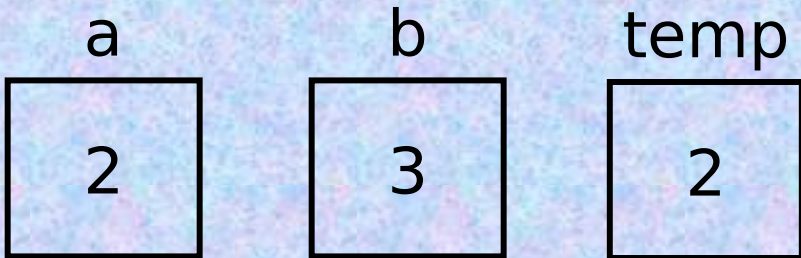
void main()  
{  
    int a = 2;  
    int b = 3;  
    → swap(a, b);  
}



# Call By Value

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

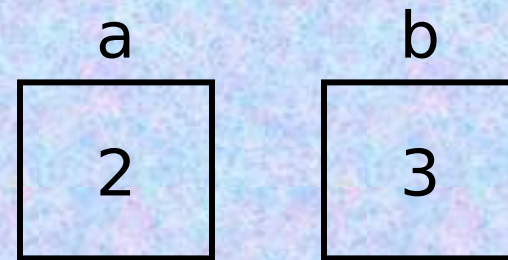
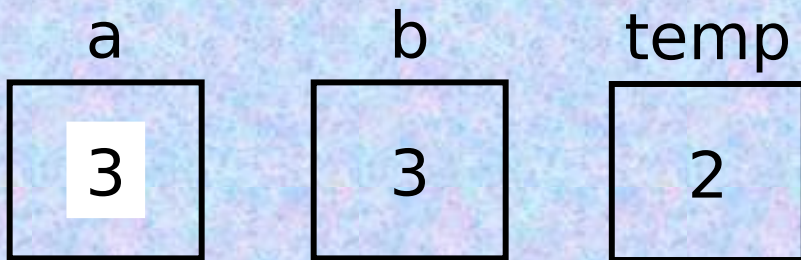
```
void main()  
{  
    int a = 2;  
    int b = 3;  
    swap(a, b);  
}
```



# Call By Value

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

```
void main()  
{  
    int a = 2;  
    int b = 3;  
    swap(a, b);  
}
```

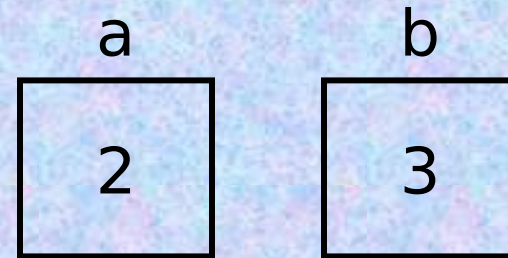
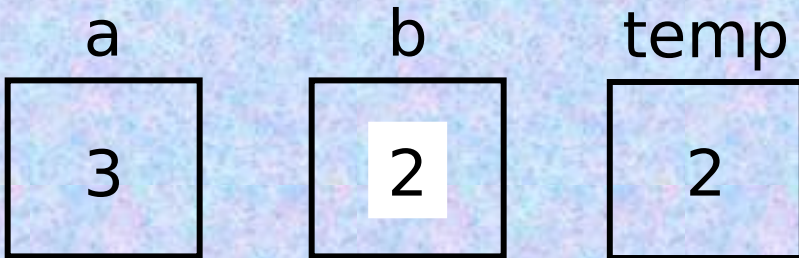




# Call By Value

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

```
void main()  
{  
    int a = 2;  
    int b = 3;  
    swap(a, b);  
}
```

