CSE 165/ENGR 140 Intro to Object Orient Programming

Lecture 1 - Introduction to Objects

CSE 165/ENGR 140: Spring 2022

- Principles that you learn in this class will be applied throughout your career
- This class is fundamental to becoming a good software engineer
- My ultimate goal is to help each of you:
  - Become a solid software engineer
  - Get a good job after you graduate
  - Become a better you

About me: Ammon Hepworth, PhD

- Grew up in San Diego, lived in UT, CT, TX, Hong Kong
- Married with 2 kids (wife from Merced)
- Developed software since 2007
- Former CEO of Jurybox Technologies
- BS, MS and PhD from Brigham Young University

TA Intro

- Hoa Nguyen
- Ghazal Zand

About you (Zoom Poll)

- Where are you from?
- What's your major?
- What Programming languages do you know?
  - Java, C, C++, Python, C#, Visual Basic, JavaScript, HTML/CSS

Contact Info

- Lecturer
  - Ammon Hepworth
  - Email: ahepworth@ucmerced.edu
  - Office: SE2 278
  - Office Hours: Tuesdays at 10:30 – 11:30am
- Teaching Assistants
  - Hoa Nguyen, hnguyen257@ucmerced.edu
  - Ghazal Zand, gzand@ucmerced.edu

Course Overview

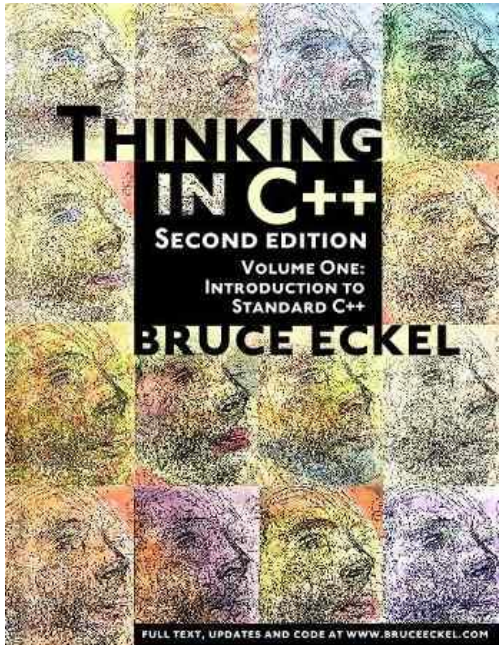
- CatCourses
  - Check regularly for announcements.
- 2 Lectures and 1 Lab per week
- Mid-term exam in class (March 29, tentative)
- Final exam on last day of class (May 5)
- Project presentation during final exam slot (May 10)

Course Objectives

- Create programs in Linux
- Learn C and C++
- Develop good programming habits
- Understand the concept of object-oriented programming
- Labs:
  - Giving each other help in finding bugs and in understanding the assignment is perfectly acceptable.
  - You may allow other students to see small portions of your code on-screen as an example, but you may not allow them to copy directly (or give them copies of your code)
  - We will be using C++; you can use any operating system

Course Material

- Text Book:
  - Bruce Eckel - *Thinking in C++: Introduction to Standard C++*, 2nd Edition, Volume 1, 2000, Prentice Hall
  - https://www.micc.unifi.it/bertini/download/programmazione/TICPP-2nd-ed-Vol-one-printed.pdf
- Online resources:
  - http://www.cplusplus.com/doc/tutorial
  - PDFs after each lecture in CatCourses



Prerequisites

- CSE 031, CSE 100 and MATH 024
- Math: logarithms, series, Boolean logic, matrices, calculus …
- Coding: basic programming experience (Java, C, C++)
- Curiosity: observe how the world is run by computers, and what problems we face.

Grading

- Lab assignments: 25%
- Participation: 5%
- Quizzes: 5%
- Mid-term: 20%
- Final exam (comprehensive): 25%
- Project: 20%

Policies

- Do:
  - Ask applicable questions on Zoom chat
  - Raise hand in Zoom to get attention
  - Participate
  - Have fun
- Do not:
  - Copy someone else's code
  - Give your code away
  - Outsource your assignments
  - Cheat

Don’t be a cheater!

- Communicating information to another student during examination.
- Knowingly allowing another student to copy one’s work.
- Offering another person’s work as one’s own.
- **You are a better than that**

You can do hard things

- Programming is hard
  - Learning another language
  - Learning new concepts
  - Applying mathematics
- Ask a lot of questions
- Just keep trying

https://youtu.be/KdxEAt91D7k

Hints for success

- Attend lectures
- Attend labs
- Read the textbook
- Do & understand the labs YOURSELF
- Take notes while reading and in lectures
- Ask questions

History Lesson

- C developed by Dennis Ritchie at AT&T Bell Labs in the 1970s.
  - Used to maintain UNIX systems
  - Many commercial applications were written in C
- C++ developed by Bjarne Stroustrup at AT&T Bell Labs in the 1980s
  - Overcame several shortcomings of C
  - Incorporated object-oriented programming
  - C remains a subset of C++

History Lesson

- Where did C++ come from?
  - Derived from the C language
  - C was derived from the B language
  - B was derived from the BCPL (Basic Combined Programming Language)
- Why the ‘++’?
  - ++ is the post-increment operator
  - Therefore, C++ is C, ++

Object oriented programming (OOP)

- Everything is viewed as an object
- A program is a bunch of objects telling each other what to do by sending messages
- Each object has its own memory made up of other objects
- Every object has a type
- All objects of a particular type can receive the same messages

Object oriented software goals

- Robustness
  - How well can it handle errors?
- Adaptability
  - How portable is it on different hardware and operating systems?
- Reusability
  - How much code can be reused in other applications?

Object oriented concepts

- Encapsulation
  - The ability to package data with functions allows you to create a new data type
  - Example: members are encapsulated in a class/structure
- Implementation hiding (same as data/information hiding)
  - Access control
  - To prevent important data from being corrupted
- Interface
  - It establishes what requests you can make for a particular object
  - It is an abstraction of an object
  - It tells what an object does without telling the details (ex. header files).

Good Programming Practices

- Good programmers format programs so they are easy to read
- Good programmers typically:
  - Place opening brace ‘{‘ and closing brace ‘}’ on a line by themselves
  - Indent statements
  - Use only one statement per line
  - Use intuitive object names
    - e.g. int count, instead of int c

C++ Compiler

- C++ compilers accepts almost any pattern of line breaks and indentation
- However, this invites bad programming practices
- We don’t want to learn bad programming habits, they are hard to unlearn

Very Simple Program

```
#include <iostream>
using namespace std;
int main()
{
        int classNumber = 165;
        cout << "Hello world!\n";
        cout << "Welcome to CSE ";
        cout << classNumber;
        cout << "!\n";
        return 0;
}
```

Output:

```
Hello world!
Welcome to CSE 165!
```

Things to notice about the example

- Variables are declared **before** they are used
  - Typically, variables are declared at the beginning of the program
- **Statements** (can be multi-line) end with a semi-colon
- The #include directive: **#include <iostream>**
  - Tells compiler where to find information about items used in the program
- **iostream** is a library containing definitions of **cin** and **cout**

C++ Syntax

- using namespace std;
  - Tells the compiler to look for methods and data types in the "std" **namespace**
  - A **namespace** allows us to have methods, classes, and data types with the same name that exist in separate "namespaces" (more detail about it later in the semester)
- In C++, our program begins with a main() method:
  - int main()
- Which returns an integer value at the end of the its execution (optional in many compilers):
  - return 0;

C++ Syntax Highlights

- By now you've probably noticed that C++ looks a lot like Java, though not identical by any means
- That means a lot of your old knowledge of simple logical structures (do, while, for, if, else, etc.) will transfer
- When the compiler fails, it will try to give you a meaningful error message
- However, sometimes they're hard to understand, so be patient

Writing C++ Code

- C++ source code can be written with a text editor
  - **gedit** is popular in Linux as well
  - **nano** is simple with less functionality
- Don’t need an IDE, but it can help
  - Visual Studio
  - NetBeans
  - Eclipse
- The compiler on your system converts the source code to object code
- The linker combines all the object code into an executable program

Reading assignment

- Reading assignment
  - Chapter 1 and 2 of textbook
- No lab this week

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 2 – Programming in C++

Annoucements

- Reading:
  - Ch 1 and 2: focus on pages 23 - 46 and 83 – 118
- A little about Jurybox
  - juryboxapp.com
  - Web and mobile app
  - Tech stack (MERN)
    - React (JavaScript library for building User Interfaces)
    - Node.js (JavaScript runtime environment)
    - Express (Node.js web framework)
    - MongoDB (No SQL database)

Object oriented programming (OOP)

- Everything is an object
- A program is a bunch of objects telling each other what to do by sending messages
- Each object has its own memory made up of other objects
- Every object has a type
- All objects of a particular type can receive the same messages

Declaration vs. Definition (Ch. 2)

- Declaration
  - Gives a name (identifier) to a variable or function
  - Variable: `extern int a;` //extern means the variable will be defined later
  - Function: `int func1(int, int);`
- Definition
  - Allocates a memory location (storage) for a variable or function
  - Variable: `int a;`
  - Function: `int func1(int length, int width) {…};`
  - It is illegal to define a variable or function multiple times in a program

Declaration vs. Definition

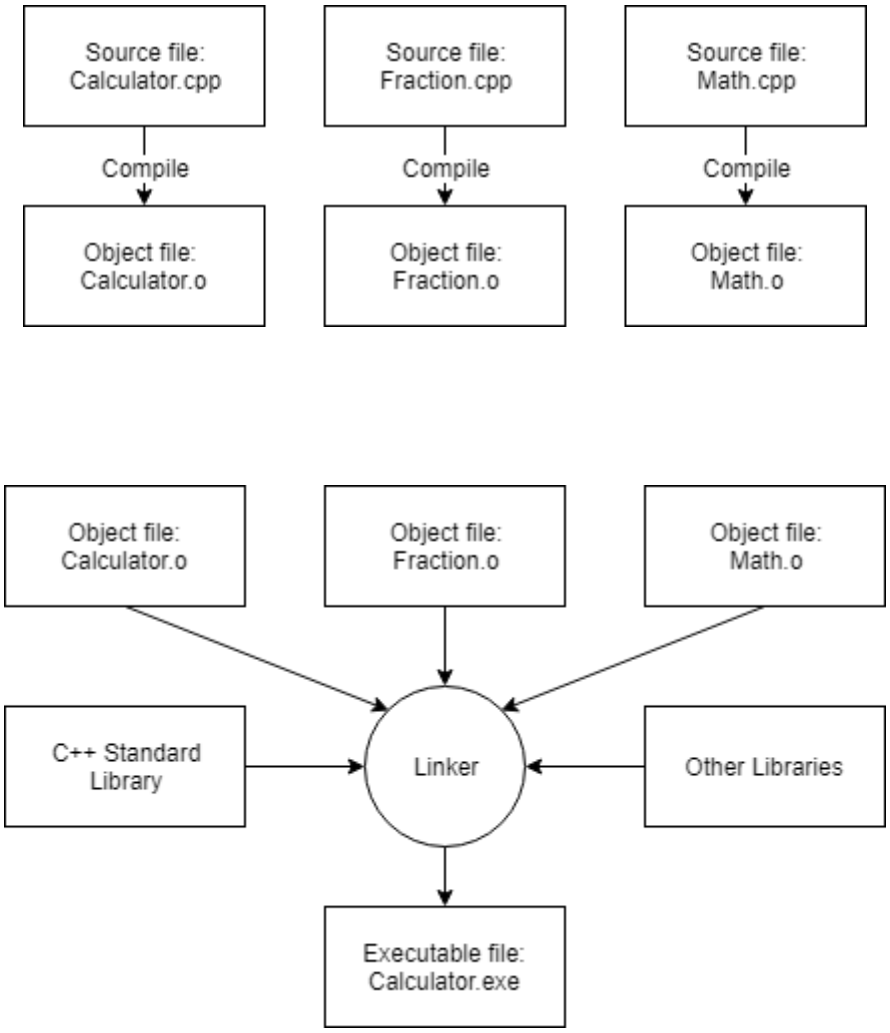
```
//: C02:Declare.cpp
// Declaration & definition examples

extern int i;              // Declaration without definition
extern float f(float);     // Function declaration

float b;                   // Declaration & definition
float f(float a) {         // Definition
    return a + 1.0;
}

int i;                     // Definition
int h(int x) {             // Declaration & definition
    return x + 1;
}

int main() {
    b = 1.0;
    i = 2;
    f(b);
    h(i);
}
```

Writing C++ Code

- C++ source code can be written with a text editor, we don’t need a fancy IDE
- Example editors:
  - **gedit** is popular in Linux
  - **nano** is simple with less functionality

Writing C++ Code

- Compiler
  - checks your code for errors
  - converts the source code to object code: **xx.cpp -> xx.o**
- Linker
  - combines all the object code into an executable
  - **(xx.o, yy.o, zz.o) -> aaa.exe**



First Program

```
//: C02:Hello.cpp
// Saying Hello with C++
#include <iostream>       // Stream declarations
using namespace std;

int main() {
  cout << "Hello, World! I am "
       << 8 << " Today!" << endl;
}
```

Output:

Hello, World! I am 8 Today!

Insertion and Extraction Operators

- Insertion Operator <<
  - cout << "This is output" << endl;
  - Inserts data into the output stream

- Extraction Operator >>
  - cin >> X;
  - Extracts data from the input stream

More About *iostream*

We can display integers in different bases:

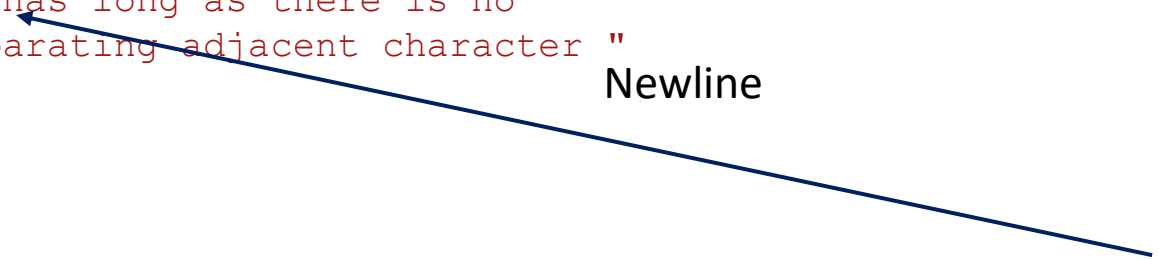
```
//: C02:Stream2.cpp
#include <iostream>
using namespace std;

int main()
{
  // Specifying formats with manipulators:
  cout << "15 in decimal: "
       << dec << 15 << endl;
  cout << "in octal: " << oct << 15 << endl;
  cout << "in hex: " << hex << 15 << endl;
  cout << "a floating-point number: "
       << 3.14159 << endl;
 }
```

Output:

15 in decimal: 15
in octal: 17
in hex: f
a floating-point number: 3.14159

String Concatenation Example

```
//: C02:Concat.cpp
#include <iostream>
using namespace std;

int main()
{
  cout << "This is far too long to put on a "
    "single line but it can be broken up with "
    "no ill effects\nas long as there is no "
    "punctuation separating adjacent character "
    "arrays.\n";
}
```



Output:

This is far too long to put on a single line but it can be broken up with no ill effects as long as there is no punctuation separating adjacent character arrays.

