Lecture: Andrei Doncescu
FILS-IPB

C++/C

Outline

- Fundamentals of C++
- Class & inheritance
- Overloading & overriding
- Templates, Error handling,…

Reference C/C++

- The C Programming Language
  - Brian Kernighan & Denis Ritchie
- The C++ Programming Language
  - Bjarne Stroustrup

C/C++ Compilers

- Differences between the compilers Windows and Linux
  - Linux
    - The compiler C the most used is GCC
    - The equivalent C++ is G++
  - Windows
    - GCC/G++ exit with Cygwin and MinGW
    - Differents IDE exist and propose their own compiler
      - Microsoft Visual Studio with CL
      - Borland C++ Builder / Turbo C++ / Borland Developper Studio avec BCC32
      - Code Blocks / Dev-C++ with MinGW

- Differences between the compilers Windows and Linux

Equivalences Linux / Windows		
	Linux/GCC	Windows/Visual C++
Object Files	.o	.obj
Static Library	.a	.lib
Dynamiv Library	.so	.dll
Executed	-	.exe

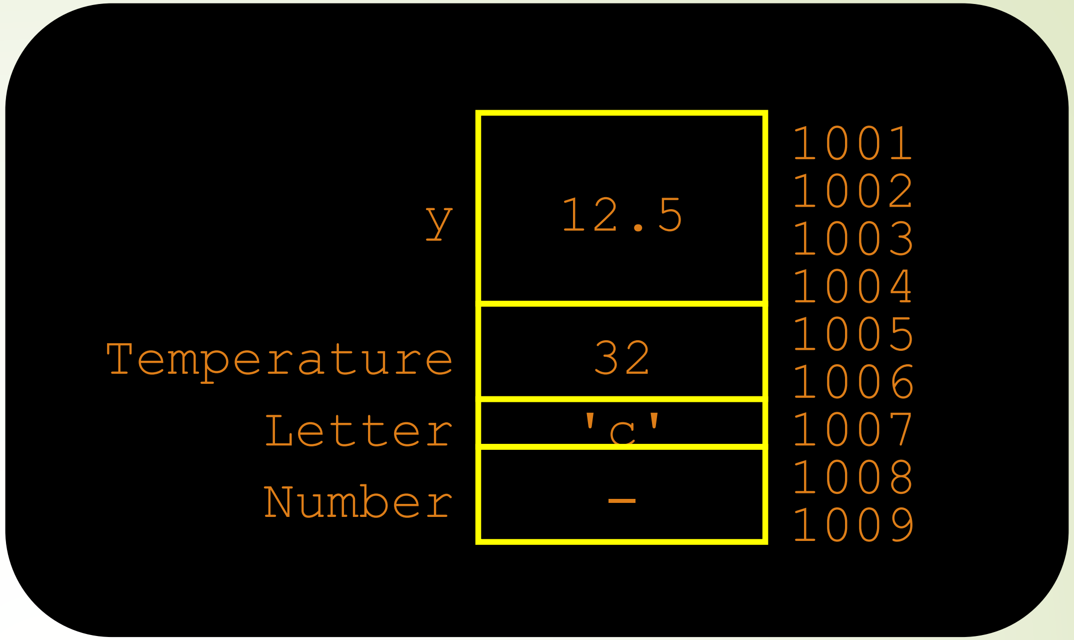
An example


```
#include<iostream.h>
int main()
{
      int i=0;
      double x=2.3;
      char s[]="Hello";

      cout<<i<<endl;
      cout<<x<<endl;
      cout<<s<<endl;

      return 0;
}
```

C Basics

Variables, Identifiers, Assignments, Input/Output

Variables



- variable can hold a number or a data of other types, it <u>always</u> holds something. A variable has a name

- the data held in variable is called *value*

- variables are implemented as memory locations and assigned certain memory address. The exact address depends on computer and compiler.

Identifiers

- *identifier* – name of a variable or any other named construct
- identifier must start with a letter or underscore symbol (_), the rest of the characters should be letters, digits or underscores
- the following are legal identifiers:

  `x  x1  x_1  _abc  sum  RateAveragE`

- the following are not legal identifiers. Why?

  `13  3X  %change  data-1  my.identifier  a(3)`

- C++ is *case sensitive:*

  `MyVar` and `myvar` are different identifiers

Identifier Style

- careful selection of identifiers makes your program clearer
- identifiers should be
  - short enough to be reasonable to type (single word is norm)
    - standard abbreviations are acceptable
  - long enough to be understandable
- two styles of identifiers
  - C-style - use abbreviations and underscores to separate the words, never use capital letters for variables
  - Camel Case - if multiple words: capitalize, do not use underscores
    - variant: first letter lowercased
- pick style and use consistently
- ex:

Camel Case 1	C-style	Camel Case 2
`Min`	`min`	`min`
`Temperature`	`temperature`	`temperature`
`CameraAngle`	`camera_angle`	`cameraAngle`
`CurrentNumberPoints`	`cur_point_nmbr`	`currentNumberPoints`

Keywords

keywords are identifiers reserved as part of the language

`int, return, float, double`

- cannot be used by the programmer to name things
- consist of lowercase letters only
- have special meaning to the compiler

Keywords

asm	do	if	return	typedef
auto	double	inline	short	typeid
bool	dynamic_cast	int	signed	typename
break	delete	long	sizeof	union
case	else	mutable	static	unsigned
catch	enum	namespace	static_cast	using
char	explicit	new	struct	virtual
class	extern	operator	switch	void
const	false	private	template	volatile
const_cast	float	protected	this	wchar_t
continue	for	public	throw	while
default	friend	register	true	union
delete	goto	reinterpret_cast	try	unsigned

Variable Declarations

- before use, every variable in C++ program needs to be *declared*
- *type* – the kind of data stored in a variable
- *declaration* – introduces a variable to a compiler
- a variable declaration specifies
  - type
  - name

`list of one or more identifiers`

- declaration syntax: `type id, id, ..., id;`

- two commonly used numeric types are:
  - `int` - whole positive or negative numbers: `1,2, -1,0,-288,` etc.
  - `double` - positive or negative numbers with fractional part: `1.75, -0.55`
- example declarations:

```
int numberOfBars;
double weight, totalWeight;
```

Declaration Location, Initial Value

- a variable should be declared as close to its use as possible
- variable contains a value after it is declared
  - until assigned, this value is arbitrary

Assignment

```
variable = value;
```

- *assignment statement* is an order to the computer to set the value of the variable on the left hand side of equal sign to what is written on the right hand side
- it looks like a math equation, but it is not
- example:

```
numberOfBars = 37;
totalWeight = oneWeight;
totalWeight = oneWeight * numberOfBars;
numberOfBars = numberOfBars + 3;
```

Output

- to do input/output, at the beginning of your program insert
  `#include <iostream>`
  `using std::cout; using std::endl;`
- C++ uses streams for input an output
- *stream* – a sequence of data to be read (*input stream*) or a sequence of data generated by the program to be output (*output stream*)
- variable values as well as strings of text can be output to the screen using `cout` (console output):

```
cout << numberOfBars;
cout << "candy bars";
cout << endl;
```

- `<<` is called *insertion operator,* it inserts data into the output stream, anything within double quotes will be output *literally* (without changes) - `"candy bars taste good"`
- note the space before letter " `c`" - the computer does not insert space on its own
- keyword `endl` tells the computer to start the output from the next line

More Output

- several insertion operators can be *stacked* together in a single statement:

```
cout << numberOfBars << "candy bars\n";
```

- symbol `\n` at the end of the string serves the same purpose as `endl`
- arithmetic expressions can be used with the output statement:

```
cout << "The total cost is $" << (price + tax);
```

Input

- `cin` - (stands for Console INput) – stream used to give variables user-input values

  need to add the following to the beginning of your program

```
using std::cin;
```

- when the program reaches the input statement it just pauses until the user types something and presses <Enter> key
- therefore, it is beneficial to precede the input statement with some explanatory output called *prompt*:

```
cout << ”Enter the number of candy bars”;
cout << ”and weight in ounces.\n”;
cout << ”then press return\n”;
cin >> numberOfBars >> oneWeight;
```

- >> is *extraction operator*
- *dialog* – collection of program prompts and user responses
- input operator (similar to output operator) can be stacked
- *input token* – sequence of characters separated by white space (spaces, tabs, newlines)
- the values typed are inserted into variables when <Enter> is pressed, if more values needed - program waits, if extra typed - they are used in next input statements if needed

Lexical elements

- Identifiers: case sensitive
  - nCount, strName, Strname
- Reservered words
  - if, else, while
- Operators
  - +, ==, &, &&, ‘? :’
- Preprocessor Directives
  - #include, #if,

Primitive Data Types

Name	Size (bytes)	Description	Range
char	1	character or eight bit integer	signed: -128..127 unsigned: 0..255
short	2	sixteen bit integer	signed: -32768..32767 unsigned: 0..65535
long	4	thirty-two bit integer	signed: -2^{31} .. $2^{31}-1$ unsigned: 0 .. 2^{32}
int	* (4)	system dependent, likely four bytes or thirty-two bits	signed: -32768..32767 unsigned: 0..65535
float	4	floating point number	3.4e +/- 38 (7 digits)
double	8	double precision floating point	1.7e +/- 308 (15 digits)
long double	10	long double precision floating point	1.2e +/- 4932 (19 digits)
bool	1	boolean value false → 0, true → 1	{0,1}

Hello World!

```
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello\n";

    return 0;
}
```

Notion of namespace

Operator

Namespace

Not necessary to use std::

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
    cout << "Hello\n";

    return 0;
}
```

Variables declaration & assignments


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

int main()
    {
    int i,j,k;
    int l;
    i=10;
    j=k=l=20;  //j=(k=(i=20))
    cout<<"i="<<i<<endl;
    cout<<"k="<<k<<endl;
    cout<<"l="<<l<<endl;
    i+=10;     //i = i + 10;
    i++;       //i = i + 1;
    cout << "i="<<i<<endl;
    }
```

Expressions

- Boolean expressions
  - == , != , >, >=, < , <=, …
  - && , | | …
- Arithmetic expression
  - + , - , *, / ,% …
  - &, | …
- Assignment
  - =
- ? :
- Expressions have values

Example of expressions

- 7 && 8
- 7 & 8
- 7 / 8
- 7 % 8
- 7 >> 1

(i > 127 ? true : false)

(i > 127 ? i-127 : i)

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
->	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
--	2	Post-/Predecrement	from right
-	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
-	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left

Statements

- Assignments
- Conditional
- Loop
- Goto,break,continue
- Compound statement

compute the largest power of 2 less than or equal to n	`int power = 1;`
	`while (power <= n/2)`
	`   power = 2*power;`
	`System.out.println(power);`

compute a finite sum $(1 + 2 + \ldots + n)$	`int sum = 0;`
	`for (int i = 1; i <= n; i++)`
	`   sum += i;`
	`System.out.println(sum);`

compute a finite product $(n! = 1 \times 2 \times \ldots \times n)$	`int product = 1;`
	`for (int i = 1; i <= n; i++)`
	`   product *= i;`
	`System.out.println(product);`

print a table of function values	`for (int i = 0; i <= n; i++)` `   System.out.println(i + " " + 2*Math.PI*i/n);`

compute the ruler function (see PROGRAM *1.2.1)*	`String ruler = "1";`
	`for (int i = 2; i <= n; i++)`
	`   ruler = ruler + " " + i + " " + ruler;`
	`System.out.println(ruler);`

Conditional

```
if A
    B ;
if A
    B
     else
    C

If ( I > 10) {cout<<" > 10";} else  {cout<<" < 10";}
```

Trick

- What is the difference?
  - If (i==1) ……
  - If (i=1)……

A better way to compare a variable with a constant

if (constant==variable)..
if (10 == i)…..