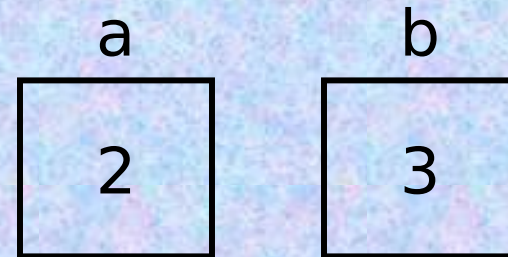




# Call By Value

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

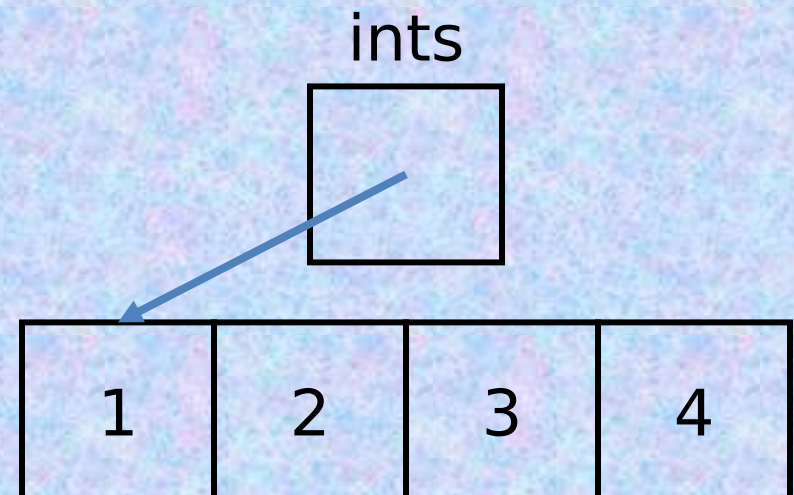
```
void main()  
{  
    int a = 2;  
    int b = 3;  
→ swap(a, b);  
}
```



# Call By Reference

```
void swap(int* ints,  
int i_1, int i_2)  
{  
    int temp =  
        ints[i_1];  
    ints[i_1] =  
        ints[i_2];  
    ints[i_2] = temp;  
}
```

```
public void main()  
{  
    int[] ints =  
        {1, 2, 3, 4};  
  
    swap(ints, 0, 3);  
}
```



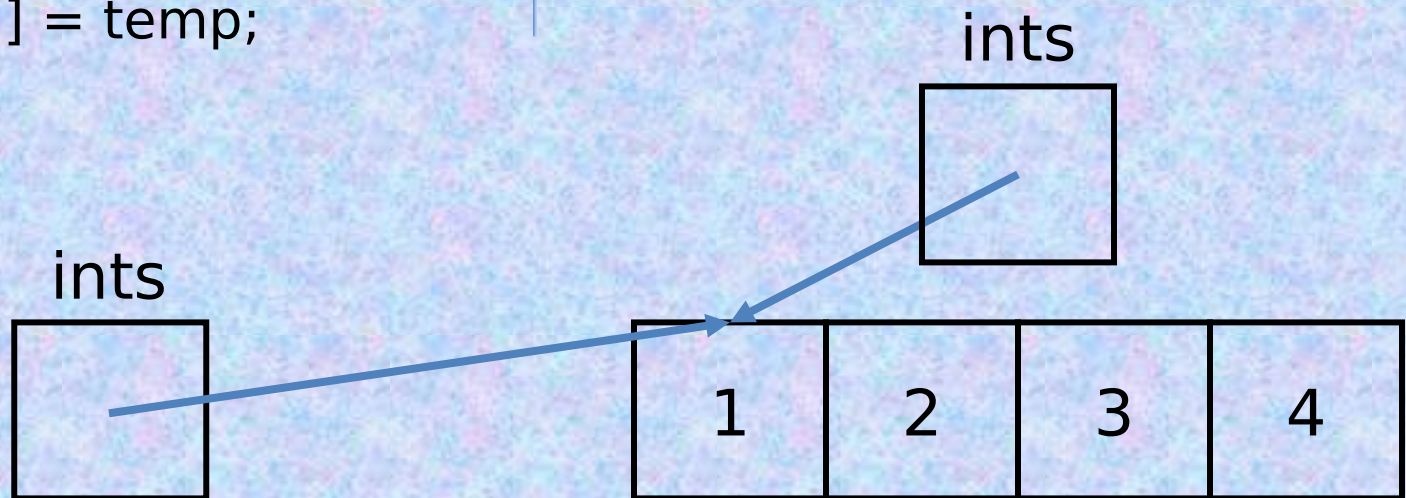
# Call By Reference

```
→ void swap(int* ints,  
    int i_1, int i_2)  
{  
    int temp =  
        ints[i_1];  
    ints[i_1] =  
        ints[i_2];  
    ints[i_2] = temp;  
}
```

```
public void main()  
{
```

```
    int[] ints =  
        {1, 2, 3, 4};
```

```
→ swap(ints, 0, 3);  
}
```