Reading Input

```
//: C02:Numconv.cpp
#include <iostream>
using namespace std;

int main()
{
  int number;
  cout << "Enter a decimal number: ";
  cin >> number; //Read input from user
  cout << "value in octal = 0"
       << oct << number << endl;
  cout << "value in hex = 0x"
       << hex << number << endl;
}
```

Output:

Enter a decimal number: 128
value in octal = 0200
value in hex = 0x80

String class

- Allows you to manipulate the content of a character array
- Needs to be included at the beginning of a program

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

int main()
{
  string proclamation, day; // Empty strings
  string greeting = "Hello, World."; // Initialized
  string iam("I am"); // Also initialized
  day = "Today"; // Assigning to a string
  proclamation = greeting + " " + iam; // Combining strings
  proclamation += " 8 "; // Appending to a string
  cout << proclamation + day + "!" << endl;
}
```

Output:

Hello, World! I am 8 Today!

Wake up!

https://youtu.be/nMJdsQL_Bco

File Input/Output

- To read from or write to a file, we need to include:
  - #include <fstream>
- Before **reading** from a file, we need to define and open a file to be read:
  - ifstream myfile(<*file_name*>); //**ifstream:** Stream class to read from *file_name*
- Before **writing** to a file, we need to define and open a file to be written:
  - ofstream myfile(<*file_name*>); //**ofstream:** Stream class to write on *file_name*
- Close the file after finishing the operations with it:
  - myfile.close();

File Input (Read)

- Once a file is opened, we can read the content by lines:
  - getline (myfile, *line*); // read current line of file and put it in *line*
  - It discards the newline character at the end
  - After reading a line, getline will start at the next line when it is called again
  - You don’t need to increment the line number in your code

File Output (Write)

- Once a file is opened, we can write the content onto the file as if writing to console:
  - myfile << "Writing to file is similar\n";
  - Instead of ***cout***, we use the instance variable that contains the file object, in this case ***myfile***

File IO Example

```
// Read/write file
#include <string>
#include <fstream>
using namespace std;

int main()
{
  ifstream input("file.txt"); // Open for reading
  ofstream output("file_out.txt"); // Open for writing
  string myString;
  while(getline(input, myString)) // Discards newline char
  {
    output << myString << "\n"; // ... must add newline back
  }
}
```

File IO Example

```
// Read an entire file into a single string
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream input("FillString.cpp");
  string myString, line;
  while(getline(input, line))
    myString += line + "\n";
  cout << myString;
}
```

File IO Example

```
// Example using is_open
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ifstream infile;
  infile.open ("test.txt");
  if (infile.is_open()) // Check if the file is open
  {
    while (!infile.eof()) // Check if it reaches the end of file
        cout << (char) infile.get(); // Read character by character
    infile.close();
  }
  else
  {
    cout << "Error opening file";
  }
  return 0;
}
```

Vector (Array list)

- It works similarly as arrays
- Elements in a vector of size N are accessed by their indices [0…N-1]
- We can change the size of a vector dynamically
  - We don’t need to worry about the size as the number of data grows
- Vector is a **template class**
  - It can work with any data type
  - vector<data_type> myVector
    - vector<int> scores

STL Vector

- There is a vector class in the Standard Template Library in C++
- Member functions of STL Vector class (given a vector V):
  - **resize(n)**: Resize V, so that it has space for n elements
  - **size()**: Return the number of elements in V
  - **front()**: Return a reference to the first element of V
  - **back()**: Return a reference to the last element of V
  - **push_back(e)**: Append a copy of the element e to the end of V, thus increasing its size by one
  - **pop_back()**: Remove the last element of V, thus reducing its size by one
  - **insert(i,e)**: Insert a copy of the element e to the i^{th} position of V
  - **erase(i)**: Remove the element at the i^{th} position of V

Vector Example

```
//: C02:Fillvector.cpp
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main()
{
  vector<string> v;
  ifstream in("Fillvector.cpp");
  string line;
  while (getline(in, line)) //getline returns true if read successfully
    v.push_back(line); // Add the line to the end of v
  // Add line numbers:
  for(int i = 0; i < v.size(); i++)
    cout << i + 1 << ": " << v[i] << endl;
}
```

Vector Example

```
//: C02:GetWords.cpp
// Break a file into whitespace-separated words
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main()
{
  vector<string> words;
  ifstream in("GetWords.cpp");
  string word;
  while(in >> word) // Extraction operator reads until white space
    words.push_back(word);
  for(int i = 0; i < words.size(); i++)
    cout << words[i] << endl;
}
```

Vector Example

```
//: C02:Intvector.cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;

  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Assignment
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
}
```

What's the output?

Output

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 10, 20, 30, 40, 50, 60, 70, 80, 90,
```

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 3 – C in C++

(Ch. 3)

Announcement

- Lab #1 this week
  - Due before next lab begins
- Reading assignment
  - Ch. 3 (Quizzes will be based on lecture and reading)

Functions

- A block of statements “called” from a program:

```
type name ( argument1, argument2, ...)
{
    statement1
    statement2
    …
}
```

Functions

- A function:
  - may return a value (use void if not returning a value)
  - may use and change arguments
  - can be called multiple times
  - could be reused in multiple programs
  - it enables to modularize the code (building blocks)

Functions

- Declaration

```
int translate ( float x, float y, float z );
int translate ( float, float, float );
```

- Definition

```
int translate ( float x, float y, float z )
{
    x = y = z;
}
```

- Functions of same name may have different arguments

Functions

- Example Definitions

```
int addIntegers ( int x, int y )
{
    return x + y;
}
void say ( string sentence)
{
    cout << sentence << endl;
}
```

- Calling the functions

```
int answer = addIntegers ( 5, 7);
say ( "Hello World!" );
```

Execution Control: if-else

```
// Demonstration of if and if-else conditionals
#include <iostream>
using namespace std;
int main() {
    int i;
    cout << "Type a number and hit 'Enter'" << endl;
    cin >> i;
    if (i > 5)
      cout << "It's greater than 5" << endl;
    else
        if (i < 5)
           cout << "It's less than 5 " << endl;
    else
        cout << "It's equal to 5 " << endl;
}
```

Execution Control: if-else

- ***To avoid obscure if-else chains, use { } and indentations.***

```
// Demonstration of if and if-else conditionals
#include <iostream>
using namespace std;
int main()
{
    int i;
    cout << "type a number and 'Enter'" << endl;
    cin >> i;
    if (i > 5)
        cout << "It's greater than 5" << endl;
    else
    {
        if (i < 5)
            cout << "It's less than 5 " << endl;
        else
            cout << "It's equal to 5 " << endl;
    }
}
```

Execution Control: while

while (<boolean_expression>)
    <loop body>

```
// Guess a number (demonstrates "while")
#include <iostream>
using namespace std;
int main()
{
  int secret = 15;
  int guess = 0;

  // "!=" is the "not-equal" conditional:
  while(guess != secret){ // Compound statement
    cout << "guess the number: ";
    cin >> guess;
  }

  cout << "You guessed it!" << endl;
}
```

Execution Control: do-while

do

    <loop body>

while (<boolean_expression>)

```
// The guess program using do-while
#include <iostream>
using namespace std;
int main() {

  int secret = 15;
  int guess; // No initialization needed here

  do {
    cout << "guess the number: ";
    cin >> guess; // Initialization happens
  } while ( guess!=secret );

  cout << "You got it!" << endl;
}
```

Execution Control: for

for ([<initialization>]; [<condition>]; [<increment>])
< loop body (statement; or {block}) >

```
// Display all the ASCII characters
// Demonstrates "for"
#include <iostream>
using namespace std;
int main()
{
  for (int i = 0; i < 128; i++)
   {
     if (i != 26) // ANSI Terminal Clear screen
        {
             cout << " value: " << i
                  << " character: "
                  << char(i) // Type conversion
                  << endl;
        }
   }
}
```

Wake up

- https://youtu.be/kKAkj3rCBEo

Execution Control: break/continue

- Break
  - Exit the loop
- Continue
  - Skip current iteration

```
// Simple menu demonstrating "break" and "continue"
#include <iostream>
using namespace std;
int main(){
  char c; // To hold response
  while(true){
    cout << "MAIN MENU> c: continue, q: quit -> ";
    cin >> c;
    if ( c == 'q' ) break; // Out of "while(1)"
    if ( c == 'c'){
       cout << "Press a or b: ";
       cin >> c;
       if ( c == 'a' ){
          cout << "you chose 'a'" << endl;
          continue; // Back to main menu
       }
       if ( c == 'b' ){
          cout << "you chose 'b'" << endl;
          continue; // Back to main menu
       }
     }
  }
}
```

Execution Control: switch

```
char command;
if (command == 'I')
    <statement1>
else if (command == 'R')
    <statement2>
else
    <statement3>
```

```
char command;
…
switch (command)
{
    case 'I' :
        <statement1>
        break;
    case 'R' :
        <statement2>
        break;
    default :
        <statement3>
        break;
}
```

Execution Control: switch

```
// A menu using a switch statement
#include <iostream>
using namespace std;
int main(){
  bool quit = false; // Flag for quitting
  while(!quit){
    cout << "Select a, b or q to quit: ";
    char response;
    cin >> response;
    switch(response){
      case 'a' : cout << "you chose 'a'" << endl;
                 break;
      case 'b' : cout << "you chose 'b'" << endl;
                 break;
      case 'q' : cout << "quitting menu" << endl;
                 quit = true;
                 break;
      default : cout << "Please use a,b, or q!" << endl;
    }
  }
}
```

Execution Control: goto

- Good programming style avoids using ***goto***

```
//: C03:gotoKeyword.cpp
// The infamous goto is supported in C++
#include <iostream>
using namespace std;
int main() {
  long val=0;
  for ( int i=1; i<1000; i++ ) {
    for ( int j=1; j<100; j+=10 ) {
      val = i * j;
      if ( val>47000 )
        goto bottom;
        // break would only go to the outer 'for'
        // use of goto may be justified in such cases
    }
  }
  bottom: // A label
  cout << val << endl;
}
```

Data types: scope

- Variables can be defined anywhere
- Scope of a variable
  - global:
    - Can be used through out the program
    - Use extern keyword for variables used across multiple files
  - local: within a scope
  - global but of limited scope: use static keyword
    - Limited to a file
    - More on static in future lectures

Data types: primitive types

- Types:
  - char, int, float, double
- Modifiers:
  - unsigned, signed, short, long
- Type limits:
  - Data type size varies depending on systems (16- vs 32- vs 64-bit CPU)
  - Use ***sizeof*** operator to determine size of each data type

Data types: bool

- ***bool*** was introduced in C++
- Can only contain true (1) or false (0)
- Conditional expressions always produce boolean types

Data types: bool

Examples:

```
bool is_small = a<=10;
cout << (is_small? "small" : "large");
```

```
bool in_unit_interval = n>=0.0 && a<=1.0? true:false;
```

```
bool initialized = false;
if(!initilized)
      init();
```

```
bool newcmd = true;
while(newcmd)
 {
      newcmd = enter_new_command();
 }
```

Data types: constants

- Modifier ***const***:
  - Can be of any type and in any scope
  - Always has to be initialized
    - const int x = 10;
  - Macro (pre-processor) alternative: # define PI 3.14159
    - Use ***const*** whenever possible
- More Complex Macros:

```
# define GS_ABS(x) ((x)>0? (x):-(x))
# define GS_TODEG(r) (180.0f*float(r)/gspi)
# define GS_MIN(a,b) ((a)<(b)? (a):(b))
# define GS_MIN3(a,b,c) ((a)<(b)?
        ((a)<(c)?(a):(c)):((b)<(c)?(b):(c)))
```

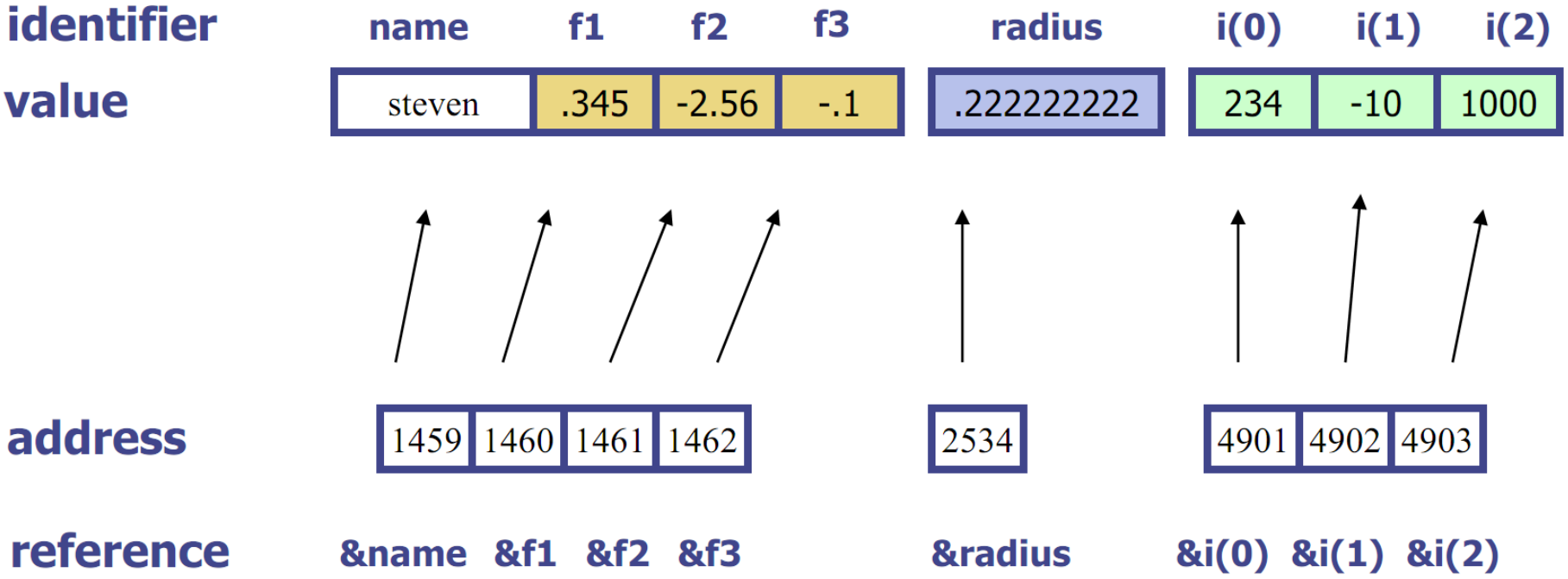
CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 4 – C in C++

Data types: pointers and references

- Pointers are variables containing a memory address
- Every variable, object, and function has an address
- References are introduced in C++ as a new way to work with the address of a variable
  - Avoids the sometime heavy syntax needed to work with pointers, and allowing the same kind of functionality
- Contrary to pointers, references are always valid
  - pointer can be “null” or be of void type (void* pt), references cannot
  - Can’t be changed to reference a different variable

Memory: address and value



Reference and De-reference

type& reference declaration

type* pointer declaration

& reference operator: "address of"