Trick

- If (i) ......
- If (1) ....

Loop, for

- for (A;B;C) D
  - 1 execute A
  - 2 execute B
  - 3 if the value of B is false(==0), exit to D
  - 4 execute C, goto 2
- for(i=0; i<n; i++){cout << A[i]<<endl;}
  - for(;;) {…}

Loop, while & do while

- while A B
  - While (i>10) { x-=4;i--;}
- do A while B
  - do {x -=4;i--} while (i>10);

Goto,break,continue

```
For (; ;){
…
If (a==b) break;
…
}
C
-------------------------------------------
-------------------------------------------

For (;;){
{B}
If (a==b) continue;
{A}
}
```

switch

- switch (grade){

```
case 'A':++nACount;break;
case 'B':++nBCount;break;
case 'C':++nCCount;break;
case 'D':++nDCount;break;
default: cout<<"Something wrong\n";break;
}
```

  - Try: write a program using the code segment. Then remove several of the 'break's and see the difference

functions

- Can not define a function within another function*
- Parameters passed by value or reference

Example

- #include<iostream>
- using namespace std;

- int square (int);
- int main ()
- {
- int z = 4;
- cout << square(z);
- }
- int square (int x)
- {   x = (x*x); return x; }   6.cpp

Pass by value

- void swap1(int x,int y)
- {
-   int temp=x;
-   x = y;
-   y=temp;
- }

Pass by reference

- void swap2(int& x,int& y)

```
{
    int temp=x;
    x = y;
    y=temp;
}
```

To pass a variable by reference, we simply declare the function parameters as references rather than as normal variables:

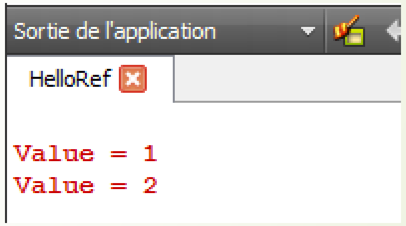
```
void addOne(int &y) // y is a reference variable
{
   y = y + 1;
}
```

```
/*      exchange values     */
#include    <stdio.h>
void swap(float *x, float *y);
main()
    {
    float   x, y;
    printf("Please input 1st value: ");
    scanf("%f", &x);
    printf("Please input 2nd value: ");
    scanf("%f", &y);
    printf("Values BEFORE 'swap' %f, %f\n", x, y);
    swap(&x, &y);       /*    address of x, y        */
    printf("Values AFTER  'swap' %f, %f\n", x, y);
    return 0;
  }
/*    exchange values within function   */
  void swap(float *x, float *y)
  {
    float   t;
    t = *x;  /*     *x is value pointed to by x   */
    *x = *y;
    *y = t;
    printf("Values WITHIN 'swap' %f, %f\n", *x, *y);     }
```

References

```cpp
#include <iostream>

void functionWithRef(int& value)
{
    value = 2;
}

int main(int argc, char* argv[])
{
    int a = 1;

    std::cout << "Value = " << a << "\n";

    functionWithRef(a);

    std::cout << "Value = " << a << "\n";

    return 0;
}
```

Pass by reference



Array in C/C++

- Definition
  - Int a[10];   //int[10] a;
  - Char b[12];
- No bounds checking
  - The cause of many problems in C/C++

Array

- Int x[7];

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]

- Int score[3][3]={{1,2,3},{2,3,4}{3,5,6}};

Array: confusing

- What is the result of the program.
- So, array is passed by reference?

Exo 8 :

Name of space


```
#include <iostream>


namespace balle

{

double centre_x=50, centre_y=0;
int rayon=5;

void deplacer (double x, double y)

{
        centre_x+=x;
        centre_y+=y;

}}


void test_bale()

{

        balle::deplacer(1,2,3);

}
```

Pointers

Why C/C++ is a good Programming Language ?

Pointers

A *pointer* is a reference to another variable (memory location) in a program

- Used to change variables inside a function (reference parameters)
- Used to remember a particular member of a group (such as an array)
- Used in dynamic (on-the-fly) memory allocation (especially of arrays)
- Used in building complex data structures (linked lists, stacks, queues, trees, etc.)

Outline

- Pointers
  - Basics
    - Variable declaration, initialization, NULL pointer
    - & (address) operator, * (indirection) operator
    - Pointer parameters, return values
    - Casting points, void *
  - Arrays and pointers
    - 1D array and simple pointer
    - Passing as parameter
  - Dynamic memory allocation
    - calloc, free, malloc, realloc
    - Dynamic 2D array allocation (and non-square arrays)

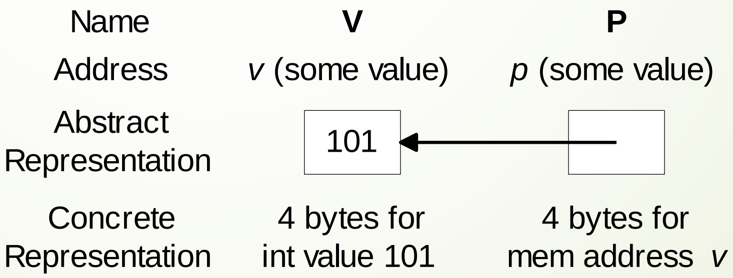
Pointer Basics

Variables are allocated at *addresses* in computer memory (address depends on computer/operating system)

Name of the variable is a reference to that memory address

A pointer variable contains a representation of an address of another variable (P is a pointer variable in the following):

```
int V = 101;

int *P = &V;
```



Pointer Variable Definition

Basic syntax: *Type *Name*

Examples:

```
int *P;   /* P is var that can point to an int var */
float *Q;   /* Q is a float pointer */
char *R;    /* R is a char pointer */
```

Complex example:

```
int *AP[5]; /* AP is an array of 5 pointers to ints */
```

- more on how to read complex declarations later

Address (&) Operator

The address (&) operator can be used in front of any variable object in C -- the result of the operation is the location in memory of the variable

Syntax: &*VariableReference*

Examples:

```
int V;
int *P;
int A[5];
&V - memory location of integer variable V
&(A[2]) - memory location of array element 2 in array A
&P - memory location of pointer variable P
```

Pointer Variable Initialization/Assignment

- NULL - pointer lit constant to non-existent address
  - used to indicate pointer points to nothing
- Can initialize/assign pointer vars to NULL or use the address (&) op to get address of a variable
  - variable in the address operator must be of the right type for the pointer (an integer pointer points only at integer variables)
- Examples:

```
int V;
int *P = &V;
int A[5];
P = &(A[2]);
```

Indirection (*) Operator

A pointer variable contains a memory address

To refer to the *contents* of the variable that the pointer points to, we use indirection operator

Syntax: **PointerVariable*

Example:

```
int V = 101;
int *P = &V;
/* Then *P would refer to the contents of the variable V (in this
   case, the integer 101) */
printf("%d",*P);  /* Prints 101 */
```

Pointer Sample

```
int A = 3;
int B;
int *P = &A;
int *Q = P;
int *R = &B;

printf("Enter value:");
scanf("%d",R);
printf("%d %d\n",A,B);
printf("%d %d %d\n",
  *P,*Q,*R);
```

```
Q = &B;
if (P == Q)
  printf("1\n");
if (Q == R)
  printf("2\n");
if (*P == *Q)
  printf("3\n");
if (*Q == *R)
  printf("4\n");
if (*P == *R)
  printf("5\n");
```

Reference Parameters

To make changes to a variable that exist after a function ends, we pass the address of (a pointer to) the variable to the function (a reference parameter)

Then we use indirection operator inside the function to change the value the parameter points to:

```
void changeVar(float *cvar) {
  *cvar = *cvar + 10.0;
}

float X = 5.0;

changeVar(&X);
printf("%.1f\n",X);
```

Pointer Return Values

A function can also return a pointer value:

```
float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);

  return theMax;
}

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;

  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("%.1f %.1f\n",*maxA,A[4]);
}
```

Pointers to Pointers

A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

Example:

```
int V = 101;
int *P = &V;    /* P points to int V */
int **Q = &P;  /* Q points to int pointer P */

printf(“%d %d %d\n”,V,*P,**Q); /* prints 101 3 times */
```

Pointer Types

Pointers are generally of the same size (enough bytes to represent all possible memory addresses), but it is inappropriate to assign an address of one type of variable to a different type of pointer

Example:

```
int V = 101;
float *P = &V; /* Generally results in a Warning */
```

Warning rather than error because C will allow you to do this (it is appropriate in certain situations)

Casting Pointers

When assigning a memory address of a variable of one type to a pointer that points to another type it is best to use the cast operator to indicate the cast is intentional (this will remove the warning)

Example:

```
int V = 101;
float *P = (float *) &V;          /* Casts int address to float * */
```

Removes warning, but is still a somewhat unsafe thing to do

The General (void) Pointer

- A void * is considered to be a general pointer
- No cast is needed to assign an address to a void * or from a void * to another pointer type
- Example:

```
int V = 101;
void *G = &V;    /* No warning */
float *P = G; /* No warning, still not safe */
```

- Certain library functions return void * results (more later)

1D Arrays and Pointers

int A[5] - A is the address where the array starts (first element), it is equivalent to &(A[0])

A is in some sense a pointer to an integer variable

To determine the address of A[x] use formula:

(address of A + x * bytes to represent int)

(address of array + element num * bytes for element size)

The + operator when applied to a pointer value uses the formula above:

A + x is equivalent to &(A[x])

*(A + x) is equivalent to A[x]

1D Array and Pointers Example

```
float A[6] = {1.0,2.0,1.0,0.5,3.0,2.0};
float *theMin = &(A[0]);
float *walker = &(A[1]);

while (walker < &(A[6]))
   {
  if (*walker < *theMin)
    theMin = walker;
  walker = walker + 1;
   }

printf("%.1f\n",*theMin);
```

1D Array as Parameter

When passing whole array as parameter use syntax *ParamName*[], but can also use **ParamName*

Still treat the parameter as representing array:

```
int totalArray(int *A, int N) {
  int total = 0;
  for (I = 0; I < N; I++)
    total += A[I];
  return total;
}
```

For multi-dimensional arrays we still have to use the *ArrayName*[][*Dim2*][*Dim3*]etc. form

Understanding Complex Declarations

Right-left rule: when examining a declaration, start at the identifier, then read the first object to right, first to left, second to right, second to left, etc.

objects:

- *Type*
- * - pointer to
- [*Dim*] - 1D array of size Dim
- [*Dim1*][*Dim2*] - 2D of size *Dim1,Dim2*
- (*Params*) - function

Can use parentheses to halt reading in one direction

Declarations Examples

int A — A is a int

float B [5] — B is a 1D array of size 5 of floats

int * C — C is a pointer to an int

char D [6][3] — D is a 2D array of size 6,3 of chars

int * E [5] — E is a 1D array of size 5 of pointers to ints

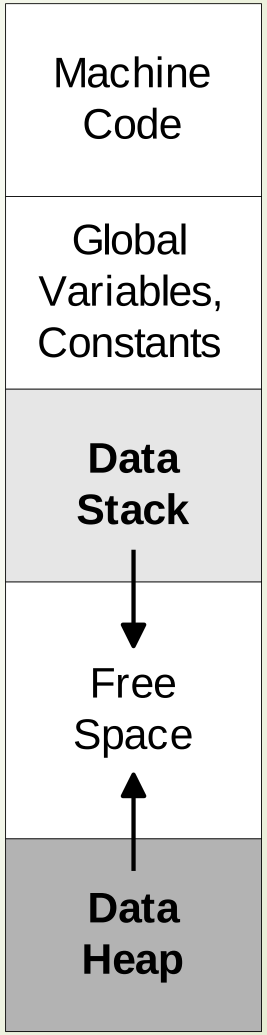
int (* F) [5] — F is a pointer to a 1D array of size 5 of ints

int G (…) — G is a function returning an int

char * H (…) — H is a function returning a pointer to a char

Program Parts

- Space for program code includes space for machine language code and data
- Data broken into:
  - space for global variables and constants
  - data stack - expands/shrinks while program runs
  - data heap - expands/shrinks while program runs
- Local variables in functions allocated when function starts:
  - space put aside on the data stack
  - when function ends, space is freed up
  - must know size of data item (int, array, etc.) when allocated (*static allocation*)



Limits of Static Allocation

What if we don't know how much space we will need ahead of time?