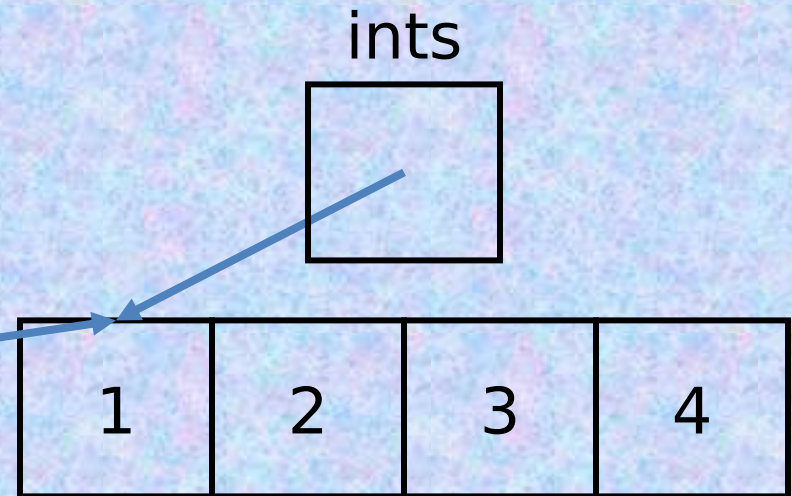
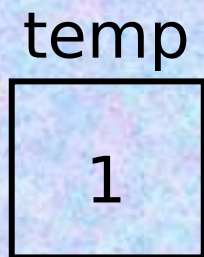




# Call By Reference

```
void swap(int* ints,  
int i_1, int i_2)  
{  
    int temp =  
        ints[i_1];  
    ints[i_1] =  
        ints[i_2];  
    ints[i_2] = temp;  
}
```

```
public void main()  
{  
    int[] ints =  
        {1, 2, 3, 4};  
    swap(ints, 0, 3);  
}
```

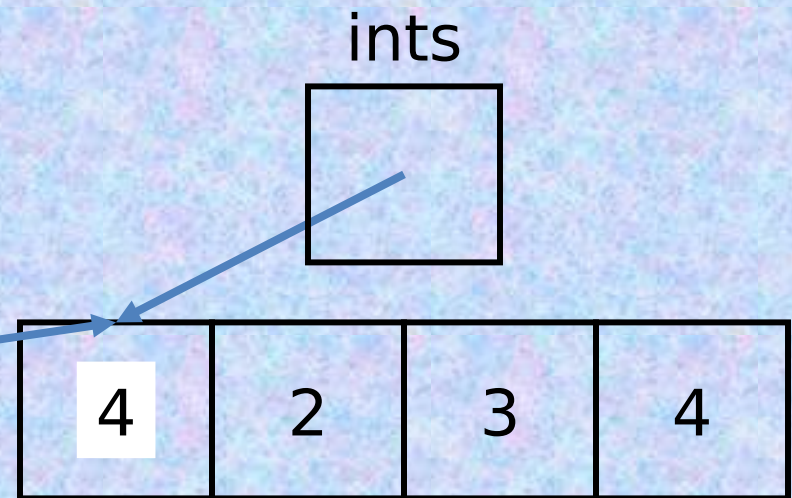




# Call By Reference

```
void swap(int* ints,  
int i_1, int i_2)  
{  
    int temp =  
        ints[i_1];  
    ints[i_1] =  
        ints[i_2];  
    ints[i_2] = temp;  
}
```

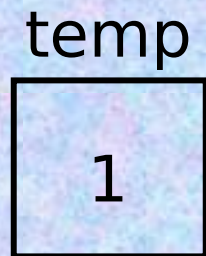
```
public void main()  
{  
    int[] ints =  
        {1, 2, 3, 4};  
    swap(ints, 0, 3);  
}
```



# Call By Reference

```
void swap(int* ints,  
int i_1, int i_2)  
{  
    int temp =  
        ints[i_1];  
    ints[i_1] =  
        ints[i_2];  
    ints[i_2] = temp;  
}
```

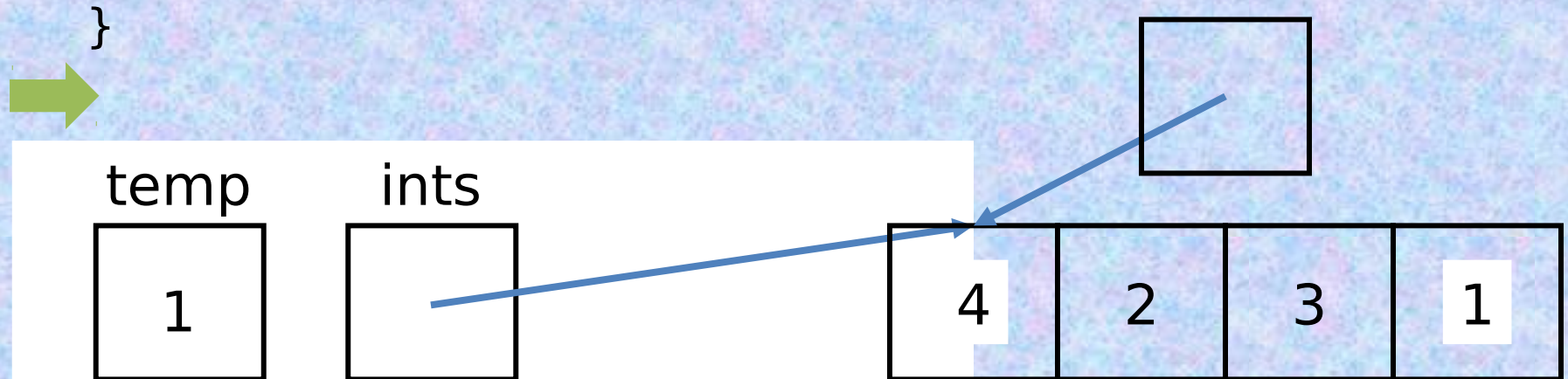
```
public void main()  
{  
    int[] ints =  
        {1, 2, 3, 4};  
    swap(ints, 0, 3);  
}
```