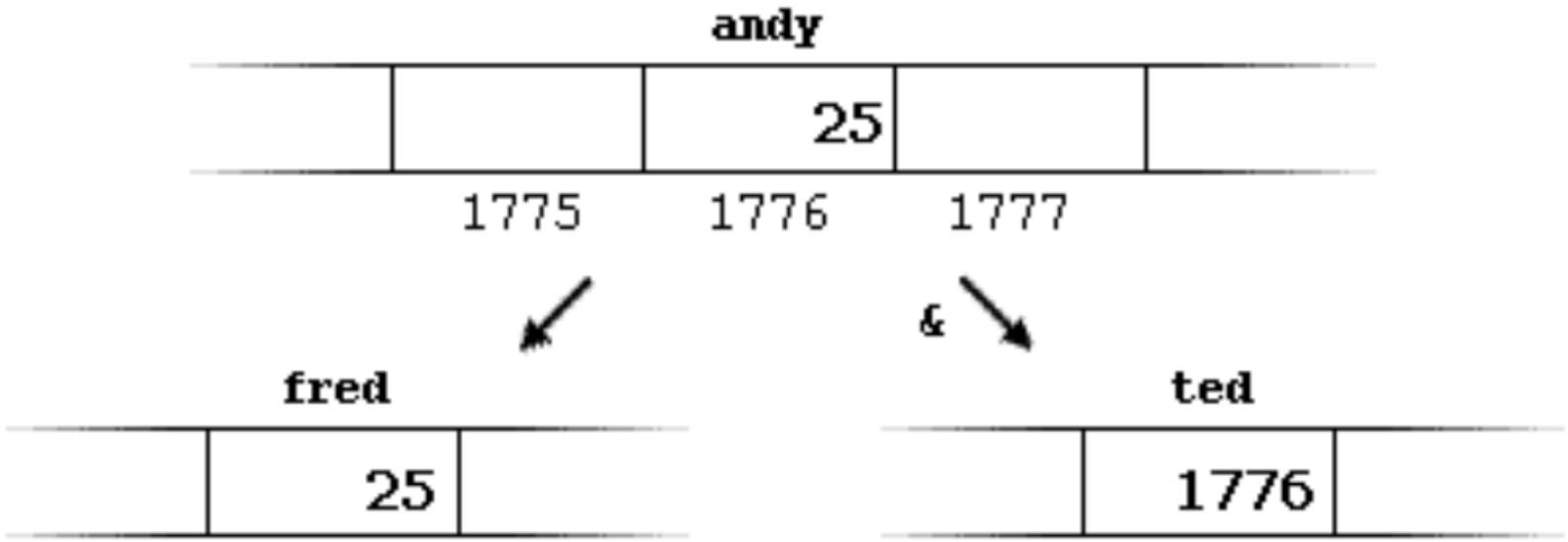
* dereference operator: "value pointed by"

Equivalent to:

&:	Address of	John	is 52 Main Street
*:	Value at	52 Main Street	is John

Reference operator

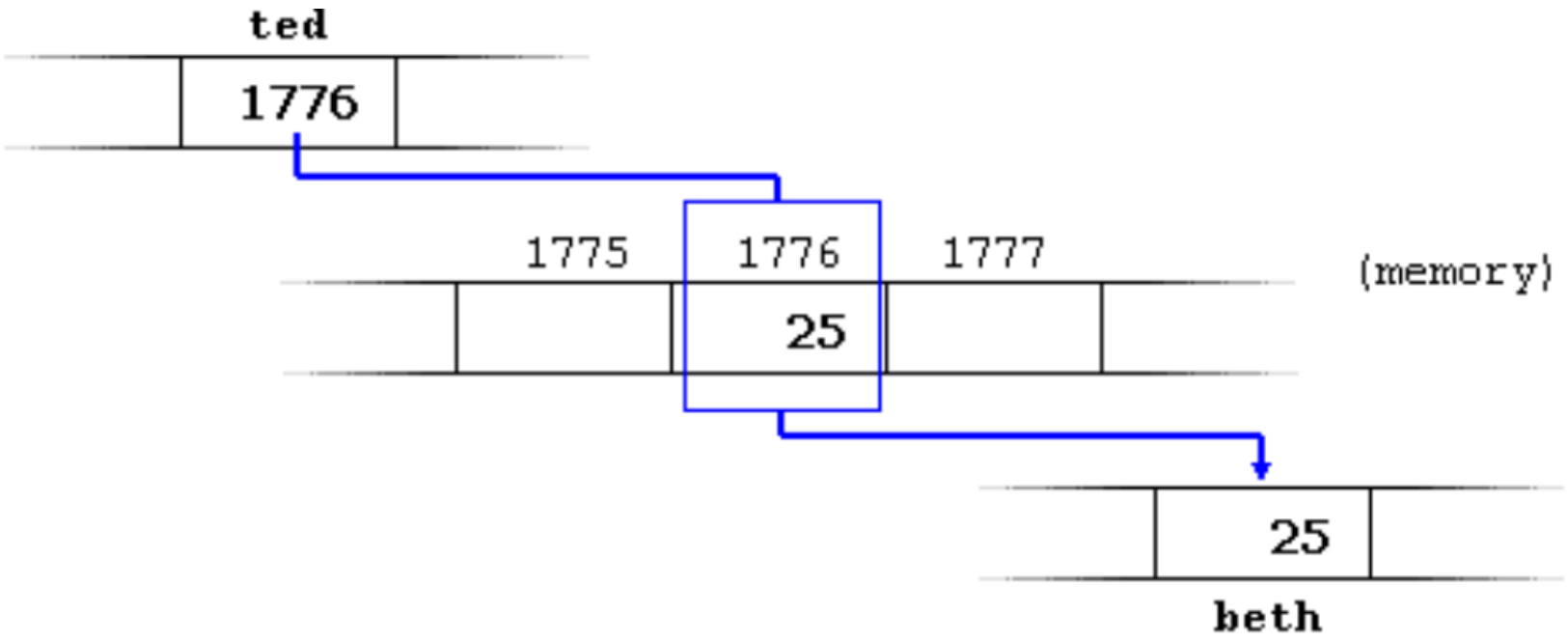
andy = 25; // andy contains 25
fred = andy; // fred contains the value of andy (25)
ted = &andy; // ted contains the address of andy (1776)



De-reference operator

beth = ted; // beth equal to ted (1776)

beth = *ted; // beth equal to value pointed by ted (25)



Variables of pointer type

- Declaration:
  - type * name;
  - type* name;
  - type *name;

- Examples:
  - int * address; // address is a pointer of type int
  - char *p_ch; // p_ch is a pointer of type char
  - float* p; // p is a pointer of type float

Pointer initialization

- int number = 1000;
- int* p_number = &number;
- cout << p_number
- cout << *p_number

Variables and pointers

- Declaration:
  - type * name1, * name2; // declares 2 pointers type*
  - type * name1, name2;   // declares 1 pointer type and 1 variable of type
- Definition:
  - int *p_i; int i, j;
  - i = 10;
  - p_i = &i;
  - *p_i = 20;
  - p_i = &j;
  - *p_i = 10;

Reference example

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

int dog, cat, bird, fish;

void f(int pet)
{
   cout << "pet id number: " << pet << endl;
}

int main()
{
   int i, j, k;
   cout << "f(): " << (long)&f << endl;
   cout << "dog: " << (long)&dog << endl;
   cout << "cat: " << (long)&cat << endl;
   cout << "bird: " << (long)&bird << endl;
   cout << "fish: " << (long)&fish << endl;
   cout << "i: " << (long)&i << endl;
   cout << "j: " << (long)&j << endl;
   cout << "k: " << (long)&k << endl;
}
```

Result:

```
f(): 4198736
dog: 4323632
cat: 4323636
bird: 4323640
fish: 4323644
i: 6684160
j: 6684156
k: 6684152
```

Reference examples

```
//: C03:PassReference.cpp
#include <iostream>
using namespace std;
void f ( int& r ) // Accepting reference
{
  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, but is actually
pass by reference
  cout << "x = " << x << endl;
}
```

Result:

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

Wake up

- https://youtu.be/hVr1YI6x-eQ

Pointers vs References

- A pointer is a variable that stores the address of another variable
  - Can be null
  - Can be changed
  - Uses *
- References refers to another variable
  - Another name of an existing variable
  - Cannot be null
  - Cannot be changed
  - Uses &

Types: operators

- Mathematical operators:
  - addition (+), subtraction (-), division (/), multiplication (*)
    - integer division truncates the result (it doesn't round)
  - modulus (%; remainder from integer division)
    - cannot be used with floating-point numbers
  - Assignment operators: +=, -=, *=, /=, etc
- Logical operators
  - **and** (&&), **or** (||) produce *true* or *false*
  - in C and C++ a statement is *true* if it has a non-zero value, and *false* if it has a value of zero
  - Negation operator not (!)
  - == comparison operator
    - different from assignment op. (=) !

Types: operators

- Bitwise operators
  - bitwise and (&)
    - 1 if both input bits are 1; otherwise 0
  - bitwise or (|)
    - 1 if either input bits are 1; 0 only if both are 0
  - bitwise exclusive or, or xor (^)
    - 1 if an input bit is one, but not both; otherwise 0
  - bitwise not (~)
    - unary operator that inverts the input bit
- Shift operators
  - left-shift operator (<<) and right-shift operator (>>)
    - Also valid: <<= and >>=
  - One bit is always lost in a shift operation
- Ternary operator “? :” (a? b:c)

Types: casting

- Types can be converted by C-like type-casts in parenthesis.

```
//: C03:FunctionCallCast.cpp
int main() {
  float a = float(200);
  // This is equivalent to:
  float b = (float)200;
}
```

- In C++ we should use:
  - **static_cast**: simple casts for type conversion
  - **const_cast**: to cast away the constness of variables
  - **reinterpret_cast**: to cast an object to something completely different
  - **dynamic_cast**: type-safe cast with run-time checking, can be used only with pointers and references to objects (will discuss it with inheritance)

Types: static_cast

```
//: C03:static_cast.cpp
void func(int) {}

int main() {
  int i = 0x7fff; // Max pos value = 32767
  long l;
  float f;
  // 1a) Typical castless conversions:
  l = i;
  f = i; // may generate a warning

  // 1b) C-style type casts:
  l = (long)i;
  f = (float)i; // no warning

  // 1c) C++ way:
  l = static_cast<long>(i);
  f = static_cast<float>(i);
```

Types: static_cast

```
//: C03:static_cast.cpp (continuation)

// 2a) Automatic narrowing conversions:
i = f; // May lose digits (will generate a warning)

// 2b) Says "I know," eliminates warnings:
i = (int)(l); // C style
i = (int)(f); // C style

// 2c) C++ way:
i = static_cast<int>(l); // C++ style
i = static_cast<int>(f); // C++ style
char c = static_cast<char>(i);

// 3a) Forcing a conversion from void* :
void* vp = &i;
// 3b) Old way produces a “dangerous” conversion:
float* fp = (float*)vp;
// 3c) The new way is equally dangerous:
fp = static_cast<float*>(vp);
// etc
}
```

Types: const_cast

```
//: C03:const_cast.cpp
int main() {
//1) const values should not be modified:
const int i = 0;

//2) But we can get a pointer to them and later modify them…:
int* j = (int*)&i;          // Deprecated form
j  = const_cast<int*>(&i); // Preferred
*j = 1;
//3) Can't do simultaneous additional casting:
//! long* l = const_cast<long*>(&i); // Error
}
```

Types: reinterpret_cast

```
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
  for ( int i = 0; i < sz; i++)  cout << x->a[i] << ' ';
  cout << endl << "--------------------" << endl;
}

int main() {
  X x;
  print(&x);
  int* xp = reinterpret_cast<int*>(&x); // Cast to integer
  for ( int* i = xp; i < xp + sz; i++ )
        *i = 0;

  // Can't use xp as an X* unless you cast it back:
  print(reinterpret_cast<X*>(xp));
}
```

Types: reinterpret_cast

```
class BaseClass { ... };

class Class1 : public BaseClass {...}; // BaseClass "derives" Class1

class Class2 {...} ; // BaseClass does not derive Class2

BaseClass  *pb; // pointer to BaseClass
Class1     *p1; // pointer to Class1

p1 = static_cast<Class1*>(pb); // Ok as long as we know pb can point to Class1
p1 = (Class1*)(pb); // C-style also ok, same as static_cast<>

Class2     *p2; // pointer to Class2
p2 = static_cast<Class2*>(pb); // Compiler error, can't convert
p2 = (Class2*)(pb);            // No compiler error…
                               // Same as reiterpret_cast<>

p2 = reinterpret_cast<Class2*>(pb); // No compiler error.
```

A reinterpret_cast<> can only be justified in rare situations. When you use a reinterpret_cast<> you tell the compiler that you need an unusual type of casting and that you know what you are doing…

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 5 – C in C++

Types: casting

- Types can be converted by C-like type-casts in parenthesis.

```
//: C03:FunctionCallCast.cpp
int main() {
  float a = float(200);
  // This is equivalent to:
  float b = (float)200;
}
```

- In C++ we should use:
  - **static_cast**: simple casts for type conversion etc
  - **const_cast**: to remove const
  - **reinterpret_cast**: to cast an object to something completely different (only justified in rare circumstances)
  - **dynamic_cast**: type-safe cast with **run-time checking**, can be used only with pointers and references to objects

Data types: creation of new types

- We can define our own data types based on existing data types
  - ***typedef*** *existing_type new_typename*;

```
typedef float coordinate;
typedef char * pointer_char;
typedef double dimension[3];
pointer_char p_char; // instead of char * p_char;
coordinate x;  // instead of float x;
coordinate y;  // instead of float y;
dimension a, b, c, d;  // instead of double a[3], b[3], …
```

Data types: structure

- Structure is a group of elements under one name
- Elements can be of different data type
- Data elements are called “members”
- Structure is defined for the rest of the program

```
struct structure_name
{
    member_type_1  member_name_1;
    member_type_2  member_name_2;
    …
    member_type_n  member_name_n;
} object_names;
```

Structure declaration

```
struct vehicle
{
    string make;
    string model;
    int year;
} car, truck, bike;
```

```
struct fruit
{
    double weight;
    float price;
    bool ripe;
};
fruit apple;
fruit banana, pear;
```

Member access in structure

- We can refer to
  - Whole object: *object_name*
  - Each member: *object_name.member_name*
- Examples
  - bike (vehicle)
  - car.model (string)
  - car.year (int)
  - apple.weight (double)
  - pear (fruit)
  - banana.ripe (bool)

```
struct vehicle
{
     string make;
     string model;
     int year;
} car, truck, bike;
```

```
struct fruit
{
     double weight;
     float price;
     bool ripe;
};
fruit apple;
fruit banana, pear;
```

Structure assignment

- To assign the whole object
  - fruit apple;
  - apple = { 0.22222, 1.75, false };
  - fruit peach = { 2./3., 2.50, true };
- To assign individual members
  - vehicle car;
  - car.make = “Acura”;
  - car.model = “NSX”;
  - car.year = 2017;

```
struct fruit
{
     double weight;
     float price;
     bool ripe;
};
```

```
struct vehicle
{
     string make;
     string model;
     int year;
};
```

Pointers to structures

- Accessing from pointers
  - variable.member
  - pointer->member
- Declaration
  - vehicle car;
  - vehicle * p_car;
  - p_car = &car;

Pointers to structures

- To access members using a pointer
  - (*pointer).member (*p_car).model = “NSX”;
  - pointer->member p_car->model = “NSX”;

```
p_car->make = “Fiat”;
(*p_car).model = “500”;
fruit * p_apple = &apple;
fruit * p_peach = &peach;
p_apple->weight = p_peach->weight;
(*p_apple).weight = p_peach->weight;
(*p_apple).weight = (*p_peach).weight;
```

Pointers to structures

- (*pointer).member ≠ *pointer.member

- *object.p_member = *(object.p_member)
  - The member of an object is a pointer
  - See operator precedence: http://www.cplusplus.com/doc/tutorial/operators/

Pointers to structures example

```
//: C03:SimpleStruct3.cpp
// Using pointers to structs
typedef struct Structure3 {  // skip typedef in C++
  char c;
  int i;
  float f;
  double d;
};

int main() {
  Structure3 s1;
  Structure3* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
}
```

Data types: enumerations

- Allows us to create new data types with predefined values.