Example:

- ask user how many numbers to read in
- read set of numbers in to array (of appropriate size)
- calculate the average (look at all numbers)
- calculate the variance (based on the average)

Problem: how big do we make the array??

- using static allocation, have to make the array as big as the user might specify (might not be big enough)

Dynamic Memory Allocation

- Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically)
- Previous example: ask the user how many numbers to read, then allocate array of appropriate size
- Idea: user has routines to request some amount of memory, the user then uses this memory, and returns it when they are done
  - memory allocated in the *Data Heap*

Memory Management Functions

calloc - routine used to allocate arrays of memory

malloc - routine used to allocate a single block of memory

realloc - routine used to extend the amount of space allocated previously

free - routine used to tell program a piece of memory no longer needed

note: memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

Array Allocation with calloc

prototype: void * calloc(size_t num, size_t esize)

- size_t is a special type used to indicate sizes, generally an unsigned int
- num is the number of elements to be allocated in the array
- esize is the size of the elements to be allocated
  - generally use sizeof and type to get correct value
- an amount of memory of size num*esize allocated on heap
- calloc returns the address of the first byte of this memory
- generally we cast the result to the appropriate type
- if not enough memory is available, calloc returns NULL

calloc Example

```
float *nums;
int N;
int I;

printf("Read how many numbers:");
scanf("%d",&N);
nums = (float *) calloc(N, sizeof(float));
/* nums is now an array of floats of size N */
for (I = 0; I < N; I++) {
  printf("Please enter number %d: ",I+1);
  scanf("%f",&(nums[I]));
}
/* Calculate average, etc. */
```

Releasing Memory (free)

prototype: void free(void *ptr)

- memory at location pointed to by ptr is released (so we could use it again in the future)
- program keeps track of each piece of memory allocated by where that memory starts
- if we free a piece of memory allocated with calloc, the entire array is freed (released)
- results are problematic if we pass as address to free an address of something that was not allocated dynamically (or has already been freed)

The Importance of free

```
void problem() {
  float *nums;
  int N = 5;

  nums = (float *) calloc(N, sizeof(float));

  /* But no call to free with nums */
} /* problem ends */
```

When function problem called, space for array of size N allocated, when function ends, variable nums goes away, but the space nums points at (the array of size N) does not (allocated on the heap) - furthermore, we have no way to figure out where it is)

Problem called *memory leakage*

Array Allocation with malloc

- prototype: void * malloc(size_t esize)
  - similar to calloc, except we use it to allocate a single block of the given size esize
  - as with calloc, memory is allocated from heap
  - NULL returned if not enough memory available
  - memory must be released using free once the user is done
  - can perform the same function as calloc if we simply multiply the two arguments of calloc together
    - malloc(N * sizeof(float)) is equivalent to
    - calloc(N,sizeof(float))

Increasing Memory Size with realloc

prototype: void * realloc(void * ptr, size_t esize)

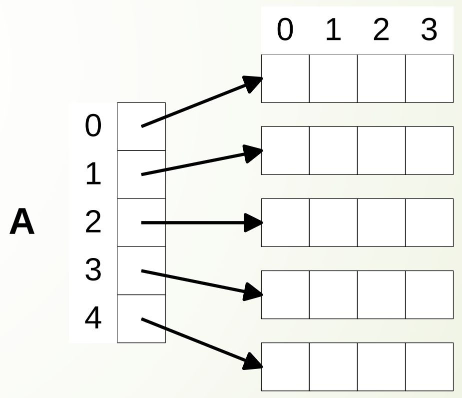
- ptr is a pointer to a piece of memory previously dynamically allocated
- esize is new size to allocate (no effect if esize is smaller than the size of the memory block ptr points to already)
- program allocates memory of size esize,
- then it copies the contents of the memory at ptr to the first part of the new piece of memory,
- finally, the old piece of memory is freed up

Realloc: Example

```
float *nums;
int I;

nums = (float *) calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */

for (I = 0; I < 5; I++)
  nums[I] = 2.0 * I;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */

nums = (float *) realloc(nums,10 * sizeof(float));
/* An array of 10 floating point values is allocated, the first
   5 floats from the old nums are copied as the first 5 floats
   of the new nums, then the old nums is released */
```

Dynamically Allocating 2D Arrays

- Can not simply dynamically allocate 2D (or higher) array
- Idea - allocate an array of pointers (first dimension), make each pointer point to a 1D array of the appropriate size
- Can treat result as 2D array



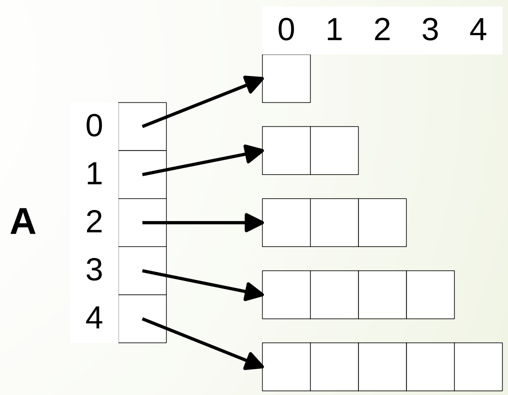
Dynamically Allocating 2D Array

```
float **A;  /* A is an array (pointer) of float
               pointers */
int I;

A = (float **) calloc(5,sizeof(float *));
/* A is a 1D array (size 5) of float pointers */

for (I = 0; I < 5; I++)
  A[I] = (float *) calloc(4,sizeof(float));
/* Each element of array points to an array of 4 float variables */

/* A[I][J] is the Jth entry in the array that the Ith member of A
   points to */
```

Non-Square 2D Arrays

No need to allocate square 2D arrays:

```
float **A;

int I;

A = (float **) calloc(5,
          sizeof(float *));

for (I = 0; I < 5; I++)
  A[I] = (float **)
           calloc(I+1,
            sizeof(float));
```



Pointer

- Example
  - int *p, char * s;
- The value of a pointer is just an address.
- Why pointers?
- Dereferencing (*)
  - Get the content
- Referencing (&)
  - Get the address of

Examples of pointer

- int *p;
- int a;
- a=10;
- p=&a;
- *p=7;
- int b=*p;
- You must initialize a pointer before you use it

arithmetic of pointer

- Suppose n is an integer and p1 and p2 are pointers
- p1+n
- p1-n
- p1-p2

Strings

- C
  - A string is an array of chars end with '\0'
  - char name[]="ABC";
  - char school_name[]={'N','Y','U};
- C++ library: string class

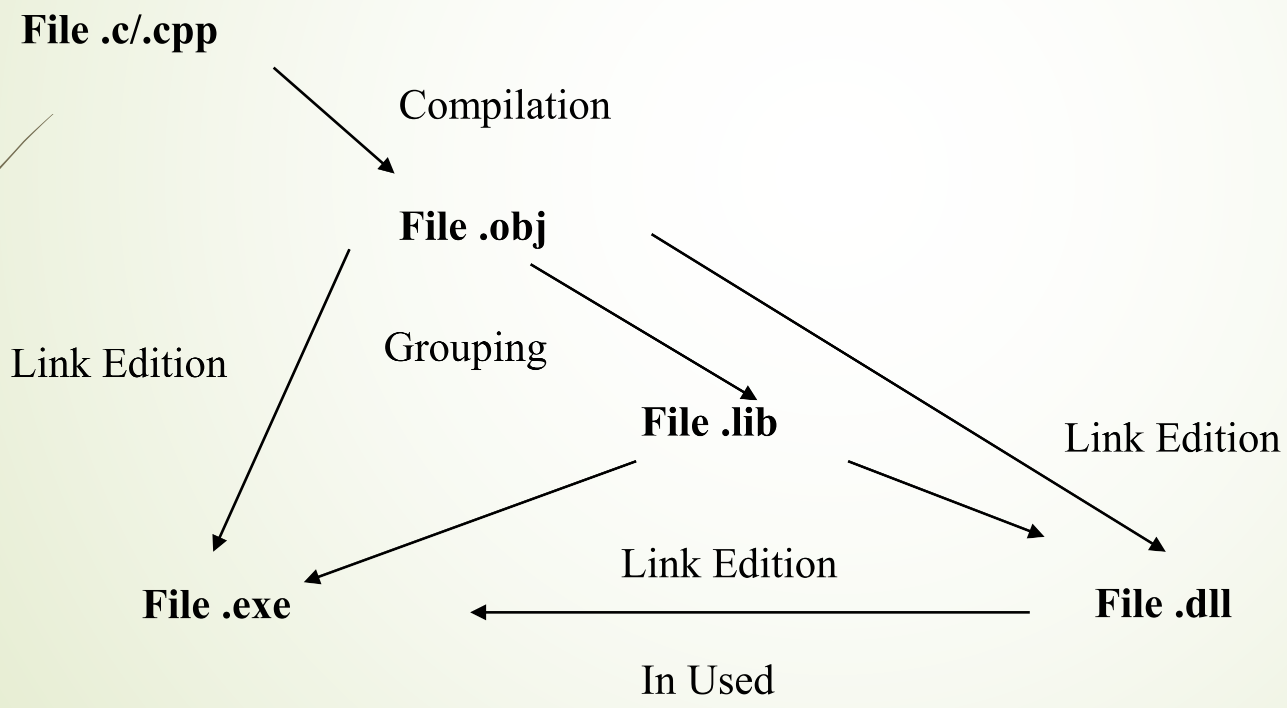
Dynamic allocating memory

- new , delete
- int *p=new int;
- int *p=new int [12];
- delete p;
- delete []p;
- malloc,…

malloc.c

```
#include <stdlib.h> /* using ANSI C standard libraries */
#include <malloc.h>


 main()
 { char *string_ptr; string_ptr = malloc(80); }
```

- Using External Libraries



From C to Assembler

- $ gcc -v
- $ hexdump –C a.out
- $ gcc –S prog1.c
- $ ldd a.out //

cdir.c

```
/* cdir.c program to emulate unix cd command */
/* cc -o cdir cdir.c */
#include<stdio.h>
/* #include<sys/dir.h> */

main(int argc,char **argv)
{
        if (argc < 2)
         { printf("Usage: %s <pathname>\n",argv[0]);
           exit(1);
         }

        if (chdir(argv[1]) != 0)
         { printf("Error in \"chdir\"\n");
       exit(1);
      }
}
```

Object-Oriented programming in C++

- Classes as units of encapsulation
- Information Hiding
- Inheritance
- polymorphism and dynamic dispatching
- Storage management
- multiple inheritance

Presentation

- C is included (99%) in C++
- The C++ is an oriented programming language (classes, inheritance, polymorphisme… like Java).

OBJECT ORIENTED PROGRAMMING (OOP)

- Object-oriented programming is a programming paradigm that uses abstraction (in the form of classes and objects) to create models based on the real world environment.
- An object-oriented application uses a collection of objects, which communicate by passing messages to request services.
- Objects are capable of passing messages, receiving messages, and processing data.
- The aim of object-oriented programming is to try to increase the flexibility and maintainability of programs. Because programs created using an OO language are modular, they can be easier to develop, and simpler to understand after development.