# Call By Reference

```
void swap(int* ints,  
int i_1, int i_2)  
{  
    int temp =  
        ints[i_1];  
    ints[i_1] =  
        ints[i_2];  
    ints[i_2] = temp;  
}
```

```
public void main()  
{  
    int[] ints =  
        {1, 2, 3, 4};  
    swap(ints, 0, 3);  
}
```





# Function Calls

---

- Additionally, using the & operator (instead of a \*) will make that parameter call-by-reference.
  - It will hide the obtained address, but still work with and alter the same object/variable.



# Call By Reference (2)

---

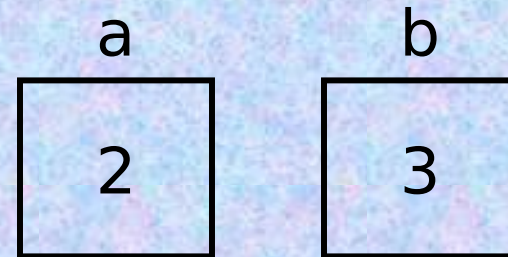
```
void swap(int &a,  
          int &b)  
{  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void main()  
{  
    int a = 2;  
    int b = 3;  
  
    swap(a, b);  
}
```

# Call By Reference (2)

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

```
void main()  
{  
    → int a = 2;  
      int b = 3;  
      swap(a, b);  
}
```



# Call By Reference (2)

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

`void main()  
{  
int a = 2;  
int b = 3;  
→ swap(a, b);  
}`





# Call By Reference (2)

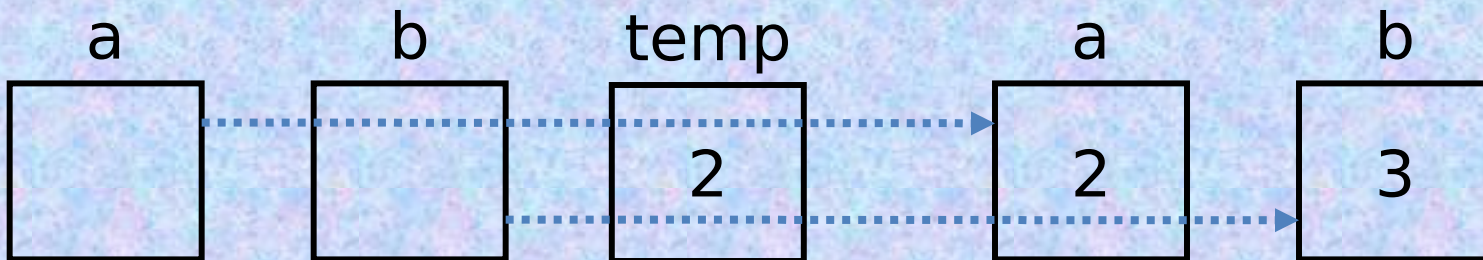
```
void swap(int &a,  
          int &b)
```

```
{  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
void main()  
{
```

```
    int a = 2;  
    int b = 3;
```

```
    swap(a, b);  
}
```