```
enum Mood { HAPPY, SLEEPY, SAD, ANGRY }; //0,1,2,3
Mood myMood = SLEEPY;

enum Color { RED=1, GREEN=10, BLU=100 };
Color myColor;
myColor = BLU;
```

Data types: enumerations

```
//: C03:Enum.cpp
   // Keeping track of shapes

   enum ShapeType {
     circle,
     square,
     rectangle
   };  // Must end with a semicolon like a struct

   int main() {
     ShapeType shape = circle;
     // Activities here....
     // Now do something based on what the shape is:
     switch (shape) {
       case circle:  /* circle stuff */ break;
       case square:  /* square stuff */ break;
       case rectangle:  /* rectangle stuff */ break;
     }
   }
```

Data types: enumerations

- Enumerators offer type checking and allow for more intuitive types than ints

```
enum ShapeType { circle=10, square=20, rectangle=50 };

enum snap { crackle=25, pop=10 };

void draw ( ShapeType t )
{
    // call drawing commands according to type here
}

void main ()
{
    snap s = pop;
    draw ( s ); // will generate warning or error
}
```

Wake up

- https://youtu.be/BGLsUxe2pik

Data types: unions

- Unions allow one location of memory to be accessed as different data types
- The size of a union is the largest of its members
- Modification of one of the members will affect the value of all members
- It is not possible to store different values for individual members

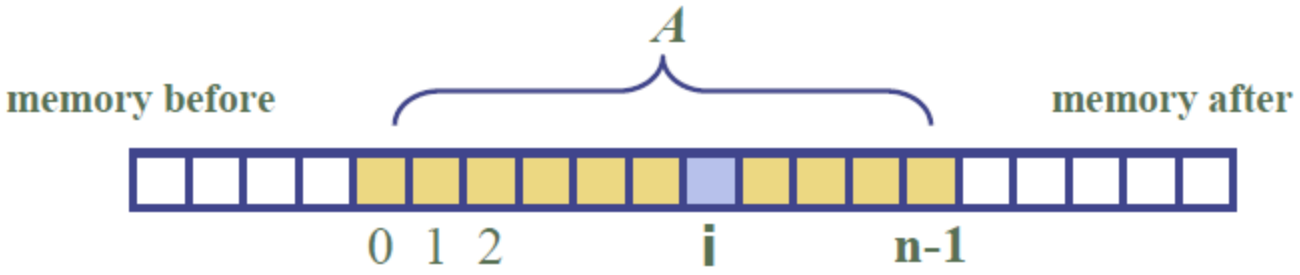
Data types: unions

```
//: C03:Union.cpp
#include <iostream>
using namespace std;

union Packed { // Declaration similar to a struct or class
  char i;      // The union will be the size of a
  short j;     // double, since that's the largest element
  int k;
  long l;
  float f;
  double d;
};

int main() {
  cout << "sizeof(Packed) = " << sizeof(Packed) << endl;
  Packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
}
```

Data types: array

- **Array**: collection of elements of the same type
- **Array size**: how many elements—integer number **n**

```
int a[10];          // a[0], ….. a[9]

int n=10;
int A[n];           // A[0], ….. A[n-1]
```



Array—Types and initialization

```
char c[4];

short s[5];

int y[2];

long l[7];

float f[10000];

double x[n];

bool b[2];
```

```
char c[] = { 'a', 'b', 'c', 'd' };

int a[5] = { 4, 3, 2, 1, 0 };

int n = 10;

double x[n];

int i = 3;

x[0] = 2.1;

x[i] = .1e5;

x[4] = i / 2;

x[a[i]] = a[i];

x[i] = i;

bool b[2] = {false, true};
```

Data types: arrays

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++) {
    a[i] = i * 10;
    cout << "a[" << i << "] = " << a[i] << endl;
  }
}
```

Data types: arrays

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

struct Point3D
{
  int x, y, z;
};

int main()
{
  Point3D p[10];
  for(int i = 0; i < 10; i++)
  {
    p[i].x = i + 1;
    p[i].y = i + 2;
    p[i].z = i + 3;
    cout << p[i].x << ", " << p[i].y << ", "<< p[i].z << endl;
  }
}
```

Pointers and arrays

- Identifier of an array equivalent to the address of array's first element
- The i-th element of ***a*** can be dereferenced either way:
  - a[i]
  - *(a+i)
- Example:
  - int a[20]; // a is a pointer to a[0]
  - a[5] = 0; // a [offset of 5] = 0
  - *(a+5) = 0; // pointed by (a+5) = 0

Pointer and arrays

- Examples:
  - int numbers[20];
  - int * p;
  - p = numbers;

  - double d[] = {0., 1., 2.} ;
  - double* p_var = d; //points to array d
  - double* p_array[3]; //array of 3 pointers
  - p_array[0] = &d[0]; //1st pointer is the address of d[0]

Pointer and arrays

```
//: C03:PointersAndBrackets.cpp (extended)
int main() {
  int a[10];

  int* ip = a; // get pointer to beginning of a
  for (int i = 0; i < 10; i++)
    ip[i] = i * 100; // access positions from pointer location

  ip = &a[0]; // another way of getting pointer to beginning of a
  for (int i = 0; i < 10; i++)
    *ip++ = i * 100; // here the pointer is incremented each time

  for (int i = 0; i < 10; i++)
    cout<<a[i]<<" ";
  cout<<endl;
}
```

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 6 – Data Abstraction

Announcement

- Reading assignment
  - Ch. 4
  - http://www.cplusplus.com/doc/tutorial/classes/

Data Abstraction

- What is the main point behind “data abstraction”?

Wikipedia says:

*“In computer science, **abstraction** is the process by which data and programs are defined with **a representation similar in form to its meaning** (semantics), while **hiding away the implementation details**. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. A system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the computer hardware where the program is run, while high-level layers deal with the business logic of the program.”*

Simplify data access and use:

Hide details and **design appropriate manipulation interface**

Object oriented concepts

- Encapsulation
  - The ability to package data with functions
  - Variables are encapsulated in a class/structure with member functions (methods)
- Implementation hiding
  - Access control
  - To prevent important data from being corrupted
- Interface
  - It establishes what requests you can make for a particular object
  - It is an abstraction of an object
  - Tells what an object does without the details (i.e. header files)

Structures in C vs C++

- C Structures
  - Cannot have member functions inside structure
  - Cannot directly initialize member variables
- C++ Structures
  - Can have member functions (methods) in structure
  - Can initialize member variables
  - Almost the same as a class with one little difference:
    - Structures default to public visibility and classes to private
  - **Convention is to use Structs for just data and classes for data and methods**

Libraries

- Code that someone else has written and packed together
- We can utilize libraries to increase our productivity
- Consists of a library file and header files
- To be able to use libraries efficiently, we must understand how libraries work

C-Like Stash library interface

```
// Header file for an array-like class

typedef struct CStashTag { // (recall the typedef is only needed in C)
  int size;      // Size (bytes) of each entry
  int quantity;  // Number of storage spaces (entries allocated)
  int next;      // Next empty space (equal to the number of elements)
  unsigned char* storage; // Dynamically allocated array of bytes
} CStash;

// Common C-like function naming style to avoid name clashes:

void  cstash_initialize ( CStash* s, int size );
void  cstash_cleanup ( CStash* s );
int   cstash_add ( CStash* s, const void* element );
void* cstash_fetch ( CStash* s, int index );
int   cstash_count ( CStash* s );
void  cstash_inflate ( CStash* s, int increase );
```

C-Like Stash class

```
//: C04:CLib.cpp {O}
// Implementation of example C-like library
// Declare structure and functions:
#include "CLib.h"         // Include the class header file
#include <iostream>
#include <cassert>
using namespace std;

// Quantity of elements to add when increasing storage:
const int increment = 100;

void initialize ( CStash* s, int size ) {
  s->size = size;  //Unit size of element in bytes
  s->quantity = 0;
  s->storage = 0;
  s->next = 0;
}
```

C-Like Stash class

```
//: C04:CLib.cpp - continue

int add ( CStash* s, const void* element ) {
  if ( s->next >= s->quantity ) // Not enough space left
    inflate(s, increment); // Inflate the stash

  // Copy element into storage, starting at next empty space:
  int startBytes = s->next * s->size;  // Locate next available position in
                                       // storage
  unsigned char* e = (unsigned char*)element; // Cast element from
                                              // type void to unsigned char
  for ( int i=0; i < s->size; i++ )
    s->storage[startBytes + i] = e[i]; // Copy character by character

  s->next++; // Update next available index
  return(s->next - 1); // Index number of last entry
}
```

C-Like Stash class

```
//: C04:CLib.cpp - continue

  void inflate(CStash* s, int increase) {
    assert(increase > 0); // Make sure expansion is positive

    int newQuantity = s->quantity + increase;
    int newBytes = newQuantity * s->size; // Total memory in bytes
    int oldBytes = s->quantity * s->size; // Total memory in bytes

    unsigned char* b = new unsigned char[newBytes]; // New array
    for(int i = 0; i < oldBytes; i++)
      b[i] = s->storage[i]; // Copy old to new

    delete [](s->storage); // Old storage

    s->storage = b; // Point to new memory
    s->quantity = newQuantity;
  }
```

C-Like Stash class

```
//: C04:CLib.cpp - continue

  void* fetch(CStash* s, int index) {
    // Check index boundaries:
    assert(0 <= index);
    if(index >= s->next)
      return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(s->storage[index * s->size]);
  }

  int count(CStash* s) {
    return s->next;  // Number of elements in CStash
  }

  void cleanup (CStash* s) {
    if ( s->storage!=0) {
      cout << "freeing storage" << endl;
      delete []s->storage;
    }
  }
```

Dynamic Storage allocation

- Heap memory
  - Memory set aside by the program during runtime
- In C:
  - malloc, calloc, realloc, free
- In C++:
  - new, delete

Dynamic Storage allocation

- pointer = new type
- pointer = new type [number_of_elements]
- Examples:
  - double* p_variable;
  - p_variable = new double;

  - int * a;
  - a = new int [5];

  - vehicle * p_vehicle;
  - p_vehicle = new vehicle;

Dynamic Storage allocation

- delete pointer;
- delete [] pointer;
- Examples:
  - delete p_variable;
  - delete [] a;
  - delete p_vehicle;

Using the C-Like Stash class

```
//C04:CLibTest.cpp (simplified)
int main() {

//1. Define variables at the beginning of the block, as in C:
CStash stash;

//2. Now remember to initialize our object:
initialize ( &stash, sizeof(int) );

//3. Now let's add some elements:
for (int i = 0; i < 100; i++)
  add ( &stash, &i );

//4. Now let's print the contents:
for(int i = 0; i < count(&stash); i++)
  cout << * ((int*)fetch(&stash, i)) << endl; // Cast from void* to int*

cleanup(&stash);
}
```

Using the C-Like Stash class

- Difficulties:
  - Manipulation of void pointers
  - Many type conversions needed
  - Long naming conventions: functions from different classes cannot have the same name
  - Explicit initialization and cleanup calls needed
  - Syntax long and sometimes not trivial
- Let’s now re-write the same class in C++

C++ Stash class

```
//: C04:CppLib.h
// C-like library converted to C++

struct Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  unsigned char* storage; // Dynamically allocated array of bytes

  // Methods
  void initialize(int size);
  void cleanup();
  int add(const void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
};
```

C++ Stash class

```
//: C04:CppLib.cpp {O}
// C library converted to C++
// Declare structure and functions:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add when increasing storage:
const int increment = 100;

void Stash::initialize(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

void Stash::cleanup() {
  // There is no need to test if(storage!=0),
  // operator delete will already make the test.
  delete []storage;
}
```

C++ Stash class

```
int Stash::add(const void* element) {
  if(next >= quantity) // Not enough space left
    inflate(increment);

  // Copy element into storage, starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];

  next++;
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  // Check index boundaries:
  assert(0 <= index);
  if(index >= next)
    return 0; // To indicate the end
  // Produce pointer to desired element:
  return &(storage[index * size]);
```

C++ Stash class

```
int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  assert(increase > 0);
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;

  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new

  delete []storage; // Old storage
  storage = b; // Point to new memory
  quantity = newQuantity;
}
```

C++ class

- Functions are now inside the structure and are called “member functions”
- No need to pass the stash address to each function
- No need to name the functions explicitly
- Functions have to be declared (usually in the header file) before they can be called
- You can access the member variables without referring to the structure

Using the C++ Stash class

- Variables can be defined at any point in the scope
- Member functions and variables are selected using (.) operators

```
int main() {

Stash stash;

stash.initialize ( sizeof(int) );

//let's add some elements:
for (i = 0; i < 100; i++)
  stash.add ( &i );

//4. Now let's print the contents:
for(i = 0; i < stash.count(); i++)
  cout << * ((int*)stash.fetch(i)) << endl;

stash.cleanup();
}
```

CSE 165/ENGR 140 Intro to Object Orient Programming

Lecture 7 – Data Abstraction

Announcement

- Quiz #1 on 2/15 (Tuesday) during lecture
- Reading assignment
  - Ch. 5 & 6
  - http://www.cplusplus.com/doc/tutorial/classes/

Recap

- Libraries
  - Class interface (header)
  - Class source code (cpp)
  - Difference between C and C++: C++ is object oriented and has:
    - Encapsulation (Binding data with functions into a class)
    - Data Abstraction (Implementation hiding with an interface)
    - Later we will talk about inheritance and polymorphism

Data Abstraction

- What is the main point behind “data abstraction”?

Wikipedia says:

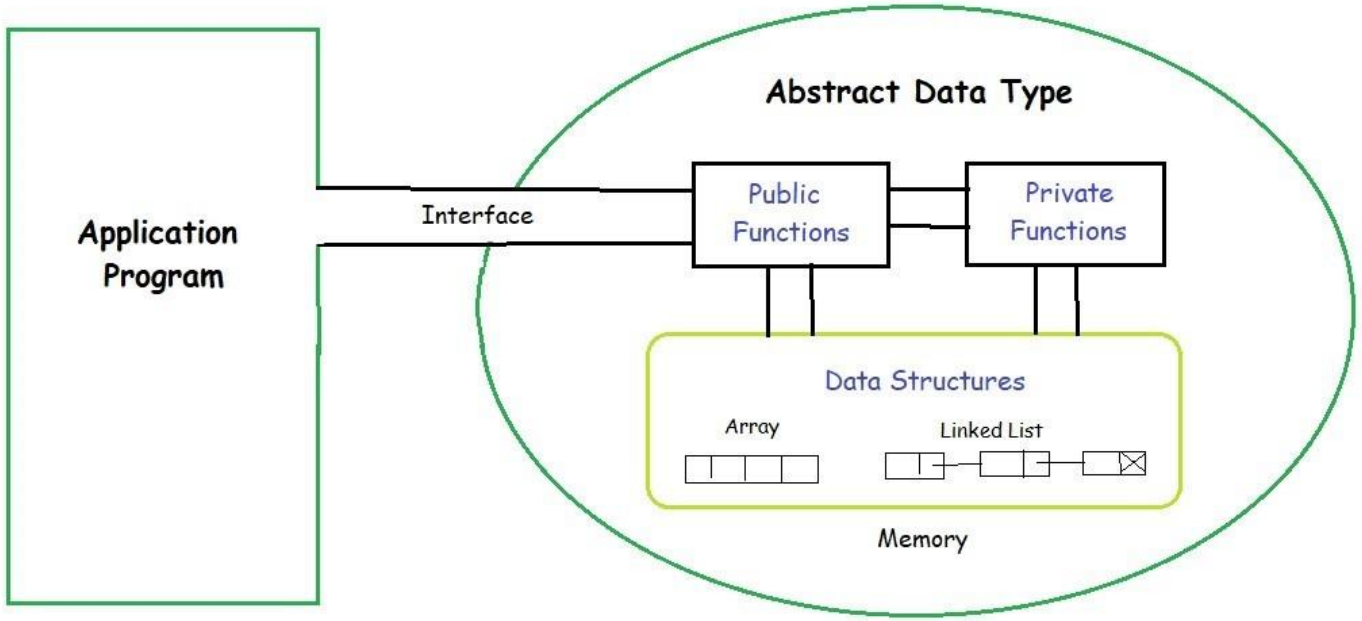
*“In computer science, **abstraction** is the process by which data and programs are defined with **a representation similar in form to its meaning** (semantics), while **hiding away the implementation details**. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. A system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the computer hardware where the program is run, while high-level layers deal with the business logic of the program.”*

Simplify data access and use:

Hide details and **design appropriate interface**

Abstract data type (ADT)

- Called “abstract” because it gives an implementation-independent view
- Think of ADT as a black box which hides the inner structure and design of the data type



Abstract data type (ADT)

- We can create new data type by packaging data with functions (using encapsulation)
  - Stash creates a new data type using array
  - Stash has functions to control data (add, fetch, inflate, etc.)
- There are many ADT in the Standard Template Library (STL) of C++
  - Vectors, lists, stacks, queues, etc.

Size of a struct