PROCEDURAL PROGRAMMING

- Procedural programming is a classic programming where the program language is used to tell the computer EXACTLY what to do - step by step.

- A program in a procedural language is a list of instruction. That is, each statement in the language tell the computer to do something.

- The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines.

PROBLEMS WITH PROCEDURAL PROGRAMMING

1. In Procedural language the programmers concentration is on problem and its solution so we need another approach where the developers study the entire system and developed.

2. After writing the programmers 10,000 to 1,00,000 lines of code in procedural language they are loosing their control on the code. they cant identify the errors easily. later if any modifications are there in the code it is difficult to modify.

   behind these above problems there are further two more problems with procedural programming i.e.

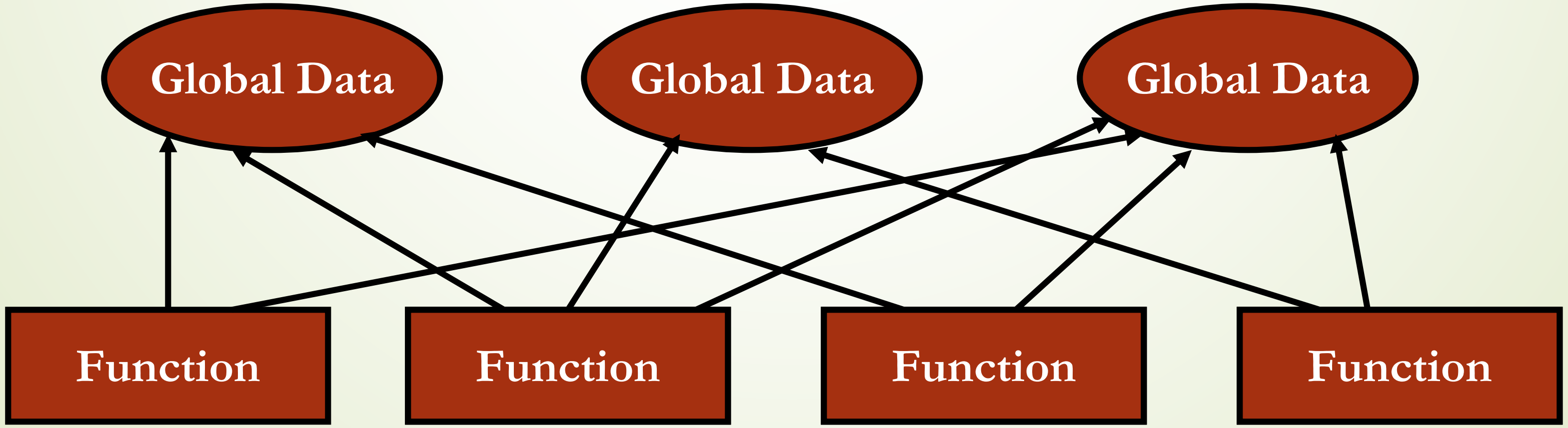
   first, in PPL functions are unrestricted access to global data.

   second, unrelated functions and data.

Lets examines these two problems…

1. In PPL, functions are unrestricted access to global data.

- In procedural program, there are two kinds of data. Local data (that is hidden inside a function and used exclusively by the function), and **Global data** (that can be accessed by any function in the program).
- In a large program, there are many functions and global data items. The problem with the procedural paradigm is that this leads to even larger number of potential connections b/w functions and data as shown in following Fig.



- This large number of connections causes problems in several ways.
  - First, it makes a program structure difficult to conceptualize.
  - Second, it makes the program difficult to modify. (because a change made in global data item may result in rewriting all the functions that access that item.)

When data items are modified in a large program it may not be easy to tell which functions access the data, and even when we figure this out, modification to the functions may cause them to work incorrectly with other global data items.

Everything is related to every thing else, so modification anywhere has far-reaching and often unintended , consequences.

The second and most important problem with the procedural paradigm is that its arrangement of separate data and functions does a poor job of modeling things in the real world.

Object-Oriented Programming vs. Procedural Programming

- Programs are made up of modules, which are parts of a program that can be coded and tested separately, and then assembled to form a complete program.
- In procedural languages (i.e. C) these modules are procedures, where a procedure is a sequence of statements, such as assignments, tests, loops and invocations of sub procedures.
- The design method used in procedural programming is called Top Down Design. This is where you start with a problem (procedure) and then systematically break the problem down into sub problems (sub procedures). This is called functional decomposition, which continues until a sub problem is straightforward enough to be solved by the corresponding sub procedure.
- The difficulties with this type of programming, is that software maintenance can be difficult and time consuming. When changes are made to the main procedure (top), those changes can cascade to the sub procedures of main, and the sub-sub procedures and so on, where the change may impact all procedures in the pyramid.

Object-Oriented Programming vs. Procedural Programming cont…

- One alternative to procedural programming is object oriented programming.
- Object oriented programming is meant to address the difficulties with procedural programming.
- In object oriented programming, the main modules in a program are classes, rather than procedures. The object-oriented approach lets you create classes and objects that model real world objects.

CONCEPTS OF OOP

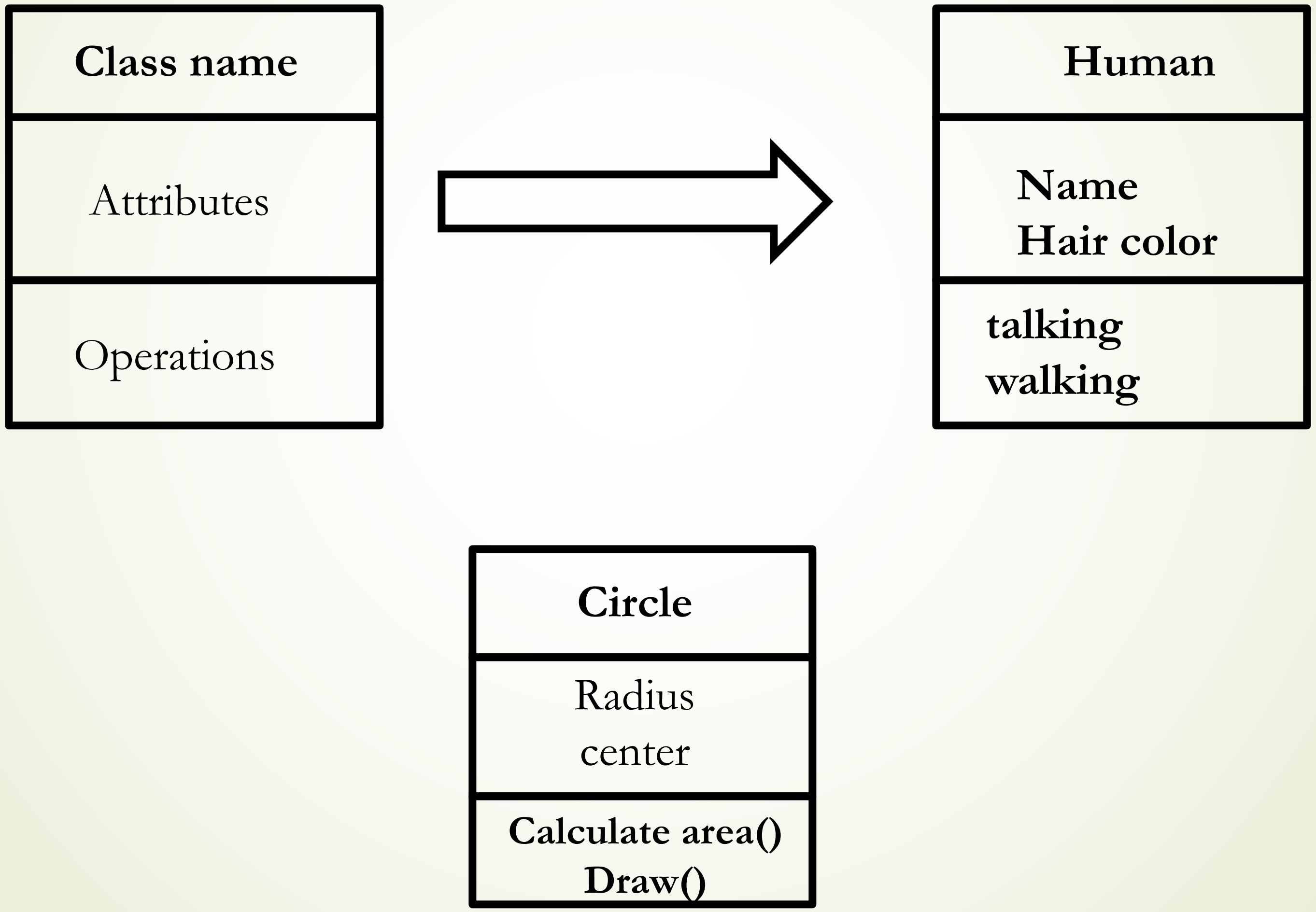
- Let’s briefly examine a few of the most elements of object-oriented programming.

- **CLASSES:** A ***class*** is an object-oriented concept which is used to describe properties and behavior of a real world entity.
  - A class is a combination of state (data) and behavior (methods).
  - In object-oriented languages, a class is a data type, and objects are instances of that data type.
  - In other words, classes are prototypes from which objects are created.

For example, we may design a class Human, which is a collection of all humans in the world. Humans have state, such as height, weight, and hair color. They also have behaviors, such as walking, talking, eating.

All of the state and behaviors of a human is encapsulated (contained) within the class human.

Representation of class



What is an Object?

Real-world objects have ***attributes*** and ***behaviors.***

Examples:

- Dog
  - Attributes: breed, color, hungry, tired, etc.
  - Behaviors: eating, sleeping, etc.
- Bank Account
  - Attributes: account number, owner, balance
  - Behaviors: withdraw, deposit

CONCEPTS OF OOP

- **OBJECTS:** An ***object*** is an instance of a class. An object can communicate with other objects using messages. An object passes a message to another object, which results in the invocation of a method. Objects then perform the actions that are required to get a response from the system.

- **Data Members:** A class contains data members. Data members are used to store characteristics information of a real world entity in a class.
- For example, name is the data member of the Human class.
  - Fields/attributes and methods/operation are referred to as class **members**.
- **Fields and Methods:** Objects in a class have a number of shared properties/features/attributes.
  - Fields are the variables contained in a class.
  - Methods are functions that represent the operations associated with a particular class.

Classes

The definitions of the attributes and methods of an object are organized into a ***class***. Thus, a class is the generic definition for a set of similar objects (i.e. *Person* as a generic definition for *Jane*, *Mitch* and *Sue*)

- A class can be thought of as a template used to create a set of objects.
- A class is a static definition; a piece of code written in a programming language.
- One or more objects described by the class are ***instantiated*** at runtime.
- The objects are called ***instances*** of the class.

CONCEPTS OF OOP

- **INHERITANCE:** One of the powerful features of c++ is inheritance.
- In object-oriented programming, inheritance is a way to form new classes using classes that have already been defined.
- Inheritance lets you increase the functionality of previously defined classes without having to rewrite the original class declaration. This can be a greater time saver in writing applications.
- **POLYMORPHISM:** In c++ you can declare two functions with the same name provided they can be distinguished by the number or type of arguments they take. This is called function overloading. Function overloading is an example of polymorphism, which means one thing serving several purposes.

- **REUSEBILITY**: Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called reusability.
- It is similar to the way a library functions in procedural language can be incorporated in different programs.