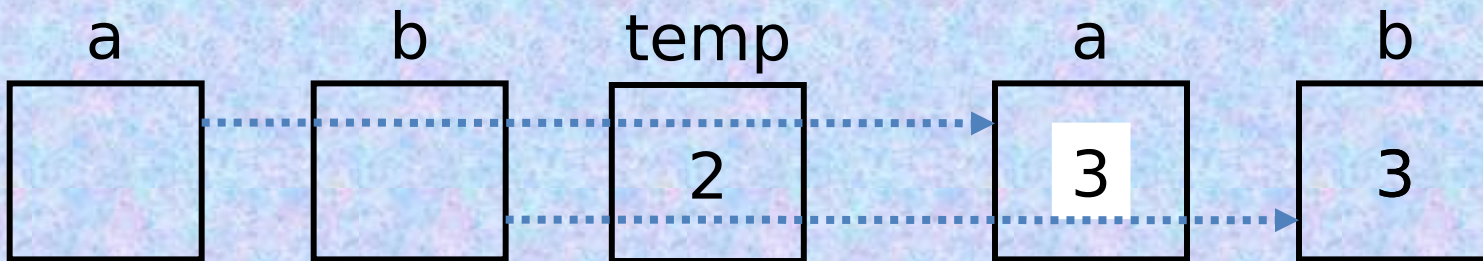




# Call By Reference (2)

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

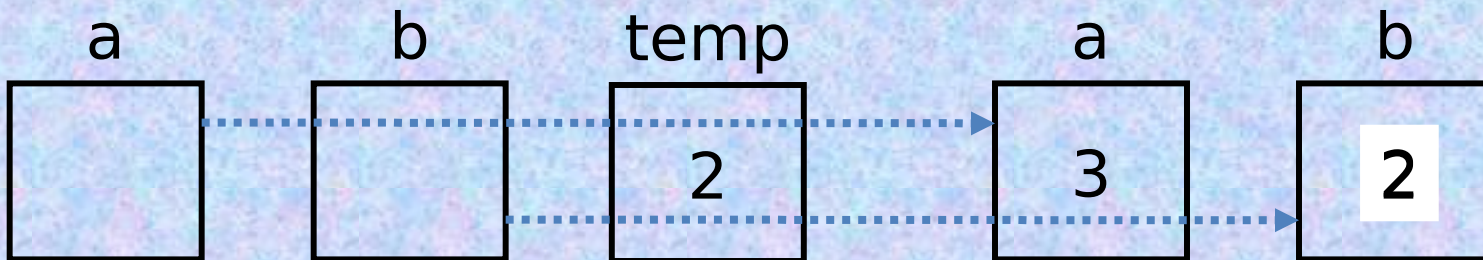
```
void main()  
{  
    int a = 2;  
    int b = 3;  
    swap(a, b);  
}
```



# Call By Reference (2)

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

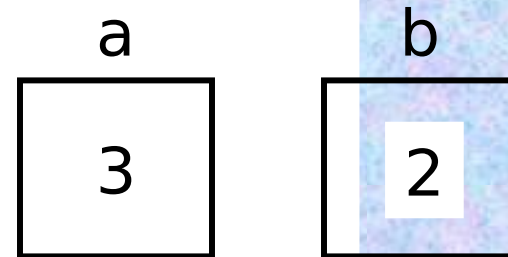
```
void main()  
{  
    int a = 2;  
    int b = 3;  
    swap(a, b);  
}
```



# Call By Reference (2)

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

```
void main()  
{  
    int a = 2;  
    int b = 3;  
→ swap(a, b);  
}
```





# An Aside

---

- To some of you, we imagine that some of C++'s syntax and structure may be pretty foreign, to say the least.
  - In particular, some people have never worked (heavily) with OO before.
  - This is because there's a whole different way of thinking about programming tasks in OO.



# Object-Orientation

---

- Object-orientation is quite different.
  - As we've seen already, part of its design is to enforce the organization of data into logical, conceptual units within the system.
  - Each object keeps its data private (ideally) and seeks to enforce constraints to keep itself in a proper form.

# Object-Orientation

---

- Object-orientation is quite different.
  - Work gets done by objects interacting with other objects.
    - As such, the exact flow of execution in the program may not be easy to track.
  - Object orientation aims to avoid making anything truly global.
    - Java doesn't even *allow* “truly” global variables.
    - C++ allows them.

# A Fraction Object

---

```
class Fraction
{
    private:
        int numerator;
        int denominator;

    public:
        Fraction add(Fraction &f);
}
```



# A Fraction Object

---

```
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```



# Coding in OO

---

- First, let's examine this line of code.

```
f1.add(f2); //Both are Fractions
```

- What is this setting up and modeling?
- Secondly, what is going on in add()?

# Coding in OO

---

```
f1.add(f2); //Both are Fractions
```

- This line is basically saying  
“Call the “Fraction.add()”  
method from the perspective of f1.”

# A Fraction Object

---

So, that line of code has an implied reference to what was previously called "f1."

```
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```



# A Fraction Object

---

This “implied reference” is known as **this** within C++. It’s understood to be implied on any “unqualified” field names in the method below.

```
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```



# A Fraction Object

---

The use of “numerator” and “denominator”, when not preceded by “f.” here, are with respect to **this**.