```
//: C04:Sizeof.cpp
// Sizes of structs
#include <iostream>
using namespace std;

struct A {
   int i[100];
};

struct B {
   void f();
};

void B::f() {}

int main() {
   cout << "sizeof struct A = " << sizeof(A) << " bytes" << endl;
   cout << "sizeof struct B = " << sizeof(B) << " bytes" << endl;
}
```

Output:

sizeof struct A = 400 bytes

sizeof struct B = 1 bytes

Nested structures

```
struct movies {
    string title;
    int year;
};
```

```
struct friends {
    string name;
    string email;
    movies favorite_movie;
} charlie, maria
```

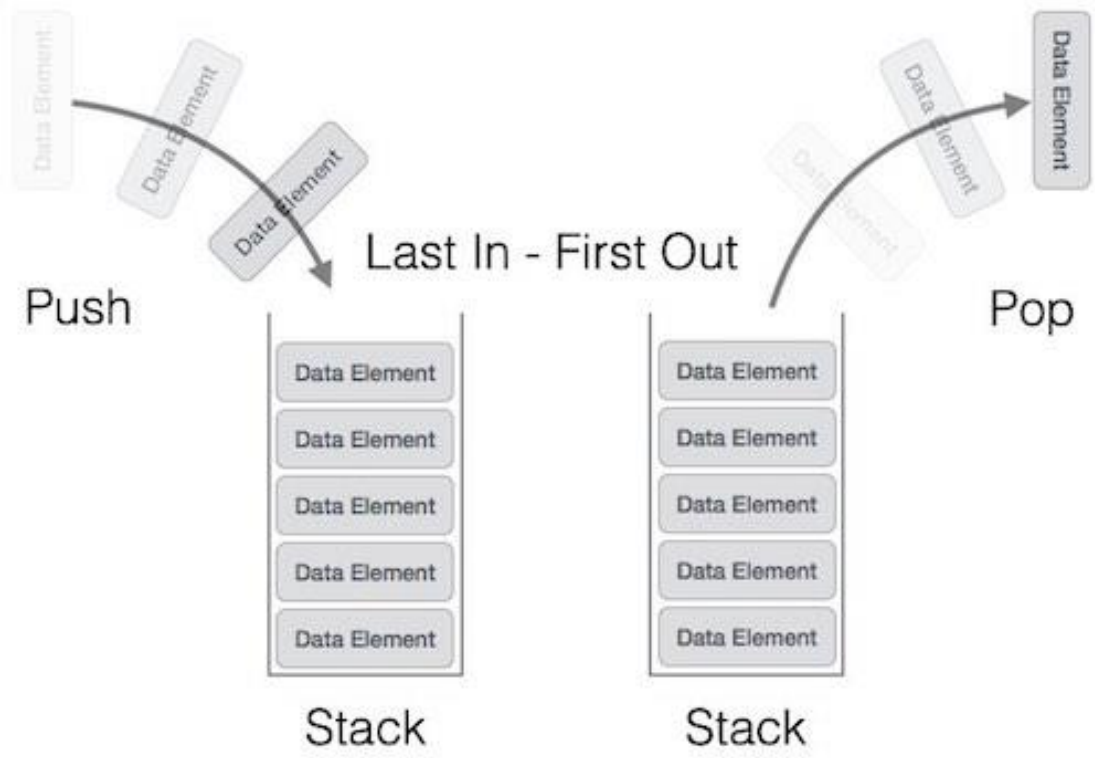
```
friends * p_friends = &charlie;
charlie.name = …
maria.favorite_movie.title = …
charlie.favorite_movie.year = …
p_friends->favorite_movie.title = …
```

Nested structures: Stack example

- What is a ***stack*** of items?
- How are the items in a stack organized?
  - Where do you put a new item (insertion) in a stack?
  - From where do you remove an item (deletion) in a stack?
  - Last-in first-out

Nested structures: Stack example

- We can also organize our data in stacks.



Nested structures: Stack example

- Why do we want to use stacks?
  - History of web browser
  - Un-do function of a text editor
  - Matching “{}” or “()” in a cpp editor

Nested structures: Stack example

```
//: C04:Stack.h
// Nested struct in linked list
#ifndef STACK_H
#define STACK_H

struct Stack {

   struct Link {
      void* data;
      Link* next;
      void initialize(void* dat, Link* nxt);
   }* head;

   void initialize();
   void push(void* dat);
   void* peek();
   void* pop();
   void cleanup();
};

#endif // STACK_H
```

Stack class

```
//: C04:Stack.cpp {O}
#include "Stack.h"
#include "../require.h"
using namespace std;

//Stack::Link has only one method:
void Stack::Link::initialize(void* dat, Link* nxt) {
   data = dat;
   next = nxt;
}
```

Stack class

```
// Stack methods:
void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
   Link* newLink = new Link;
   newLink->initialize(dat, head);
   head = newLink;
}

void* Stack::peek() {
   require(head != 0, "Stack empty");
   return head->data;
}

void* Stack::pop() {
   if(head == 0) return 0;
   void* result = head->data;
   Link* oldHead = head;
   head = head->next;
   delete oldHead;
   return result;
}
```

Stack class

```
// Stack methods:
void Stack::cleanup() {
   // This implementation does not do anything, it just
   // requires the stack to be empty:
    require(head == 0, "Stack not empty");
}

void Stack::cleanup_notused() {
   // We could do something like empty the stack, BUT
   // we do not know the type of the objects stored in the
   // stack, so we cannot free them…
   while ( pop() ); // works, but may create memory leak…
   require(head == 0, "Stack not empty");
}
```

Using stack

```
//: C04:StackTest.cpp
using namespace std;

int main(int argc, char* argv[]) { //Run the program with input arguments
   ifstream in(argv[1]); //input argument as file name
   Stack textlines;
   textlines.initialize();
   string line;

   // Read file and store lines in the Stack:
   while(getline(in, line))
      textlines.push(new string(line));

   // Pop the lines from the Stack and print them:
   string* s;
   while((s = (string*)textlines.pop()) != 0) {
      cout << *s << endl;
      delete s;
   }
   textlines.cleanup();
}
```

Global scope resolution operator

- We use the global scope resolution operator (::, with nothing in front of it) to select a global identifier.

```
//: C04:Scoperes.cpp
// Global scope resolution

   int a;       // 1. Variable a is in the global scope
   void f() {} // 2. Function f is in the global scope

   struct S {
      int a;     // 3. a is a member of S, its global scope is S::a
      void f(); // 4. f is a member of S, its global scope is S::f
   };

   void S::f() {
      ::f();     // 5. Would be recursive otherwise!
      ::a++;     // 6. Select the global a
      a--;       // 7. The a at struct scope
   }
   int main() { S s; f(); s.f(); }
```

Access Control

- Access control is defined with keywords:
  - private: accessible only by original/base class
  - protected: accessible by base and derived (inheritance) classes
  - public: accessible by everyone

```
struct A {
      float val; // in a struct, members are public by default
     private:   // but we may change the access for the next members
      int size; // private members can only be accessed by methods of A
};

 class C {
      float val; // in a class members are private by default
     public :   // but we may change the access to public
      int size; // now this member is private
};
```

Access Control

```
struct A {
     float val; // 1) in a struct members are public by default
    private:
     int size; // 2) private members can only be accessed by methods of A
    protected:
     float x;  // 3) protected members are similar to private,
               // but inherited classes are given full access to them
    public :
     void setSize () {}; // 4) resize represents an interface method to class A
    private :
     void freemem () {}; // 5) this method is used for internal operations only
    protected:
     void inflate () {}; // 6) internal method accessible by derived classes
   };

   int main() {
     A a;
     a.val = 0.1f; // ok
     a.setSize();  // ok
     a.size = 3;   // compilation error (member inaccessible)
     a.inflate();  // compilation error
   }
```

Access Control: friends

- Private members of a class cannot be accessed outside of class.
- Generic functions and classes can be declared to be a “friend” and gain access to private members.
- Within the class, precede function declaration with keyword ***friend***.

Access Control: friends

```
//: C05:Friend.cpp

struct X; // incomplete type specification (or forward declaration)
          // needed for the definition of f

struct Y {
  void f(X*);
};

struct X { // Definition
 private:
  int i;
 public:
  void initialize();
  friend void g(X*, int); // Global friend
  friend void Y::f(X*);  // Struct member friend
  friend struct Z; // Entire struct is a friend
  friend void h(); // Another global friend
};
```

Access Control: friends

- A typical use of friend functions is to give access to low-level functions that perform special operations in a class:

```
class MyWindow
 {
    // a low-level OS function needs friend access so that it can control
    // key functionality of our window class: to signal when to draw
    friend void ::sysdraw ( MyWindow* );

  public:
    // send a window redraw request to the OS, ok to be public:
    void redraw();

  private :
    // the draw() function is private because it should only be called by
    // the OS (via sysdraw) when the drawing context is ready to be used:
    void draw ();
 };
```

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 8 – Classes:

Access control, constructors, and destructors

Announcement

- In person on 2/15 in CLSSRM 116
- Quiz on 2/17 in during lecture
- Reading assignment
  - Ch. 14
  - http://www.cplusplus.com/doc/tutorial/inheritance/

Access Control: friends

- Private members of a class cannot be accessed outside of class
- Generic functions and classes can be declared to be a “friend” and gain access to private members
- Within the class, precede function declaration with keyword ***friend***

Access Control: friends

```
//: C05:Friend.cpp

struct X; // incomplete type specification (or forward
declaration)
          // needed for the definition of f

struct Y {
  void f(X*);
};

struct X { // Definition
 private:
  int i;
 public:
  void initialize();
  friend void g(X*, int); // Global friend
  friend void Y::f(X*);  // Struct member friend
  friend struct Z; // Entire struct is a friend
  friend void h(); // Another global friend
};
```

Access Control: friends

- A typical use of friend functions is to give access to low-level functions that perform special operations in a class:

```
class MyWindow
 {
    // a low-level OS function needs friend access so that it can control
    // key functionality of our window class: to signal when to draw
    friend void ::sysdraw ( MyWindow* );

   public:
    // send a window redraw request to the OS, ok to be public:
    void redraw();

   private :
    // the draw() function is private because it should only be called by
    // the OS (via sysdraw) when the drawing context is ready to be used:
    void draw ();
 };
```

Stash class with access control

```
//: C05:Stash.h
// Converted to use access control
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);

public: // here is the public interface, some coding styles will
        // prefer the interface to appear before data member declarations
  void initialize(int size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};

#endif
```

Hiding implementation from interface

- We may not to want to have our implementation visible to our client
  - Our competitors may be able to obtain it
  - For security reasons: encryption algorithm
  - To prevent others from “cracking” our program

Hiding implementation from interface

```
// Window.h:
class Window {
  struct Internal;  // Forward declaration only
  Internal* intwin; // Put in Internal all the many private data and methods
                    // Internal is only declared in the .cpp
                    // (ok since the size of a pointer is type independent)
public:
   void init ();
   int run ();
};
```

```
// Window.cpp:
struct Window::Internal {
     int i, a, b;
     void readEvents ();
     void wait ();
     ...
};

Window::init () { intwin = new Internal; ...  }
Window::run () { intwin->readEvents(); ... }
```

Wake up

- https://youtu.be/c5n6lnEineQ

Constructors

- A constructor is a special function to initialize objects.
  - Avoid undetermined results.
  - Executed at creation of object.
  - Cannot be called like any other function.
  - No return and no void.

```
Class CRectangle {
        int width, height;
    public:
        CRectangle (int ,int );   // constructor declaration
        int area () {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {    //definition
    width = a;
    height = b;
}
```

Overloading Constructors

- Constructors can be overloaded, like operators and functions.
  - Defined multiple times.
  - For different number of parameters or types.
  - The one called is the one with matching parameters.
  - A constructor without input parameter is called a default constructor.

Constructors

```
class X {
     int i;
   public:
     X()                // Default Constructor
     { i=0; }

     X(int n)           // Alternative constructor
     { i=n; }
   };


   void f() {
     X x1;        // Default constructor called
     X x2(3);     // Alternative constructor called
     ...
   }
```

Destructors

- A destructor is a special function to destroy objects.
  - Release dynamically allocated memory.
  - When an object is created with ***new***, destructor is called upon delete.
  - When an object is created ***locally*** within a function, destructor is called when function returns.

```
~CRectangle () {  // destructor definifion
    delete …;
    …
    delete …;
}
```

Destructors

```
class X {
      int i;
    public:
      X()                 // Default Constructor
      { i=0; }

      X(int n)            // Alternative constructor
      { i=n; }

     ~X() { ... };        // Destructor (only one can exist)
    };


    void f() {
      X x1;       // Default constructor called (x1.i is 0)
      X x2(3);    // "int constructor" called (x2.i is 3)
      ...
    }  // At the end of their scope, a and b destructors will
       // be automatically called!
```

Revisiting the Stash class

```
//: C06:Stash2.h
// With constructors & destructors

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size);   // Constructor takes care of initialization
  ~Stash();          // Destructor
  int add(void* element);
  void* fetch(int index);
  int count();
};
```

Revisiting the Stash class

```
// Constructor:
Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

// Destructor:
Stash::~Stash() {
  if(storage != 0) cout << "freeing storage" << endl;
  delete []storage;
}
```

Aggregate Initialization

- We can initialize an array of any primitive type with aggregate initialization:

```
int a[5] = { 1, 2, 3, 4, 5 }; // 1) each element goes to one "entry" of a

int b[6] = {0}; // 2) all the missing elements of b will be initialized to 0

int b[6];       // 3) none of the elements are initialized

int c[] = { 1, 2, 3, 4 }; // 4) this is also valid, the size of c
                             will be automatically counted from the
                             number of initialized values

sizeof(c)          // 5) will return the size (in Bytes)of the entire array c

size c             // 6) will return the number of entries in the array c
```

Aggregate Initialization

- We can initialize an array of classes with aggregate initialization:

```
struct X {
  int i; float f; char c;
};

X x1 = { 1, 2.2, 'c' }; // 1) initialization of one object
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} }; // 2) initialization of two objects,
                                            //    the 3rd object is
                                            //    initialized to 0


struct Y {
  float f; int i;
  Y(int a);
};