Struct:

```
struct person
{   long nId;
    char strName[30];
    int nAge;
    float fSalary;
    char strAddress[100];
    char strPhone[20];  };

struct person a ,  b, c;
struct person *p;
```

union

```
union num
{
  int x;
  float y;
}
```

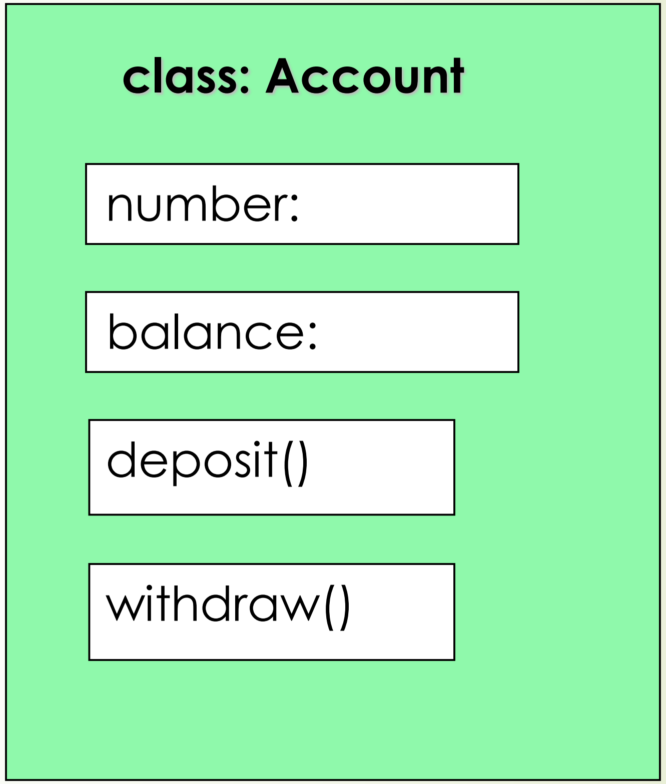
```
typedef struct foo {
int a;
char b;
} foo;
foo x;
typedef union bar {
int a;
char b;
} bar;
bar y;
x.a = 10; //OK
x.b = 'a'; //OK
y.a = 10; //OK
y.b = 'a'; //NOT OK
```

You are supposed to use only one of the elements in union because they are all stored in the same memory location. This is useful when you want to store something that could be one of several datatypes.
On the other hand, a struct has separate memory locations for each of its elements and all of these elements can be used at once.
Take the following case for example:

```
bar y;
y.a = 10;
y.b = 'a';
printf("%d %d", y.a, y.b);
```

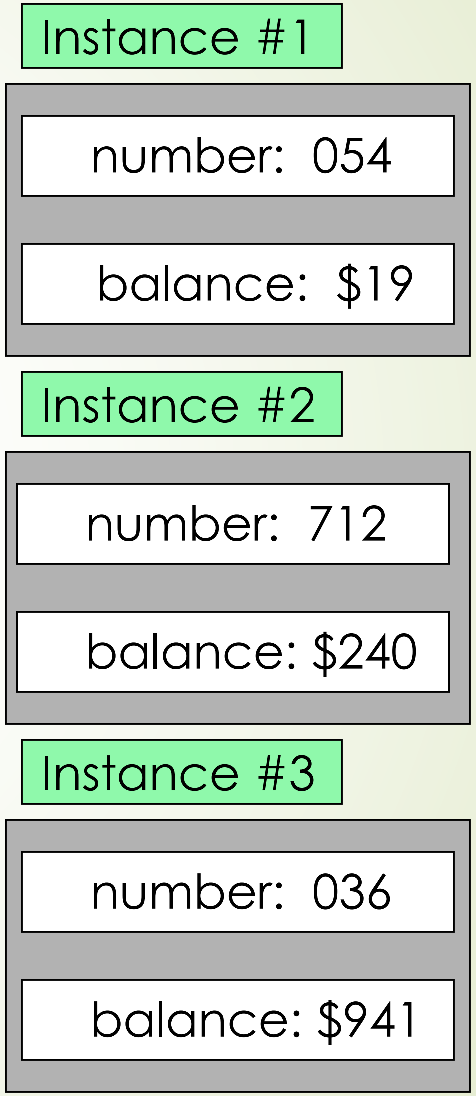
Bank Example

- The "account" class describes the attributes and behaviors of bank accounts.
- The “account” class defines two state variables (account number and balance) and two methods (deposit and withdraw).



Bank Example - Cont’d

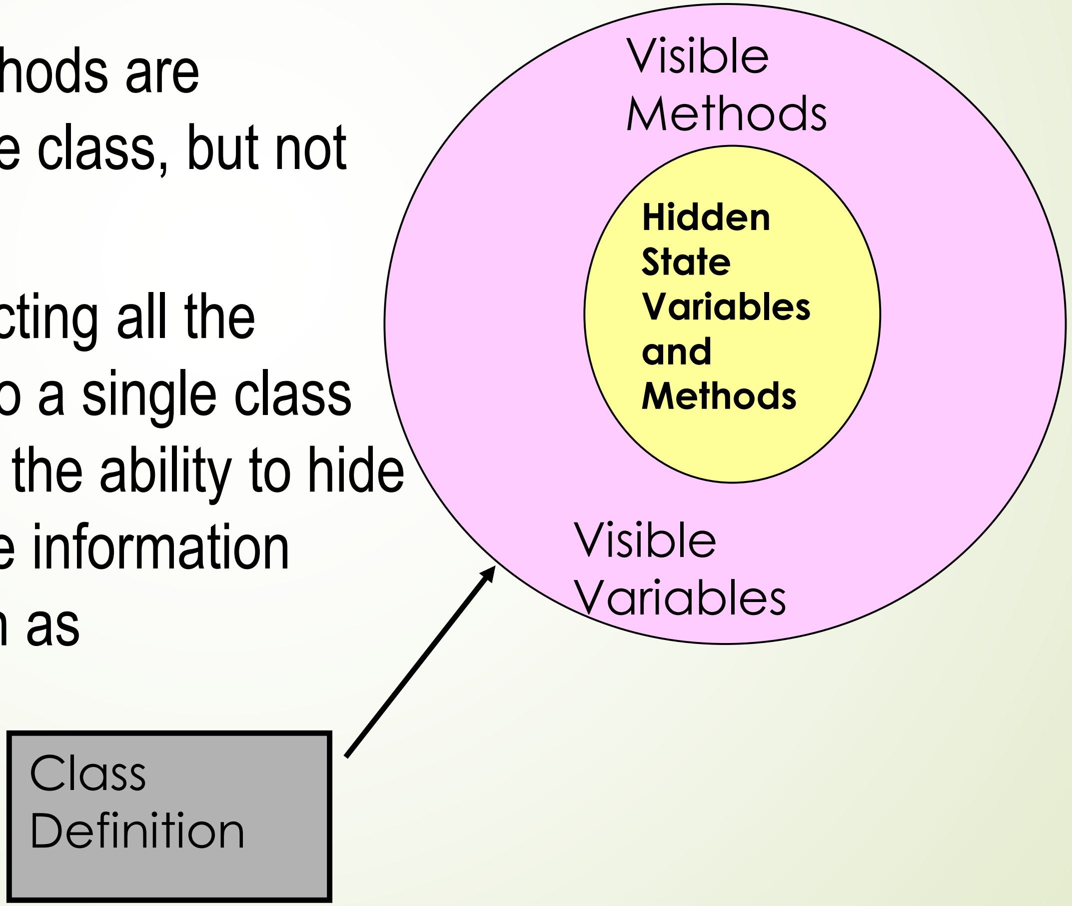
- When the program runs there will be many instances of the account class.
- Each instance will have its own account number and balance (*object state*)
- Methods can only be invoked .



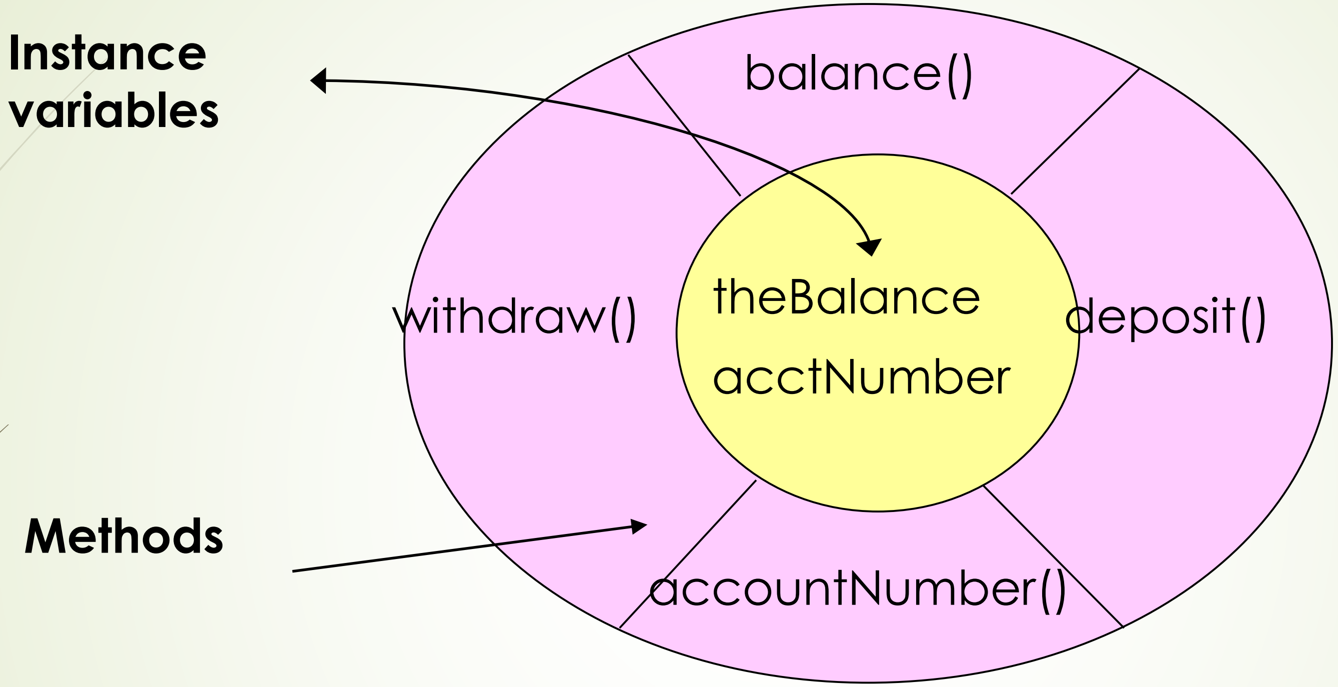
Encapsulation

When classes are defined, programmers can specify that certain methods or state variables remain hidden inside the class.

- These variables and methods are accessible from within the class, but not accessible outside it.
- The combination of collecting all the attributes of an object into a single class definition, combined with the ability to hide some definitions and type information within the class, is known as encapsulation.



Graphical Model of an Object



State variables make up the nucleus of the object. Methods surround and hide (encapsulate) the state variables from the rest of the program.