```
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```

# A Fraction Object

---

What about when we *do* have "f." preceding numerator and denominator?

```
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```

# A Fraction Object

---

In such cases, the perspective *shifts* to that of the object `f`, from which it then operates for the field or method after the `“.”`.

```
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```



# Coding in OO

---

```
f1.add(f2); //Both are Fractions
```

- Even though the `add()` method is operating with two different `Fraction` class instances, the code is able to keep track of which is **this** and which is the parameter `f`.



# Documentation

---

- Documentation is the “plain” English text accompanying code that seeks to explain its structure and use.
  - Some of this documentation is typically in comments, directly in the code.
  - Other documentation may be in external documents.

# Documentation

---

- For complex code, it can be very helpful to place inline comments on a “paragraph” level,  
explaining what purpose that block of code is accomplishing.
  - A line-by-line commentary may clarify *what* the code is doing, but rarely indicates *why*.
    - Note the purpose of your code – its goal.

# Documentation

---

- We've already noted two different ways to comment within C++:

// This is a one-line comment.

/\* This is a block comment,  
spanning multiple lines. \*/



# Documentation

---

- In producing documentation for a method, it is wise to place some form of the “relationships” criterion within the description.
  - Generally, the conceptual purpose which a method, field, or class serves.



# Documentation

---

- One should also include an explanation of the method's *pre-conditions*, if it has any.
  - Pre-conditions: the limitations a particular method imposes on its inputs.
  - If a method is called with arguments that do not match its preconditions, its behavior is considered to be undefined.

# Documentation

---

- As there exists a notion of *preconditions*, there also exist *post-conditions*.
  - Post-conditions: the effect a method has on its inputs (any unaffected/unlisted input should remain untouched), any generated exceptions, information about the return value, and effects on object state.

# Benefits

---

- Documentation helps other programmers to understand the role of each accessible field and method for a given class.
- Documentation inside the code provides great reference material for future maintenance efforts.

# Exceptions

---

When Good Code Goes Bad



# Analysis

---

- Object-orientation is all about recognizing the different “actors” at work in the system being modeled.
  - First, the different data available within the system are organized appropriately.
  - Secondly, functionalities relating to the state of data and its management are bound to that data.

# Analysis

---

- Object-orientation is all about recognizing the different “actors” at work in the system being modeled.
  - These objects (“actors”) may then interact with other objects through well-formed, bounded relationships.

# Analysis

---

1. Constraints: should our object have limitations imposed on it, beyond those implied by the language we're using?
  - Some of our internal state variables (fields) may allow values which make no sense in the context of what our object represents.

# Encapsulation

---

- In a previous lecture, we built a “Person” class with well-defined states and behavior.
  - This involved setting up accessor and mutator methods to ensure the object held a logical, valid state.
  - What we *didn't* cover was how an object should behave when something tries to make its state invalid.



# A First Object

---

```
public class Person
{
    private:
        const string name;
        int age;

    public:
        Person(string name, int age)
        string getName();
        int getAge();
        void haveABirthday();
}
```

# A First Object

---

```
string Person::getName()  
{  
    return this->name;  
}
```

```
int Person::getAge()  
{  
    return this->age;  
}
```

# A First Object

---

```
public void haveABirthday()  
{  
    this->age++ ;  
}
```

# Errors

---

- What if someone initializes our class incorrectly? (Bad “input”?)
  - Naïve solution: be “passive-aggressive.”
  - To be more technical, we *could* refuse to change our object for bad inputs.
  - The problem here is that this is **impossible** for constructors to do. They *must* return an instance.
  - Going passive-aggressive would result in an invalid instance of our object! Not cool!



# Errors

---

- A further problem with going “passive-aggressive” is that this gives *no* indication to outside code that something actually *has* gone wrong.
  - It’s left up to outside code to realize that something is amiss... and if this is ignored, it causes strange behaviors later that can get tricky to test for.

# Errors

---

- What we'd really like to do: we'd like to indicate – forcibly, if necessary – that some sort of error has occurred within the program.
  - This “error” may have happened due to different reasons.
  - Could be the result of a mistake by a programmer...
  - Or it could be the result of a mistake by the program's user.

# Errors

---

- Before examining how we can signal that an error has occurred, let's examine other sources of errors.
  - For example, dividing an integer by zero is one classic source of an error.
  - In calculator programs, this could easily be a user's actual request.

# Errors

---

- What usually happens whenever a program has an error?



A problem has been detected and windows has been shut down to prevent damage to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE\_FAULT\_IN\_NONPAGED\_AREA

If this is the first time you've seen this stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

\*\*\* STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)

\*\*\* SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, Datestamp 3d6dd67c

# Error Handling

---

- Many programming languages have built-in support for “catching” errors that occur.
  - This allows programs to attempt a graceful recovery...
  - Or to crash to the desktop safely instead.