Y y1[] = { Y(1), Y(2), Y(3) }; // initialization using constructor
```

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 9 – Inheritance/Derivation

Annoucement

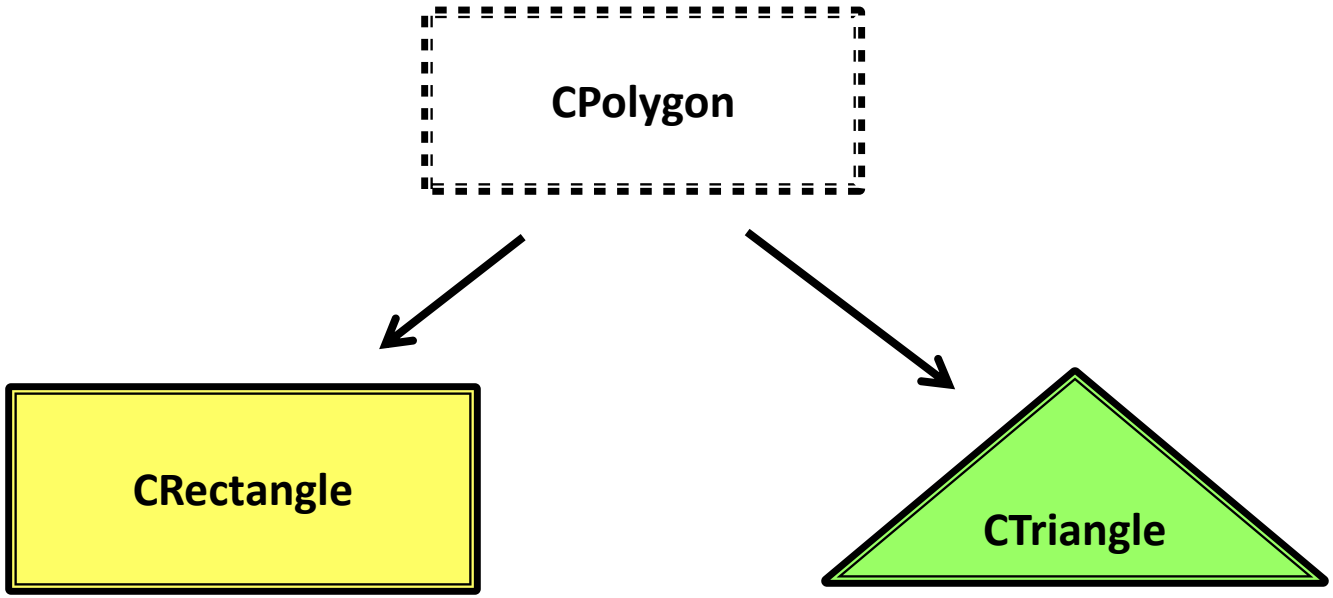
- Quiz next time

Inheritance

- Imagine you need an Object that is slightly different from the existing one.
- Instead of re-designing an entire new object from scratch, you can **inherit** (or **derive**) the existing object and just "add" the needed modifications.
- If Apple inherits Fruit, then
  - Apple is also Fruit.
  - Apple contains all members of Fruit, plus its own members.

Inheritance

- Class derived from another class
  - Base class (or parent class, or superclass)
  - Derived class (or child class, or subclass)



Inheritance

```
class derived_class_name: public base_class_name
{
    ...
};


class CRetangle: public CPolygon
{
    ...
};
```

Inheritance Terminology

- Suppose **class B** inherits from **class A**
- The classes form a part of a **class hierarchy**.
  - B is a ***derived class (subclass, child class)*** of A, class B inherits from class A.
  - A is a ***base class (superclass, parent class)*** of B, class A derives class B.
  - The class immediately above a given class is known as its ***immediate superclass***.
- A class inherits all members of the base class (with exceptions).
  - Includes functions/variables inherited by that class.
  - It can add additional variables and functions.
  - It can override (change) the inherited functions.

What is inherited

- Derived classes inherit all accessible members of base class:
  - Base class ***A*** has member ***a***.
  - Derived class ***B*** has own member ***b***, and also member ***a***.
- Derived class DOES NOT inherit these from base class:
  - Constructor and destructor
  - Overloaded operators
  - Friend functions

Inheritance versus composition

- A class can include members of another class in two ways:

```
class Rect
 { public :
    float xa, ya, xb, yb; // rectangle min/max coordinates
 };

// 1) Example of composition:
class RoundedRect1
 { public :
    Rect r;       // rect is included as a member
    float cornerLen; // how much to round on each corner
 };

// 2) Example of inheritance:
class RoundedRect2 : public Rect // members of Rect are inherited
 { public :
    float cornerLen; // how much to round on each corner
 };

//=> In both examples, the members of RoundedRectX are the same.
```

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 9.5 – Inheritance/Derivation

Quiz

- 10 mins to complete the quiz on CATCourses

Access control for inheritance

```
class Rect
{
    public :
    float xa, ya, xb, yb;
};

class RoundedRect : public Rect // 1) we are deriving with public
                                //    access to Rect's members
{
    public :
    float cornerDist;
};

void main ()
{
   RoundedRect r;
   r.xa = 1.0;              // 2) we have public access to the
                            //    members of the base class
   r.cornerDist = 0.2;      // 3) we also have public access to the
                            //    members of RoundedRect
}
```

Access control

```
class Rect
{
    float xa, ya, xb, yb; // 1) => these members are now private
};

class RoundedRect : public Rect // 2) we are still deriving with
                                //    public access
{
   public :
    float cornerDist;
};

void main ()
{
   RoundedRect r;
   r.xa = 1.0;             // 3) error: public derivation will not
                           //    break access control of the base class
   r.cornerDist = 0.2;
}
```

Access control

```
class Rect
{
   protected :
    float xa, ya, xb, yb; // 1) these members are now protected
};

class RoundedRect : public Rect // 2) we are still deriving with
                                //    public access
{
   public :
    float cornerDist;
    void setXa ( float f ) { xa = f; } // 3) => set gives access to xa
};

void main ()
 {
   RoundedRect r;
   r.xa = 1.0;            // 4) error: public derivation will not
                          //    break access control of the base class
   r.setXa ( 1.0 );       // 5) ok
   r. cornerDist = 0.2;
 }
```

Inheritance Protection

class derived_class_name: **public** base_class_name {…}

public members of base are **public** in derived

protected members of base are **protected** in derived

class derived_class_name: **protected** base_class_name {...}

public members of base are **protected** in derived

protected members of base are **protected** in derived

class derived_class_name: **private** base_class_name {...}

public members of base are **private** in derived

protected members of base are **private** in derived

Access control: private derivation

```
class X
{
   private:        int privx;
   protected:      int protx;
   public :        int publx;
};

class Y : private X // private derivation
{
   public :
    void protset ( int i ) { protx=i; } // 2) Ok
    void publset ( int i ) { publx=i; } // 3) Ok
};

void main ()
{
   Y y;
   y.privx=1; // 4) error
   y.protx=2; // 5) error
   y.publx=3; // 6) error
}
```

Everything in X is private through Y now!

Inheritance access matrix

Access	Public	Protected	Private
Same class member	Yes	Yes	Yes
Derived class member	Yes	Yes	No
Non-member	Yes	No	No

Constructors of Derived Classes

```
// Example of typical constructors in a class:
class Rect
{
   public :
    float x, y, w, h; // rectangle upper-left corner (x,y) and size (w,h)

    Rect () { x=y=w=h=0; }  // 1) Default constructor declared in-line

    Rect ( const Rect& r )  // 2) Copy constructor, takes in an object of same type
     { x=r.x; y=r.y; w=r.w; h=r.h; }

    Rect ( float rx, float ry, float rw, float rh ) // 3) Another constructor
     { x=rx; y=ry; w=rw; h=rh; }
};
```

Constructors of Derived Classes

```
// Constructors in a derived class must call
// the correct constructors of the base class:
class RoundedRect : public Rect
 {
   public :
    float cornerLen; // how much to round on each corner

    RoundedRect () { cornerLen=0; }      // 1) Default constructor of base class
                                            automatically called

    RoundedRect ( const RoundedRect& r ) // 2) Copy Constructor declaration
     :Rect(r)                            // 3) Calling copy constructor of Rect
     { cornerLen=r.cornerLen; }


    RoundedRect ( float rx, float ry, float rw, float rh, float len )
     :Rect(rx,ry,rw,rh),  // 4) Calling constructor of base class
      cornerLen(len)      // 5) Calling float "pseudo-constructor"
      { }
 };
```

Constructors of Derived Classes

- The parenthesis syntax for constructors can be used in several ways:

```
// 1) Example of "pseudo-constructors" :
int i(100); // same as int i=100;
int* ip = new int(47); // different than new int[47]!

// 2) Default constructor of an object automatically called:
Rect r; // no need to use ()

// 3) Primitive types do not have default constructors!
int i; // no initialization done here

// 4) Object initialization will call the copy constructor:
Rect a;    // will call default constructor
Rect b=a;  // will call copy constructor, same as Rect b(a)
Rect c(a); // will call copy constructor, same as Rect c=a
```

Order of Constructors

- The constructor of a base class is always called before the constructor of its derived class.
- The same rule applies to long chains of derivation:

```
class A
 { };

class B : public A
 { };

class C : public B
 { };

 ...
```

Upcasting

- Casting an object type to the type of its base class (as a pointer or reference)
- So that classes can work with objects of known behavior (methods), even if an object may actually be of a derived type

```
class A  {  };

class B : public A {  };

void main ()
 {
   B bobject;
   B* bpt = &bobject; // 1) get a pointer to bobject

   A* a = (A*) bpt;   // 2) upcast bpt to a pointer to A:
                         always ok since "B is also A"
 }
```

Redefining versus overriding methods

- Redefinition of Methods
  - Methods with same name in a base and derived classes are disambiguated by the type of the object
- Overriding Methods
  - The ***virtual*** keyword allows to call a descendant method even if the object being used is of the base class type
  - Makes sense only when upcasting is used
- Polymorphism
  - The use of virtual methods is a key concept behind polymorphism
  - To be covered when we get to Chapter 15

Redefining versus overriding methods

```
class Animal
 { public:
    void eat () { cout<<"I eat generic food\n"; }
    virtual void fur () { cout<<"I have fur\n"; }
 };

class Cat : public Animal
 { public:
    void eat () { cout<<"I eat cat food\n"; } // 1) method redefined
    void fur () { cout<<"I have fluffy cat fur\n"; } // 2) overrided!
  };

void main ()
 {
   Cat cat;
   cout<<cat.eat();
   Animal* animal = (Animal*) &cat; // 3) upcast cat to a pointer to Animal
   cout<< animal->eat();   // 4) will print: "I eat generic food"
   cout<< animal->fur();   // 5) will print: "I have fluffy cat fur"
 }
```

Wake up

Pointers to functions

- A function pointer is a variable that stores the address of a function
- It allows a function to change its behavior when it is called separately
  - The same sort of function can either sort in an ascending or descending way
  - A compare function can be passed as an argument
- It enables “callback functions” or “event listener”
  - Passing a function to another function as an argument
  - In a graphic user interface, a function is called when a mouse click takes place

Pointers to functions

```
int add(int a, int b){
  return a + b;
}
int subtract(int a, int b){
  return a - b;
}

int main ()
{
  int num1, num2;
  char addOrSubtract = 'a';

  int (*myMath)(int, int);
  if(addOrSubtract == 'a'){
    myMath = add;
  }
  else{
    myMath = subtract;
  }
  int answer = myMath(num1, num2);
  cout<<"Answer is: "<<answer<<endl;
}
```

References

- References are always tied to someone else's storage
  - When you change the value of a reference you are always changing the value of someone else's variable/object
- Similar to pointers, BUT:
  - References always manipulate someone else's storage
    - References cannot be null
  - References must be initialized
    - You cannot declare a reference without initialization
  - A reference cannot be changed to refer to something else
    - Assignment will assign contents, not make the reference to reference another object

References

```
//: C11:FreeStandingReferences.cpp
   #include <iostream>
   using namespace std;

   // Ordinary free-standing reference:
   int y;
   int& r = y;          // (1) When a reference is created, it must
                        // be initialized to an existing object.

   const int& q = 12;   // (2) This is valid (note the const)

   // References are always tied to someone else's storage:
   int x = 0;
   int& a = x;          // (3) a is a reference to x

   int main() {
     a++;               // (4) we are actually incrementing x here
   }
```

References in functions

- References are commonly used as function arguments and return values
  - Any modification to the reference inside the function will cause changes to the argument outside the function

```
//: C11:Reference.cpp - Simple C++ references


int* f(int* x) { // 1) pointer passed to a function
  (*x)++;
  return x;      // Safe, x is outside this scope
}


int& g(int& x) { // 2) reference passed to a function
  x++;           // Same effect as in f()
  return x;      // Safe, x is outside this scope
}
```

References in functions

```
//: C11:Reference.cpp - continue

int& h() {        // 3) function returning a reference
  int q;
  return q;       // 3.1) this would generate an error since q is local

  static int x;   // 3.2) static makes x become a global variable
  return x;       // Safe, x lives outside this scope (even if not visible)
}

int g(int& a){
}

int main() {
  int a = 0;
  g(&a);          // Sending a pointer to a to f: ugly (but explicit)
  g(a);           // Sending a reference to a to g: clean (but hidden)
}
```

Passing a pointer by reference

```
//: C11:ReferenceToPointer.cpp

#include <iostream>
using namespace std;

void increment(int*& i) { i++; }  //Passing the reference of a pointer

int main() {
  int* i = 0; //i is a pointer
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
} ///:~
```

Output:

i = 0
i = 0x4

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 10 – References & Copy-Constructor Overloading & Default Arguments

Passing objects as arguments

- Whenever possible, always pass an object to a function as a **const** reference
- If the object has to be modified, then pass a simple (non-const) reference
- Passing an object by value will include the overhead of constructor call and copy of contents
- Pointers are only helpful if you want the possibility of an optional object argument (since the pointer can be null)

```
void process_event ( const Event& e, Event* newevent=0 );
```

Don’t do this!

```
//: C11:PassingBigStructures.cpp
struct Big {
  char buf[1000];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) // Passing Big by value will copy 1000 chars to
{                 // the new local Big b object!! Use a reference!
  b.i = 100;      // Do something to the argument
  return b;
}

int main() {
  B2 = bigfun(B);
}
```

Copy Constructor

- When you pass an object by value it calls the copy constructor
- If you don't make a copy-constructor, the compiler will create one for you

```
class Point
{
  private:
    int x, y;
  public:
    Point(int x1, int y1) { x = x1; y = y1; }
    Point(const Point &p1) {x = p1.x; y = p1.y; } // Copy constructor
    int getX() {  return x; }
    int getY() {  return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here
    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
    return 0;
}
```

Preventing pass-by-value

- **“How do you know that an object will never be passed by value?”**
  - You can declare a private copy-constructor. You don’t even need to create a definition (unless one of your member functions or a friend function needs to perform a pass-by-value.)

Preventing pass-by-value

```
//: C11:NoCopyConstruction.cpp - Preventing copy-construction

class NoCC {
  int i;
  NoCC(const NoCC&); // No definition ok (just declaration)
public:
  NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
  NoCC n;
  f(n);          // 1) Error: copy-constructor called
  NoCC n2 = n;   // 2) Error: c-c called
  NoCC n3(n);    // 3) Error: c-c called
}
```

Overloading functions

- Overloading allows the definition of functions with the same name, but with different arguments.
  - Example:

```
void print(char);
void print(float);
```

  - OpenGL is a C interface; therefore, cannot use overloading.
    - API 1.1 of OpenGL:

```
glVertex2f ( float, float );
glVertex2d ( double, double );
glVertex3d ( double, double, double );
```

Overloading on return values

- Not possible!
  - C++ allows ignoring a return value, making it difficult to overload on a returned value.
- So, you cannot do this:

```
void f();
int f();    → will generate error (f already exists)
```

Function Overloading: Updating Stash

- Very useful in constructors!