Structure of Object Oriented Program in c++

```
#include<iostream.h>
Class class-name
{
    access-specifiers…
     int i,j;
     float f;
     char ch;
     double b;
    access-specifiers…
     void function-name()
     {
          statement 1;
          statement 2;
     }
}
main()
{
    class-name.obj-name;
}
```

Header File

class

Variables or fields

Function or Method

Class members

Object of class

More in a stucture: operations

```
struct box
{

    double dLength,dWidth,dHeight;
    double dVolume;

    double get_vol()
      {
        return dLength * dWidth * dHeight;
      }
}
```

Operator overloading

- Define new operations for operators (enable them to work with class objects).
- + - * / = < > += -= *= /= << >> <<= >>= == != <= >= ++ -- % & ^ ! | ~ &= ^= |= && || %= [] () new delete
- Class date x ,y
  - X+y x-y x>y, x&y

Classes

- Encapsulation of type and related operations

```
class point {
    double x,y;                       // private data members
  public:
    point (int x0, int y0);           //  public methods
    point () { x = 0; y = 0;};        //  a constructor
    void move (int dx, int dy);
    void rotate (double alpha);
    int distance (point p);
}
```

A class is a type : objects are instances

point p1 (10, 20);   // call constructor with given arguments

point p2;   // call default constructor

Methods are functions with an implicit argument

p1.move (1, -1);   // special syntax to indicate object

// in other languages might write *move (p1, 1, -1)*

// special syntax inspired by message-passing metaphor:

// objects are autonomous entities that exchange messages.

Implementing methods

No equivalent of a body: each method can be defined separately

```
void point::rotate (double alpha) {
    x = x * cos (alpha) -  y * sin (alpha);
    y = y * cos (alpha) + x * cos (alpha);
};
//  x and y are the data members of the object on which the
// method is being called.
// if method is defined in class declaration, it is inlined.
```

Class

```
class box
{
   double dLength,dWidth,dHeight;
   double dVolume;
public:
   double vol(){return dLength * dWidth * dHeight;}

}
```

Constructors

- One of the best innovations of C++
- special method (s) invoked automatically when an object of the class is declared

```
point (int x1, int x2);
 point ();
 point (double alpha; double r);
point p1 (10,10), p2; p3 (pi / 4, 2.5);
```

- Name of method is name of class
- Declaration has no return type.

The target of an operation

- The implicit parameter in a method call can be retrieved through **this**:

```
class Collection {
   Collection& insert (thing x) {      // return reference
      …  modify data structure
      return *this;                    // to modified object
   };
};
my_collection.insert (x1).insert (x2);
```

class


```
class box
{
   double dLength,dWidth,dHeight;
   double dVolume;
public:
   double vol() ;
}
double box::vol()
{
return dLength * dWidth * dHeight;}
}
```

Vector.hpp

```
namespace cassebrique
{
	classe Vector {
	public:
	/*constructor*/
	Vector(double x, double y);

	/*method*/
	void newCoordonates(double x, double y);
	void getCoordonates(double &x,double &y) const;

	private:
		double m_x;
		double m_y;

	};
};
```

Constructors

- A special member function with the same name of the class
- No return type (not void)
- Executed when an instance of the class is the created

Deconstructors

- A special member function with no parameters
- Executed when the class is destroyed

Static members

- Need to have computable attributes for class itself, independent of specific object; e.g. number of objects created.
- Static qualifier indicates that entity is unique for the class

```
static int num_objects = 0;
point () { num_objects++;};  // ditto for other constructors
```

- Can access static data using class name or object name:

```
if (point.num_objects != p1.num_objects) error ();
```

Classes


```cpp
#ifndef TESTCLASS_H
#define TESTCLASS_H

class TestClass
{
private:
    double value;
public:
    // Member functions (EX : getters and setters)
    double getValue() const;
    void setValue(double);
};

#endif // TESTCLASS_H
```

TestClass.h

```cpp
#include "TestClass.h"

// Member functions (EX : getters and setters)
double TestClass::getValue() const
{
    return value;
}

void TestClass::setValue(double value)
{
    this->value = value;
}
```

TestClass.cpp

```cpp
#include <iostream>
#include "TestClass.h"

using namespace std;

int main(int argc, char* argv[])
{
    TestClass tst;

    tst.setValue(2.1);

    cout << tst.getValue() << endl;

    return 0;
}
```

Main.cpp

Public vs. private

- Public functions and variables are accessible from anywhere the object is visible
- Private functions and variable are only accessible from the members of the same class and “friend”
- Protected

Inheritance

	base	derived
Public inheritance	public	public
	protected	protected
	private	N/A
Private inheritance	public	private
	protected	private
	private	N/A
Protected inheritance	public	protected
	protected	protected
	private	N/A

Constructor/Destructor and Surcharged


```cpp
#ifndef TESTCLASS_H
#define TESTCLASS_H

#include <iostream>

class TestClass
{
private:
    double value;
public:
    // Constructors
    TestClass();
    TestClass(const TestClass&);
    TestClass(const double&);

    // Destructor
    ~TestClass();
};

#endif // TESTCLASS_H
```

TestClass.h

```cpp
#include "TestClass.h"

using namespace std;

TestClass::TestClass()
{
    value = 0;
}

TestClass::TestClass(const TestClass& a)
{
    value = a.value;
}

TestClass::TestClass(const double& value)
{
    this->value = value;
}

// Destructor
TestClass::~TestClass()
{
    value = 0;
}
```

TestClass.cpp

Initialisation

```
TestClass::TestClass(const TestClass& a) :
        value(a.value),
        tmpValue(1)
        // Initialization list
{

}
```

Member Initialisation
can be done in this way

Operators

```cpp
#ifndef TESTCLASS_H
#define TESTCLASS_H

class TestClass
{
private:
    double value;
public:
    // Member functions (EX : getters and setters)
    double getValue() const;
    void setValue(double);

    // Member operators (EX : unary operator =)
    TestClass& operator=(const TestClass&);
};

#endif // TESTCLASS_H
```

```cpp
#include "TestClass.h"

// Member functions (EX : getters and setters)
double TestClass::getValue() const
{
    return value;
}

void TestClass::setValue(double value)
{
    this->value = value;
}

// Member operators (EX : unary operator =)
TestClass& TestClass::operator=(const TestClass& a)
{
    value = a.value+10;

    return *this;
}
```

TestClass.h

TestClass.cpp

Static members in class

- Static variables
  - Shared by all objects
- Static functions
  - Have access to static members only
- Static members can be accessed by the class name
- 29.cpp 21.cpp

Operators

```cpp
#include <iostream>
#include "TestClass.h"

using namespace std;

int main(int argc, char* argv[])
{
    TestClass tst1;
    TestClass tst2;

    tst2.setValue(1.5);

    tst1 = tst2;

    cout << tst1.getValue() << endl;

    return 0;
}
```

Main.cpp

Sortie de l'application

HelloOperators

```
11.5
```

Résultat

Friend Fonctions

```
#ifndef TESTCLASS_H
#define TESTCLASS_H

#include <iostream>

class TestClass
{
private:
    double value;
public:
    // Member functions (EX : getters and setters)
    double getValue() const;
    void setValue(double);

    // Friend functions
    friend TestClass compute(const TestClass&, const TestClass&);
};

#endif // TESTCLASS_H
```

TestClass.h

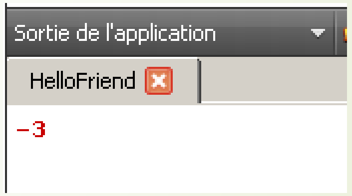
Friend Functions

```cpp
#include "TestClass.h"

// Member functions (EX : getters and setters)
double TestClass::getValue() const
{
    return value;
}

void TestClass::setValue(double value)
{
    this->value = value;
}

// Friend functions
TestClass compute(const TestClass& a, const TestClass& b)
{
    TestClass c;

    c.setValue(a.value + 2* b.value);

    return c;
}
```

TestClass.cpp

Functions Friends

```cpp
#include "TestClass.h"

using namespace std;

int main(int argc, char* argv[])
{
    TestClass tst1;
    TestClass tst2;

    tst1.setValue(1);
    tst2.setValue(-2);

    TestClass tst = compute(tst1, tst2);

    cout << tst.getValue() << endl;

    return 0;
}
```



Main.cpp

Inheritance

Classe Based

Derived Classes

```
// constructors and derived classes
#include <iostream>
using namespace std;

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (0);
  son daniel(0);

  return 0;
}
```

Appel à un constructeur spécifique de la classe de base

Sortie de l'application

HelloInheritance

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

Classes abstraites

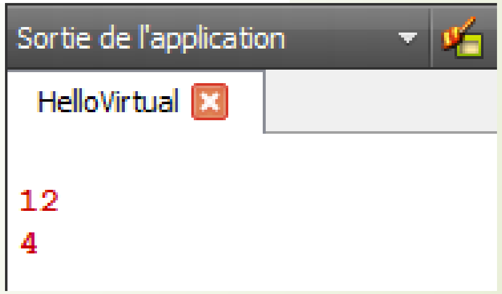
Abstract Based Classes

Derived Classes

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

class CPolygon {
protected:
    int area;
public:
    virtual int computeArea(void) = 0;
};

class CRectangle : virtual public CPolygon {
private:
    int width, height;
public:
    CRectangle(int w, int h) : width(w), height(h) {}
    int computeArea(void)    {
        area = width * height;
        return area;
    }
};

class CSquare : virtual public CPolygon {
private:
    int width;
public:
    CSquare(int w) : width(w) {}
    int computeArea(void)    {
        area = width * width;
        return area;
    }
};
```

Abstract Classes