# Error Handling

---

- One mechanism used for signaling an error is that of the *exception*.
  - The *exception* operates by interrupting the standard flow of a program by jumping to specialized code for correcting the error.
  - Note that once an error is encountered within a line of code, it is not safe to evaluate the lines after it, as they may be affected by the error.

# Error Handling

---

- In hardware, this is accomplished through something actually known as an *interrupt*.
  - Interrupts cause the CPU to cease evaluation of a program temporarily to handle something time-sensitive.
  - There are mechanisms in place to resume operation properly, however.



# Error Handling in C++

---

- Within C++, many such *exceptions* are represented by a form of the `std::exception` class.
  - Use `#include <stdexcept>`.
  - In particular, most object-oriented exceptions are represented this way.
  - Other sorts of errors (e.g. arithmetic) may not be represented this way.
  - Note – these are “object” exceptions.

# Error Handling in C++

---

- Exceptions, once *thrown*, must be *caught* by specialized error-handling code.
  - If an exception goes uncaught, the program will crash to the desktop.

# Error Handling

---

- Many of C++'s built-in classes (especially those in the `std` namespace) are designed to *throw* exceptions when errors occur.
- Examples:
  - `exception`
  - `runtime_error`
  - `range_error`

# Throwing Exceptions

---

- For a class to signal an error, it can create an instance of the appropriate exception type and then throw it.
- Exceptions often may take on a message, in string form, which can be useful for debugging purposes.



# A First Object

---

```
Person::Person(string name, int age)
{
    if(name.compare("") == 0)
    {
        throw invalid_argument
            ("Name may not be null!");
    }
    ...
}
```

# A First Object

---

```
Person::Person(string name, int age)
{
    ...
    if(age < 0)
    {
        throw out_of_range
            ("Age must be non-negative!");
    }
    ...
}
```

# A First Object

---

- Note that whenever the constructor throws an exception that isn't handled internally, it cancels the construction process.
  - Constructor problem solved!

# Error Handling

---

- The general structure for handling an error that has occurred:

```
try {  
    throw exception();  
} catch (exception e){  
    //Fix the error!  
}
```



# Error Handling

---

```
try  
{  
    throw exception()  
}  
catch (exception e)  
{  
    //Fix the error!  
}
```

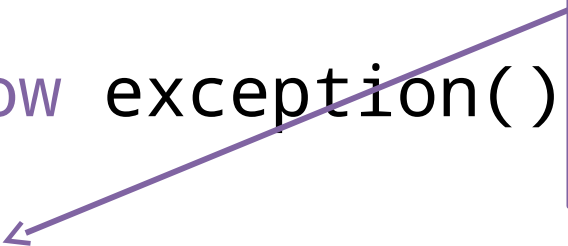


“try” this code and  
see if any errors  
happen

# Error Handling

---

```
try
{
    throw exception()
}
catch (exception e)
{
    //Fix the error!
}
```

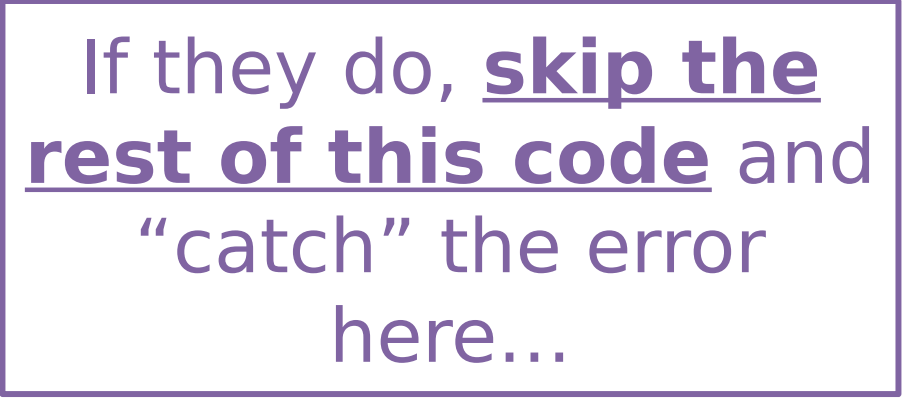


If they do, skip the rest of this code and “catch” the error here...