```
//: C07:Stash3.h - Function overloading
class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage
spaces
  int next;      // Next empty space
  // Dynamically allocated array of
bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size); // Zero quantity
  Stash(int size, int initQuantity);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
```

Function Overloading: Updating Stash

```
//: C07:Stash3.h - Function overloading

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}
```

Wake-up!

- https://youtu.be/-hVCk4GSJOQ

Default Arguments

- Sometimes default arguments can be used to reduce overloading:

```
// The 2 constructors in the Stash class:

     Stash(int size); // Zero quantity is used here
     Stash(int size, int initQuantity);

// Can be reduced to one:

     Stash(int size, int initQuantity = 0);

// These definitions now use the same constructor:

     Stash A(100), B(100, 0);
```

Default Arguments

- Rules:
  - only the last arguments can have default values
  - when an argument has a default value, all the next ones will also need to have default values
  - default arguments only appear in the declaration (not in the definition/implementation of the method/function)

```
// f.h:
void f (int x, int y=0, float f=1.1);
//or even:
void f (int x, int = 0, float = 1.1);

// f.cpp (no default values):
void f (int x, int y, float f) {  ...  }
```

Overloading vs. default arguments

- It is a design choice
  - Efficiency considerations:
    - if you start using if statements to test contents of default values, it may be a better design to split into several overloaded functions
  - Code maintenance considerations:
    - avoid implementing the same initialization code twice

Overloading vs. default arguments

```
//: C07:Mem.h
typedef unsigned char byte;
class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz); // could become a default value in
               // the previous constructor
 ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
};
```

Example of techniques

```
class GsVec2
 { public :
    union { struct{float x, y;};
            float e[2];
          };
   public :
    static const GsVec2 null;     //!< (0,0) null vector
    static const GsVec2 one;      //!< (1,1) vector
    static const GsVec2 minusone; //!< (-1,-1) vector
    static const GsVec2 i;        //!< (1,0) vector
    static const GsVec2 j;        //!< (0,1) vector
   public :

    /*! Initializes GsVec2 as a null vector. Implemented inline. */
    GsVec2 () : x(0), y(0) {}

    /*! Copy constructor. Implemented inline. */
    GsVec2 ( const GsVec2& v ) : x(v.x), y(v.y) {}

    /*! Initializes with the two given float coordinates. Implemented inline. */
    GsVec2 ( float a, float b ) : x(a), y(b) {}

    /*! Initializes with the two given int coordinates converted to floats. Implemented inline. */
    GsVec2 ( int a, int b ) : x(float(a)), y(float(b)) {}

    /*! Initializes with one int and one float. Implemented inline. */
    GsVec2 ( int a, float b ) : x(float(a)), y(b) {}
```

allow access to coordinates
by member or by float array syntax

multiple overloaded constructors
to reduce type-casting
when instantiating GsVec2

Example of techniques

```
/*! Initializes with one float and one int. Implemented inline. */
GsVec2 ( float a, int b ) : x(a), y(float(b)) {}

/*! Initializes with the two given double coordinates converted to floats. Implemented inline. */
GsVec2 ( double a, double b ) : x(float(a)), y(float(b)) {}

/*! Initializes from a float pointer. Implemented inline. */
GsVec2 ( const float* p ) : x(p[0]), y(p[1]) {}

/*! Set coordinates from the given vector. Implemented inline. */
void set ( const GsVec2& v ) { x=v.x; y=v.y; }

/*! Set coordinates from the two given float values. Implemented inline. */
void set ( float a, float b ) { x=a; y=b; }

/*! Set coordinates from the two given int values. Implemented inline. */
void set ( int a, int b ) { x=float(a); y=float(b); }

/*! Set coordinates from the two given double values. Implemented inline. */
void set ( double a, double b ) { x=float(a); y=float(b); }

…etc
```

multiple overloaded set methods to reduce type-casting

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 12 – Polymorphism

Announcement

- Reading assignment
  - Ch. 15

Redefining versus overriding methods

- Redefinition of Methods
  - Methods with same name in a base and derived classes are disambiguated by the type of the object.
- Overriding Methods
  - The virtual keyword allows to call a descendant method even if the object being used is of the base class type.
  - Makes sense only when ***upcasting*** is used.
- Polymorphism
  - The use of virtual methods is the key concept behind polymorphism.

Redefining versus overriding methods

```
class Animal
 { public:
    void eat () { cout<<"I eat generic food\n"; }
    virtual void fur () { cout<<"I have fur\n"; }
 };

class Cat : public Animal
 { public:
    void eat () { cout<<"I eat cat food\n"; } // 1) method redefined
    void fur () { cout<<"I have fluffy cat fur\n"; } // 2) overrided!
  };

void main ()
 {
   Cat cat;
   cout<<cat.eat();
   Animal* animal = (Animal*) &cat; // 3) upcast cat to a pointer to Animal
   cout<< animal->eat();   // 4) will print: "I eat generic food"
   cout<< animal->fur();   // 5) will print: "I have fluffy cat fur"
 }
```

Polymorphism

- **Inheritance** lets us inherit attributes and methods from another class
- **Polymorphism** uses those methods to perform different tasks
- Polymorphism in C++ is achieved with **virtual** functions.
  - allows an object to have its behavior extended, without the need to know about derived types, or if it was derived or not.

- It can be seen as the third essential feature in objected oriented programming.
  - the other two are:
    - data abstraction and
    - inheritance

Example Without Virtual Methods

```
///: C15:Instrument2.cpp - Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const { cout << "Instrument::play" << endl; }
};

// Wind objects are Instruments because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const { cout << "Wind::play" << endl; }
};

void tune(Instrument& i) { // Takes in an Instrument type
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

method Instrument::play()
will be called here...

Example With Virtual Methods

```
///: C15:Instrument3.cpp - Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const { cout << "Instrument::play" << endl; }
};

// Wind objects are Instruments because they have the same interface:
class Wind : public Instrument {
public:
  // Override interface function:
  void play(note) const { cout << "Wind::play" << endl; }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

virtual methods will cause “late biding”

method **Wind::play()** will now be called!

Wake up!

- https://youtu.be/lSYEnvl3LeE

The Virtual Table

- How C++ knows which method to call?
  - for each class with a virtual method a hidden VTABLE is created.
  - each class with a virtual method will have a hidden pointer VPTR pointing to its VTABLE.

- The extra hidden code achieves the polymorphism
  - compilers may implement their virtual tables in different ways, there is no standard for how the “hidden code” has to be.

The Virtual Table – example 1

```
// C15:Sizes.cpp - Object sizes with/without virtual functions
class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "NoVirtual: " << sizeof(NoVirtual) << endl;
  cout << "OneVirtual: " << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: " << sizeof(TwoVirtuals) << endl;
}
```

Output:

int: 4
void* : 4
NoVirtual: 4
OneVirtual: 8
TwoVirtuals: 8

Only 1 VPTR is
added even when a
class has two virtual methods:

let's check what is printed.

The Virtual Table

```
///: C15:Instrument4.cpp
enum note { middleC, Csharp, Eflat }; // Etc.
class Instrument {
public:
  virtual void play(note) const { cout << "Instrument::play" << endl; }
  virtual char* what() const { return "Instrument"; }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const { cout << "Wind::play" << endl; }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const { cout << "Percussion::play" << endl; }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const { cout << "Stringed::play" << endl; }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};
```

The Virtual Table

```
///: C15:Instrument4.cpp (continue…)
class Brass : public Wind {
public:
  void play(note) const { cout << "Brass::play" << endl; }
  char* what() const { return "Brass"; }
};
void tune(Instrument& i) {i.play(middleC);}
// New function:
void f(Instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  f(flugelhorn);
} ///:~
```

Output:

Wind::play

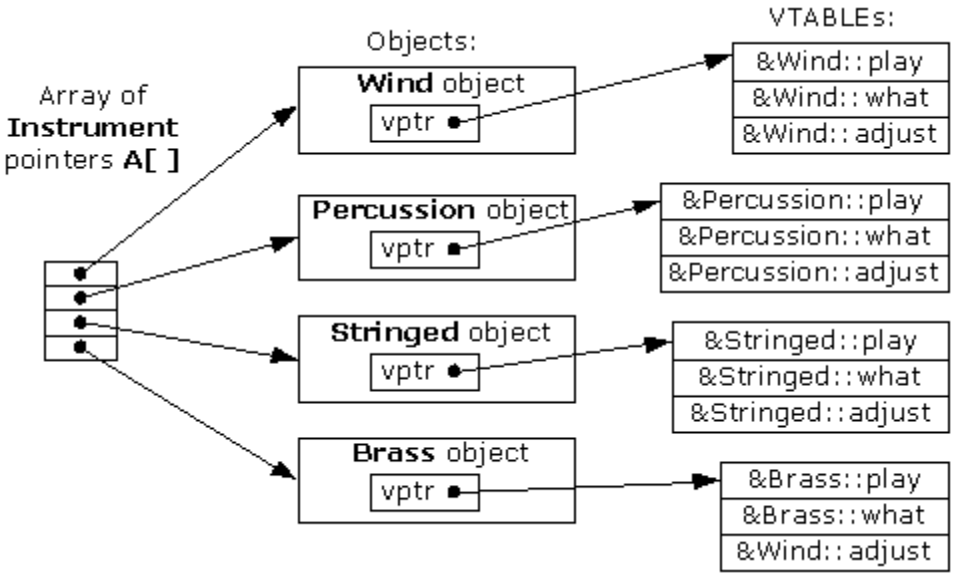
Percussion::play

Stringed::play

Brass::play

The Virtual Table

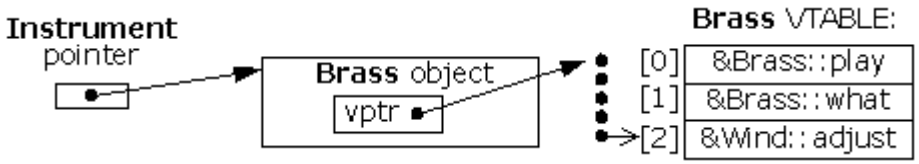
- Here are the vptrs and vtables created:



  - Each class has 1 vptr point to its vtable.
    - Objects of the same class can share vtables.
  - Each vtable keeps pointers to all virtual methods of an object.

The Virtual Table

- Example:
  - when a call to Brass::adjust is made, the compiler will say “call vptr+2” :



  - the correct pointers are stored at object creation
  - the correct methods to call can then be found at run-time even after upcasting (late binding).

The Virtual Table

- If methods are not declared virtual:
  - then they are simple methods, no vtable overhead is used
  - polymorphism is limited.

- Why virtual methods are not always employed in C++?
  - Idea is: “If you don't use it, you don't pay for it”

Abstract Classes

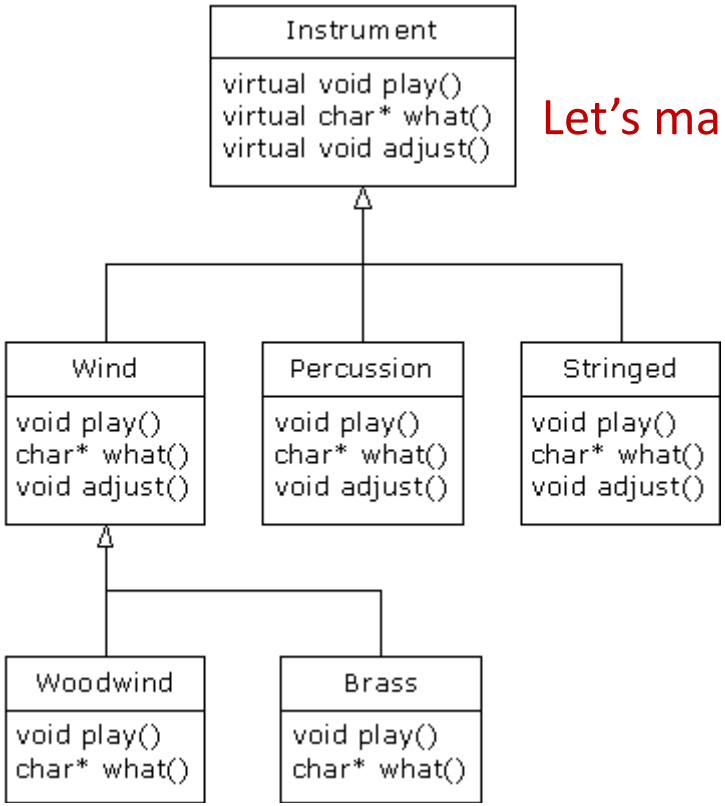
- When a class only presents an interface for derived classes
  - it cannot be instantiated
  - it sets a standard interface for extensions

- How to declare an abstract class:
  - just declare at least one "***pure virtual method***" with the "=0" syntax:

```
virtual void f()=0;
```

Abstract Classes

- Example:
  - Our “Instrument” class is a good candidate for becoming an abstract class.



Let’s make this interface abstract

Abstract Classes

```
//: C15:Instrument5.cpp - Pure abstract base classes
    class Instrument { public:
      // Pure virtual methods, all of them MUST be overridden by a derived class:
      virtual void play(note) const = 0;
      virtual char* what() const = 0;
      virtual void adjust(int) = 0;
    };

    class Wind : public Instrument { public:
      void play(note) const { cout << "Wind::play" << endl; }
      char* what() const { return "Wind"; }
      void adjust(int) {}
    };

    class Percussion : public Instrument { public:
      void play(note) const { cout << "Percussion::play" << endl; }
      char* what() const { return "Percussion"; }
      void adjust(int) {}
    };

    class Woodwind : public Wind { // Woodwind does not need to override all methods
     public:                       // since it inherits the non-abstract class Wind
      void play(note) const { cout << "Woodwind::play" << endl; }
      char* what() const { return "Woodwind"; }
    };
```

Abstract Classes

```
//: C15:Instrument5.cpp - Pure abstract base classes
(continue…)

    int main() {
      Instrument i; // not possible, will generate an error!
      Wind flute;
      Percussion drum;
      Woodwind recorder;
      ...
    }
```

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 13 – Polymorphism (2)

Announcement

- Reading assignment
  - Ch. 15

The Virtual Table

```
///: C15:Instrument4.cpp
enum note { middleC, Csharp, Eflat }; // Etc.
class Instrument {
public:
  virtual void play(note) const { cout << "Instrument::play" << endl; }
  virtual char* what() const { return "Instrument"; }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const { cout << "Wind::play" << endl; }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const { cout << "Percussion::play" << endl; }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const { cout << "Stringed::play" << endl; }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};
```

The Virtual Table