```
int main() {
    CRectangle rect = CRectangle(3,4);
    CSquare square = CSquare(2);
    CPolygon * pPoly1 = &rect;
    CPolygon * pPoly2 = &square;
    cout << pPoly1->computeArea() << endl;
    cout << pPoly2->computeArea() << endl;
    return 0;
}
```



Empty constructor & Copy constructor

- Empty constructor
  - The default constuctor with no parameters when an object is created
  - Do nothing: e.g. Examp::Examp(){}
- Copy constructor
  - Copy an object (shallow copy)
  - The default consturctor when an object is copied (call by value, return an object, initialized to be the copy of another object)
    - 22.cpp {try not to pass an object by value)

Virtual function & overriding

- Define a member function to be virtual
- Use pointer/reference/member functions to call virtual functions
- Dynamic binding
  - Time consuming
- The constructor cannot be virtual
- Must be a member function

Pure virtual functions & abstract class

- Pure virtual functions
  - A function declared without definition
  - **virtual ret_type func_name(arg_list)= 0;**
- Abstract class
  - A class contains one or more pure functions
  - Can not be instantiated
  - Can be used to define pointers

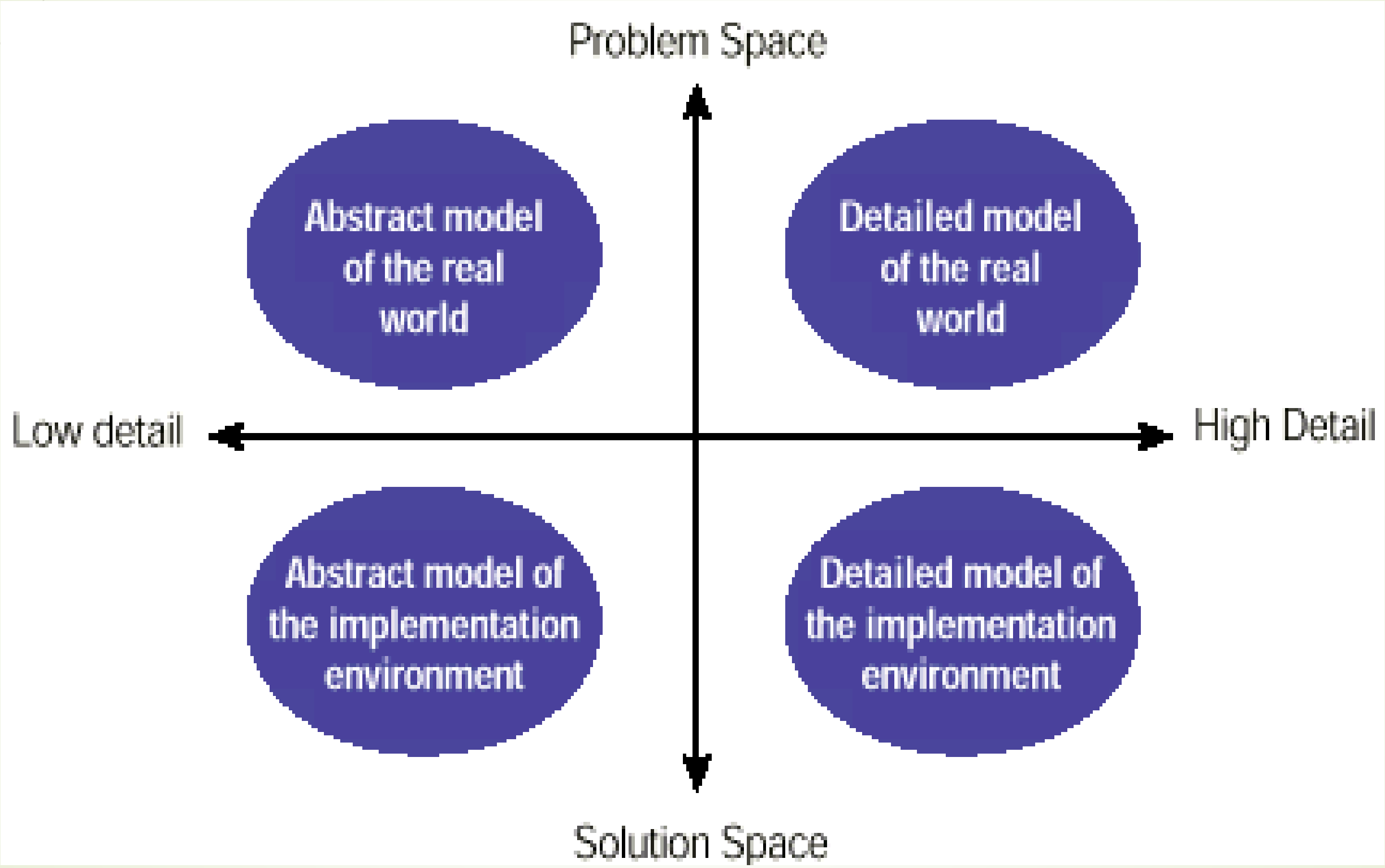
Template specialization

- An example from www.cplusplus.com
- **template <> class** *class_name* **<***type***>**

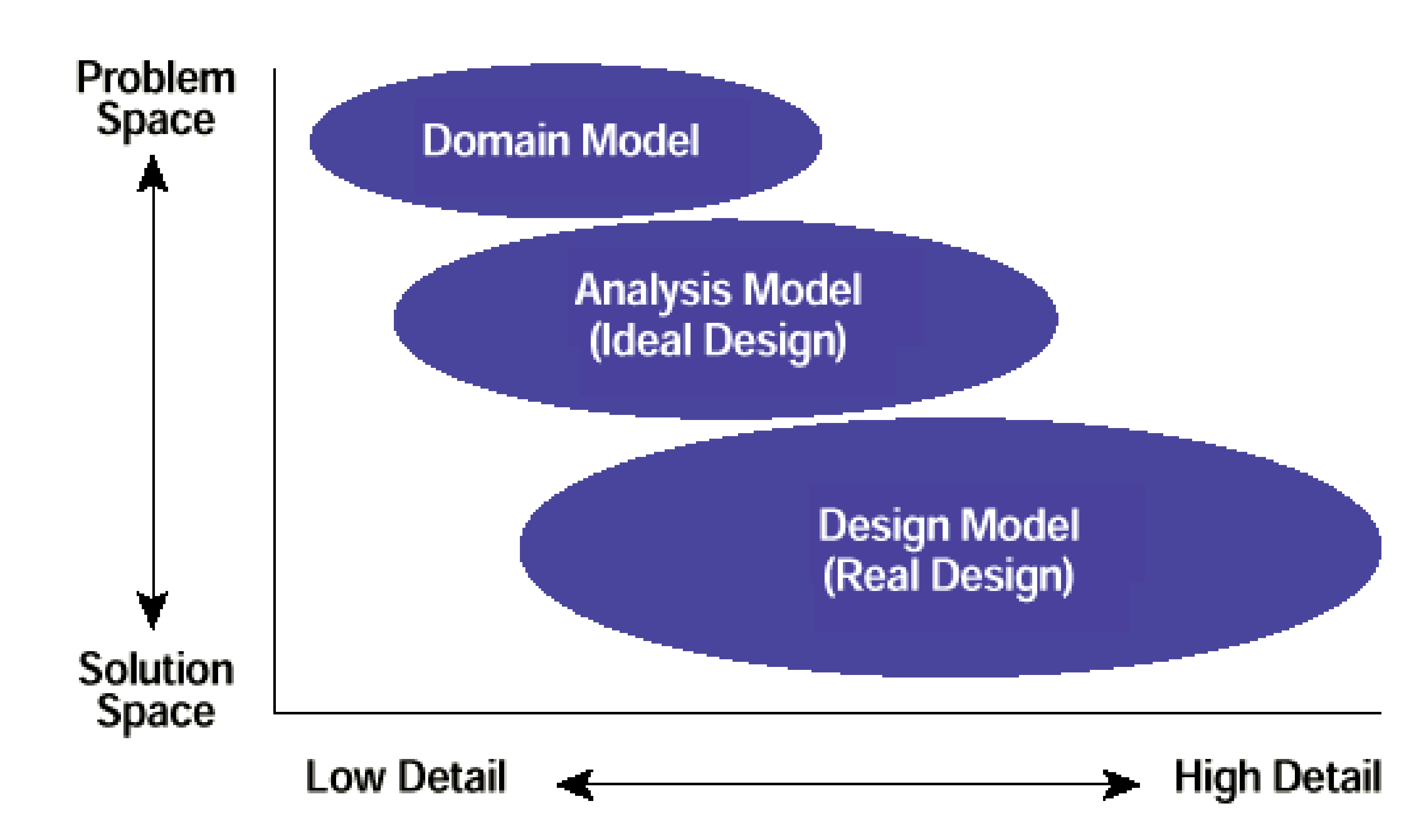
Default value for templates parameters

- template <class T = char> // With a default value.
- template <int Tfunc (int)> // A function as parameter.
- the implementation (definition) of a template class or function <u>must</u> be in the same file as the declaration.

Analysis and Design Space



Analysis and Design Space - Cont'd



Use Cases

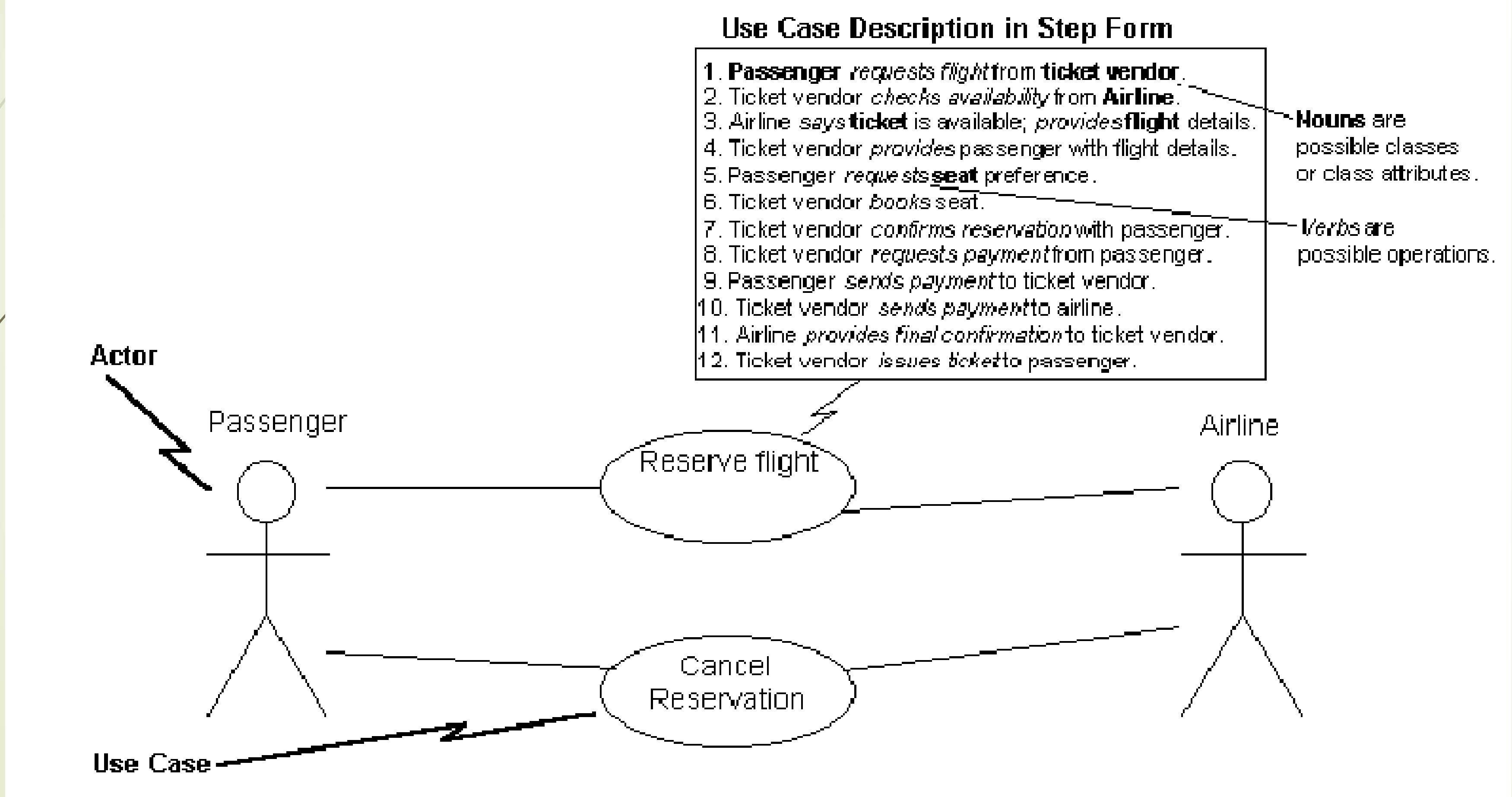
Use cases describe the basic business logic of an application.

- Use cases typically written in structured English or Diagrams
- Represent potential business situations of an application