# Error Handling

---

```
try
{
    throw exception()
}
catch (exception e)
{
    //Fix the error!
}
```

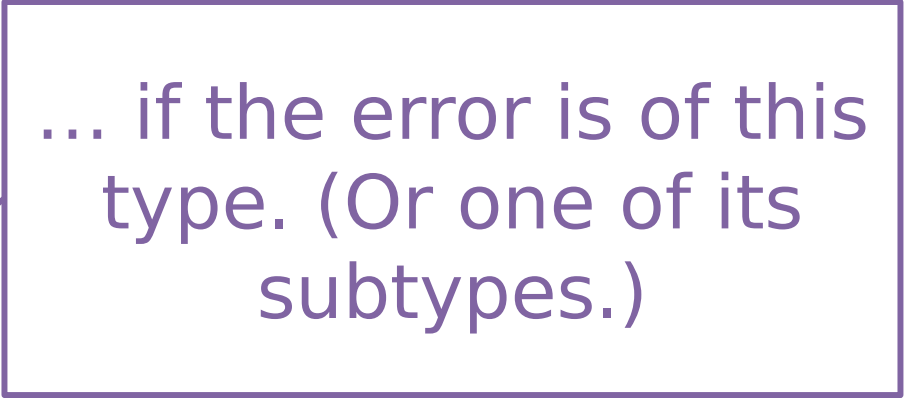


If they do, skip the rest of this code and “catch” the error here...

# Error Handling

---

```
try
{
    throw exception()
}
catch (exception e)
{
    //Fix the error!
}
```



... if the error is of this type. (Or one of its subtypes.)



# Error Types

---

- There are actually multiple sorts of errors which can occur within a program and its code.
  - *Compile-time* errors: the interpreter / compiler can't make sense of your code.
  - *Logical* errors: the program doesn't crash, but it behaves differently than intended.

# Error Types

---

- There are actually multiple sorts of errors which can occur within a program and its code.
  - *Run-time* errors: Errors which crash a program, but were not intentionally generated by programmer code.
  - Generally, user-generated issues caused by bad inputs. (GIGO, PEBKAC)
  - When a calculator program is told to divide by zero, if it doesn't check for illegalness, a runtime error will occur.

# Error Types

---

- There are actually multiple sorts of errors which can occur within a program and its code.
  - *Generated* errors: the program detects that it is malfunctioning and generates an error to signal it.
  - Often generated to prevent run-time errors from crashing the program. Making these gives a chance for recovery if they are caught elsewhere.

# Error Types

---

- When a program hangs (goes unresponsive), it's typically a logical error.



File Edit Format View Help

SYSINTERNALS SOFTWARE LICENSE TERMS

These license terms are an agreement between Sysinternals (a Microsoft Corporation company) and you.

- \* update
- \* suppl
- \* Intern
- \* suppor

for this

BY USING

If you c

1. INSTA

2. SCOPE

You may not:

- \* work around any technical limitations in the binary version of the software;
- \* reverse engineer, decompile or disassemble the binary version of the software, except to the extent applicable law expressly permits, despite this limitation;

Notepad

Notepad is not responding

If you close the program, you might lose information.

→ Close the program

→ Wait for the program to respond

# Recovery

---

- Unfortunately, like in the prior example, not all errors can be detected.
  - Sometimes, an application can get stuck in an infinite loop, rendering it completely unresponsive.
  - Multithreaded applications can also become stuck due to “deadlock” and “livelock” situations.

# On the Use of Exceptions

---

- Exceptions are extremely handy to have as a tool for an object to indicate bad inputs to its constructor or method.
- However, exceptions are quite “expensive”, computationally, to throw.
  - Remember, they interrupt *everything*.

# On the Use of Exceptions

---

- Objects should throw exceptions when:
  - A constructor receives (bad) inputs that would result in an invalid object.
  - A method receives bad input
  - Out of range index or value
  - Null pointer



# On the Use of Exceptions

---

- Objects should throw exceptions when:
  - A method cannot perform the requested action.
  - Some objects may have different “modes,” where certain actions may only be possible in certain situations.
  - Example: a file must be opened to read or write from/to it.

# On the Use of Exceptions

---

- Objects should throw their *own* exceptions, rather than relying on a future method to throw exceptions.
  - When debugging, it is better to know the underlying source of the erroneous error.
  - Thus, the sooner code can detect that an error will occur (even if later in the chain), the better.

# On the Use of Exceptions

---

- Objects should throw their *own* exceptions, rather than relying on a future method to throw exceptions.
  - Failure to do so will make it seem as if the object is miscoded, using the future method incorrectly.

# On the Use of Exceptions

---

- Exceptions are a useful tool for object-orientation, allowing objects to actively prevent actions that would be invalid.



# On the Use of Exceptions

---

- Exceptions are a useful tool for object-orientation, allowing objects to actively prevent actions that would be invalid.
  - They also allow objects to report *why* those actions are invalid, which aids debugging.
  - Sometimes, it may even be possible to recover from the error, depending on its type.