```
///: C15:Instrument4.cpp (continue…)
class Brass : public Wind {
public:
  void play(note) const { cout << "Brass::play" << endl; }
  char* what() const { return "Brass"; }
};
void tune(Instrument& i) {i.play(middleC);}
// New function:
void f(Instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  f(flugelhorn);
} ///:~
```

Output:

Wind::play

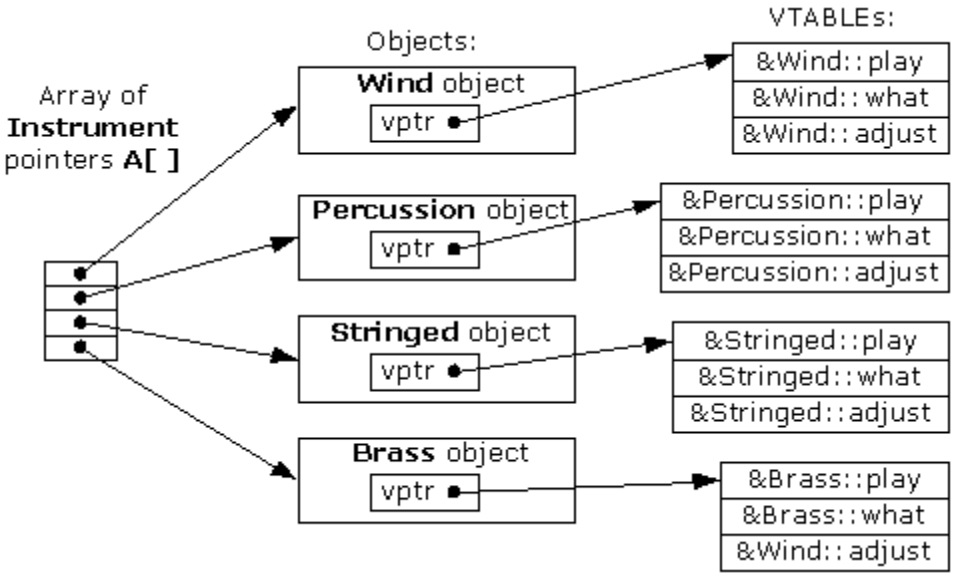
Percussion::play

Stringed::play

Brass::play

The Virtual Table

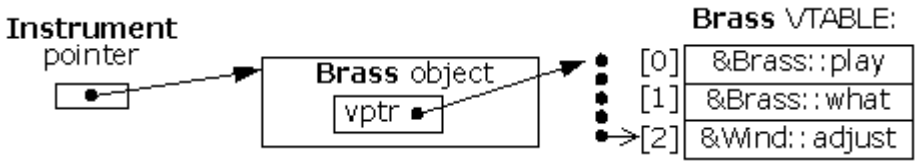
- Here are the vptrs and vtables created:



  - Each class has 1 vptr point to its vtable.
    - Objects of the same class can share vtables.
  - Each vtable keeps pointers to all virtual methods of an object.

The Virtual Table

- Example:
  - when a call to Brass::adjust is made, the compiler will say “call vptr+2” :



  - the correct pointers are stored at object creation
  - the correct methods to call can then be found at run-time even after upcasting (late binding).

Abstract Classes

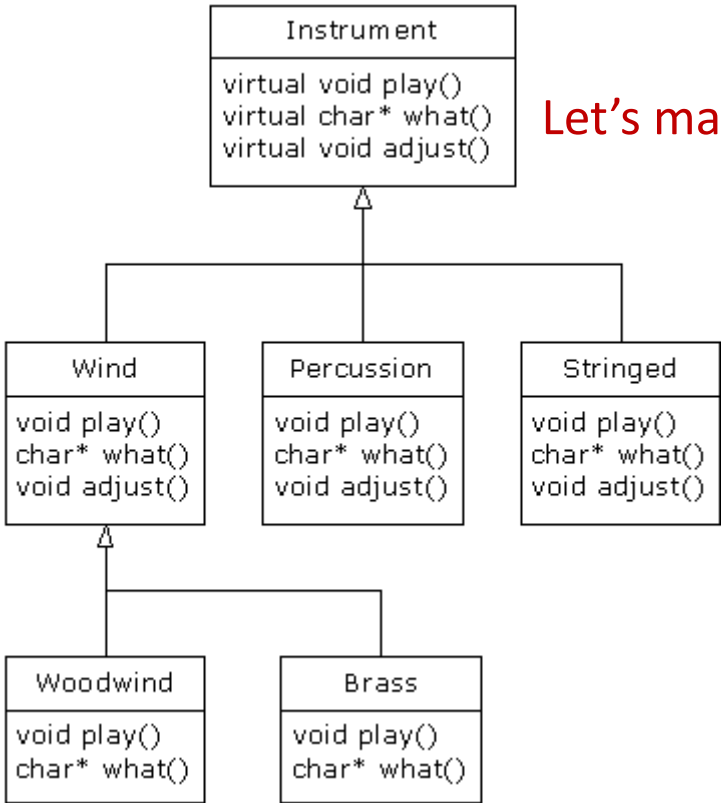
- When a class only presents an interface for derived classes
  - it cannot be instantiated
  - it sets a standard interface for extensions

- How to declare an abstract class:
  - just declare at least one "***pure virtual method***" with the "=0" syntax:

```
virtual void f()=0;
```

Abstract Classes

- Example:
  - Our “Instrument” class is a good candidate for becoming an abstract class.



Let’s make this interface abstract

Abstract Classes

```
//: C15:Instrument5.cpp - Pure abstract base classes
    class Instrument { public:
      // Pure virtual methods, all of them MUST be overridden by a derived class:
      virtual void play(note) const = 0;
      virtual char* what() const = 0;
      virtual void adjust(int) = 0;
    };

    class Wind : public Instrument { public:
      void play(note) const { cout << "Wind::play" << endl; }
      char* what() const { return "Wind"; }
      void adjust(int) {}
    };

    class Percussion : public Instrument { public:
      void play(note) const { cout << "Percussion::play" << endl; }
      char* what() const { return "Percussion"; }
      void adjust(int) {}
    };

    class Woodwind : public Wind { // Woodwind does not need to override all methods
     public:                       // since it inherits the non-abstract class Wind
      void play(note) const { cout << "Woodwind::play" << endl; }
      char* what() const { return "Woodwind"; }
    };
```

Abstract Classes

```
//: C15:Instrument5.cpp - Pure abstract base classes
(continue…)

    int main() {
      Instrument i; // not possible, will generate an error!
      Wind flute;
      Percussion drum;
      Woodwind recorder;
      ...
    }
```

Abstract Classes

- Extending Virtual Methods
  - Notice that the virtual method of the derived class (at the bottom of the hierarchy) will override the ones of the base classes
  - Sometimes we want to “add” and not really to “override”
    - For that, from your overriding method, you can always explicitly call the base class implementation using the scope operator

Abstract Classes

```
// SysWindow provides an abstract interface for windows to interact with the system
class SysWindow {
public:
  virtual void draw ()=0;     // Notice that virtual methods may have an implementation!
  virtual int handle ( const Event& e )=0;
};


// in SysWindow.cpp:
void SysWindow::draw ()
 {
   // make critical settings (but nothing to draw)
   glViewport ( ... );
   glEnable ( ... );
 }


int SysWindow::handle ( const Event& e )
 {
   // test if there is a UI attached (but SysWindow itself does not react to events)
   if ( user_interface_attached() ) return ui()->handle(e);
   return 0;
 }
```

Abstract Classes

```
class MyWindow : public SysWindow { // MyWindow implements my application
public:
  virtual void draw ();
  virtual int handle ( const Event& e );
};

// in MyWindow.cpp:
void MyWindow::draw ()
 { // first call the base class settings:
   SysWindow::draw();

   // now draw what you need to draw:
   drawWindowTitleBar();
   drawWindowDecoration();
   drawWindowContents();
 }

int MyWindow::handle ( const Event& e )
 { // first let the base class check for UI events:
   if ( SysWindow::handle ( e ) ) return 1;

   // now check events that are interesting for my window:
   if ( e.type == MouseClick )
    { moveToTop(); return 1; }
   else if ()
    { ... }

   // if event not useful:
   return 0;
 }
```

By calling the base class methods first we are able to ADD functionality to the base class implementation

Abstract Classes

```
class MyWindow : public SysWindow { // MyWindow implements my application
public:
  virtual void draw ();
  virtual int handle ( const Event& e );
};

// in MyWindow.cpp:
void MyWindow::draw ()
 { // first call the base class settings:
   SysWindow::draw();
   // now draw what you need to draw:
   ...
 }

int MyWindow::handle ( const Event& e )
 { // here we check a high-priority event that we do not
   // want the base class to handle:
   if ( e.type == UIClick ) { doSomethingElse(); return 1; }

   // ok now let the base class do its work:
   if ( notInFocus() )
    if ( SysWindow::handle ( e ) ) return 1;

   // finally check my events:
   if ( e.type == MouseClick )
    { moveToTop(); return 1; }
   ...
   // if event not useful:
   return 0;
 }
```

We can also selectively override some of the behavior of the base class in different ways

Wakeup

- https://youtu.be/6Z-y5-7Ywko

Abstract Classes

```
// Example with more complex derivation hierarchies:

class AppRect // Generic Graphical Object
public:
  virtual void draw ()=0;
  virtual int handle ( const Event& e )=0;
};

// Here is one specific graphical object:
class RectButton : public AppRect {
public:
  void draw (); // draw a button-like object
  int handle ( const Event& e ); // respond to mouse clicks
};
```

Abstract Classes

```
// Example with more complex derivation hierarchies: (continue…)

// Here is a new specialized abstract class:
class RectData : public AppRect {
public:
  bool load ( const char* file );
  bool save ( const char* file );
  Vec computeAverage ();
  ...
  virtual void draw ()=0; // the implementation here shows the data points
  virtual int handle ( const Event& e ); // interact (pan,zoom,etc) with the data
  virtual void swapmem ( void* pt, unsigned bytes )=0; // new pure method for
                                                       // custom memory management
};

class RectLineGraph : public RectData {
public:
  virtual void draw (); // draw a line graph from the data
  virtual void swapmem ( void* pt, unsigned bytes )=0; // may use disk swap
};

class RectBarGraph : public RectData {
public:
  ...
};
```

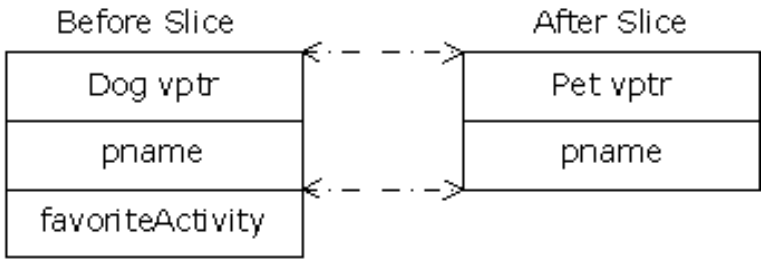
New Abstract Classes
may be created at any point

Object Slicing – passing by values

```
////: C15:ObjectSlicing.cpp

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}
```

Output:

This is Alfred
This is Fluffy




```
int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
}
```

Avoid Object Slicing With Pointers

```
void describe(Pet *p) {
  cout << p->description() << endl;
}


int main() {
  Pet* p = new Pet("Alfred");
  Dog* d = new Dog("Fluffy", "sleep");
  describe(p);
  describe(d);
}
```

Output:

This is Alfred
Fluffy likes to sleep

Overload vs Override

- Sound similar, but they are very different things
- What's the difference?

CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 14 – Polymorphism (3)

Announcement

- Reading assignment
  - Ch. 13

Object Slicing

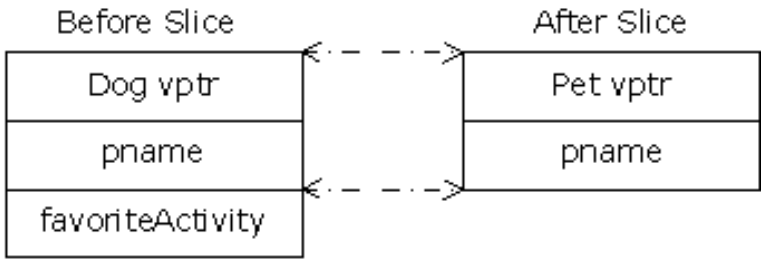
- Happens when a derived class object is assigned to a base class object
- We can avoid above unexpected behavior with the use of pointers or references

Object Slicing – passing by values

```
////: C15:ObjectSlicing.cpp

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}
```

Output:

This is Alfred
This is Fluffy




```
int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
}
```

Overriding virtual methods

```
class Base {
 public:
  virtual int f(){ cout << "Base::f()\n"; return 1; }
  virtual void f(string){}
  virtual void g(){}
};

class Derived1 : public Base {
 public:
  void g(){} // ok, only one match for overriding
};

class Derived2 : public Base {
 public:
  int f(){ cout << "Derived2::f()\n"; return 2; } // ok, overriding int Base::f()
};

class Derived3 : public Base {
 public:
  void f(){ cout << "Derived3::f()\n";} // ERROR: return type of Base::f() is different
};

class Derived4 : public Base {
 public:
  int f(int){ cout << "Derived4::f()\n";  return 4; } // Here we are NOT OVERRIDING !
};
```

Overriding virtual methods

- Case where overriding with different return type is ok:

```
class PetFood {
  public:
    virtual string foodType() const = 0;
};

class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
};

class Pet {
  public:
    virtual string type() const = 0;
    virtual PetFood* eats() = 0;
};

class Cat : public Pet {
  private:
    CatFood cf;
  public:
    string type() const { return "Cat"; }
    CatFood* eats() { return &cf; } // Ok to return CatFood, because it derives from PetFood
};
```

Constructors and destructors

- When an object containing virtual functions is created, its vptr must be initialized to point to the proper VTABLE.
  - This must be done before there's any possibility of calling a virtual function.
- Default Constructor
  - If you do not provide a default constructor for a class, the compiler will create one for you only for ensuring that the vptr of the object is correctly assigned.
    (no member initialization code is generated)

Constructors and destructors

- Order of Constructor Calls
  - The base class constructor is always called first, before the derived class constructor is called

- Calling virtual methods from a Constructor
  - What happens?
    - Only the local version of the method is called!!
    - the virtual mechanism doesn't work within the constructor.
    - even if it is virtual, the overridden version is not called. One reason is because the derived class is not yet initialized.

Constructors and destructors

- Virtual Destructors
  - must be used to ensure all classes in a derivation hierarchy are properly destroyed.
  - when delete is called for an object, all its virtual destructors are called starting from the derived class.
  - Forgetting to make a destructor **virtual** can introduce memory leak.

Constructors and destructors

```
// C15:VirtualDestructors.cpp - behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};
```

Output:

~Base1()
~Derived2()
~Base2()

```
int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
}
```

Constructors and destructors

- Calling virtual methods from a Destructor
  - What happens?
    - Only the local version of the method is called !!
    - even if it is virtual, the overridden version is not called. The reason is because, due to the order of destructor calls, the derived classes “are already destroyed”!

Constructors and destructors

- Good rule to follow
  - whenever an object is supposed to derive another object, make its destructor virtual
    - this will ensure correct destruction of all objects in a derivation hierarchy

Wake up!

- https://youtu.be/FM8sIVzvOlw

Operator overloading

- Operators can also be declared virtual and can be overloaded
  - Ex:

```
class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix*) = 0;
  virtual ~Math() {}
};
```

  - Overloaded virtual operators are not commonly used, avoid using them.
  - But simple overloading of operators is very useful, we will cover it later in detail (ch 12)

Downcasting

- It is possible, but it requires an explicit type cast, example:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){} };
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast ok
  // We know it is a cat, so we can just cast it to Cat*:
  Cat* d2 = (Cat*)(b);
}
```

C-like casts like this can be dangerous since there is no check if the cast is reasonable

Downcasting

- But recall the C++ casting keywords:
  - **static_cast**

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){} };
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast ok
  // We know it is a cat, so we can just cast it to Cat*:
  Cat* d2 = static_cast<Cat*>(b);
}
```

A static_cast will make the compiler test if the two types are on the same hierarchy (better but not really safe)

Downcasting

- **dynamic_cast** provides safer casts:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){} };
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  // Try to cast it to Dog*:
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Try to cast it to Cat*:
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
}
```

Output:

d1 = 0
d2 = 7409616

dynamic_cast:
you must be working with
a true polymorphic hierarchy
(one with virtual functions)

A dynamic_cast will return 0 if
the casting is not correct,
so you can check that to
guarantee a safe cast!

Downcasting

- Another way of using run-time type information (RTTI) is with **typeid**.
  - Need to include<typeinfo>

- Again, as with **dynamic_cast**, **typeid** requires a polymorphic object
  - otherwise the local(static) type is returned.

Downcasting

```
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
     Circle c;
     Shape* s = &c; // Upcast: normal and OK
     s = static_cast<Shape*>(&c); // More explicit but unnecessary

     Circle* cp = 0;
     Square* sp = 0;
     Shape* shapePnter = 0;

     // Example of using typeid():
     if (typeid(s) == typeid(cp)) // C++ RTTI
       cout << "It's a circle!" << endl;
     if (typeid(s) == typeid(sp))
       cout << "It's a square!" << endl;
     if (typeid(s) == typeid(shapePnter))
       cout << "It's a shape!" << endl;
}
```

Static navigation is ONLY an efficiency hack; dynamic_cast is always safer. However:

```
Other* op = static_cast<Other*>(s);
```

Conveniently gives an error message, while

```
Other* op2 = (Other*)s;
```

does not.

Downcasting

- Summary
  - C-like casts are fast but not safe
  - **static_cast** is a bit faster than **dynamic_cast** but will only prevent you from casting out of the hierarchy
  - **dynamic_casts** and **typeid** use RTTI to check for safe casts.
    - but recall they require polymorphic objects (with virtual functions) to be used