- Describes a way in which a real-world actor – a person, organization, or external system – interacts with the application.

For example, the following would be considered use cases for a university information system:

- Enroll students in courses
- Output seminar enrolment lists
- Remove students from courses
- Produce student transcripts.

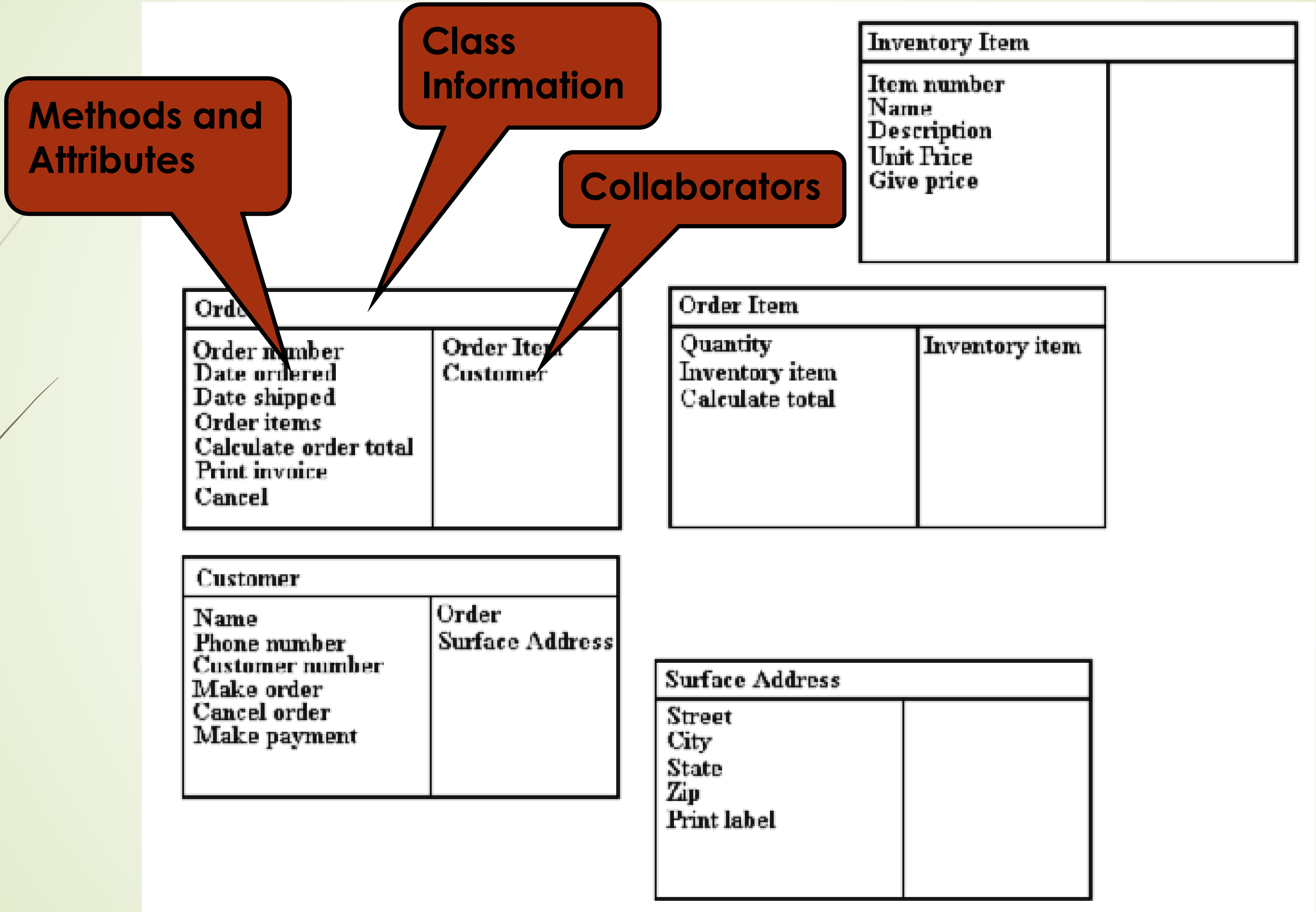
Use Cases Diagrams



Class Responsibility Collaborator Cards

- A CRC model is a collection of CRC cards that represent whole or part of an application or problem domain
- The most common use for CRC models is to gather and define the user requirements for an object-oriented application
- The next slide presents an example CRC model for a shipping/inventory control system, showing the CRC cards as they would be placed
- Note the placement of the cards: Cards that collaborate with one another are close to each other, cards that don't collaborate are not near each other

CRC Example



Methods and Attributes
Class Information
Collaborators
Inventory Item
Item number
Name
Description
Unit Price
Give price
Order Item
Customer
Order number
Date ordered
Date shipped
Order items
Calculate order total
Print invoice
Cancel
Order Item
Quantity
Inventory item
Calculate total
Inventory item
Customer
Name
Phone number
Customer number
Make order
Cancel order
Make payment
Order
Surface Address
Surface Address
Street
City
State
Zip
Print label

CRC Card Layout

Class Name:	
Parent Class:	Subclasses:
Attributes: Responsibilities:	Collaborators (Sends Messages to):

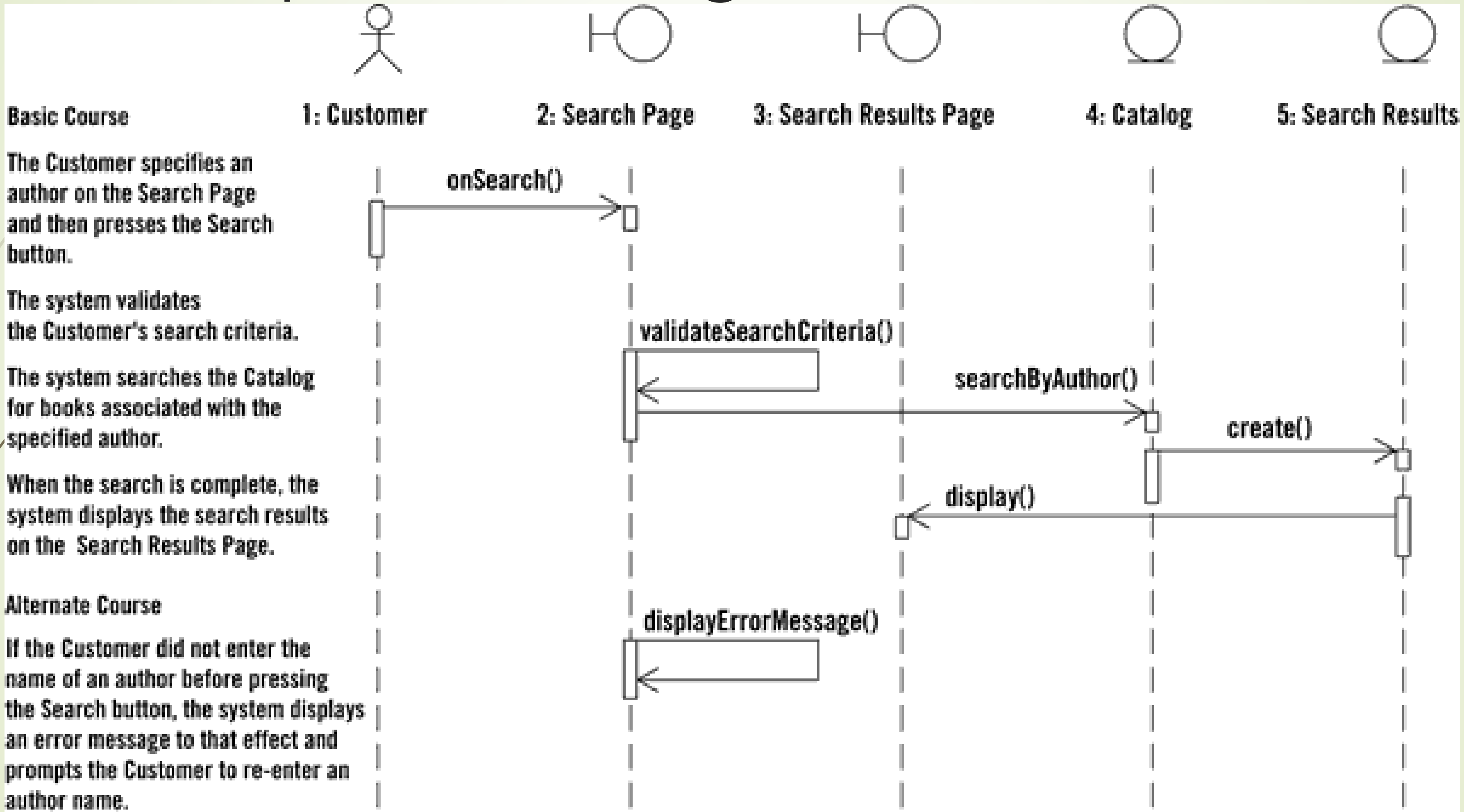
Sequence Diagrams

Traditional sequence diagrams show:

- The objects involved in the use case
- The messages that they send each other
- Return values associated with the messages

Sequence diagrams are a great way to review your work as they force you to walk through the logic to fulfill a use-case scenario and match the responsibilities and collaborators in CRC cards.

Sequence Diagrams

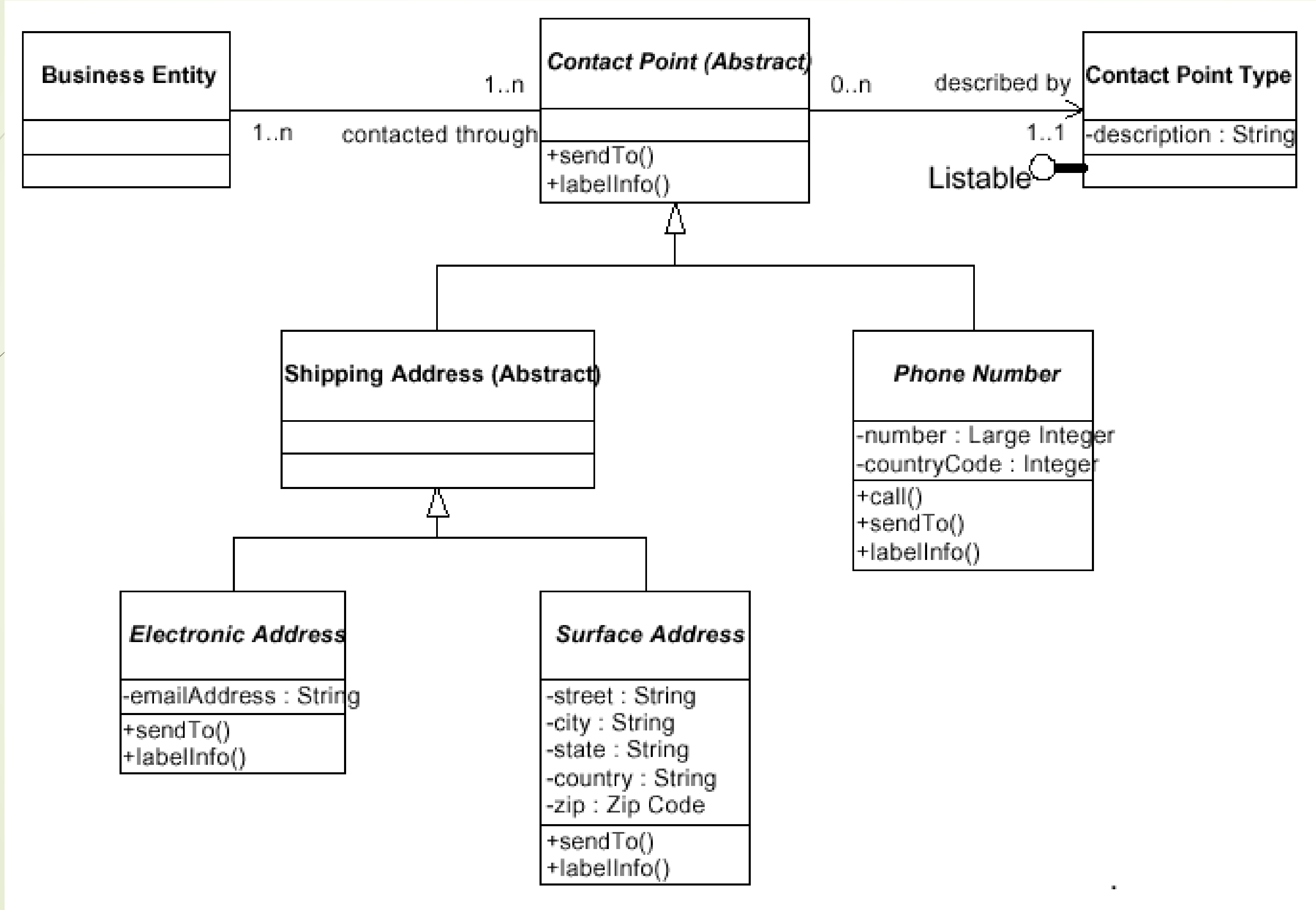


Class Diagrams

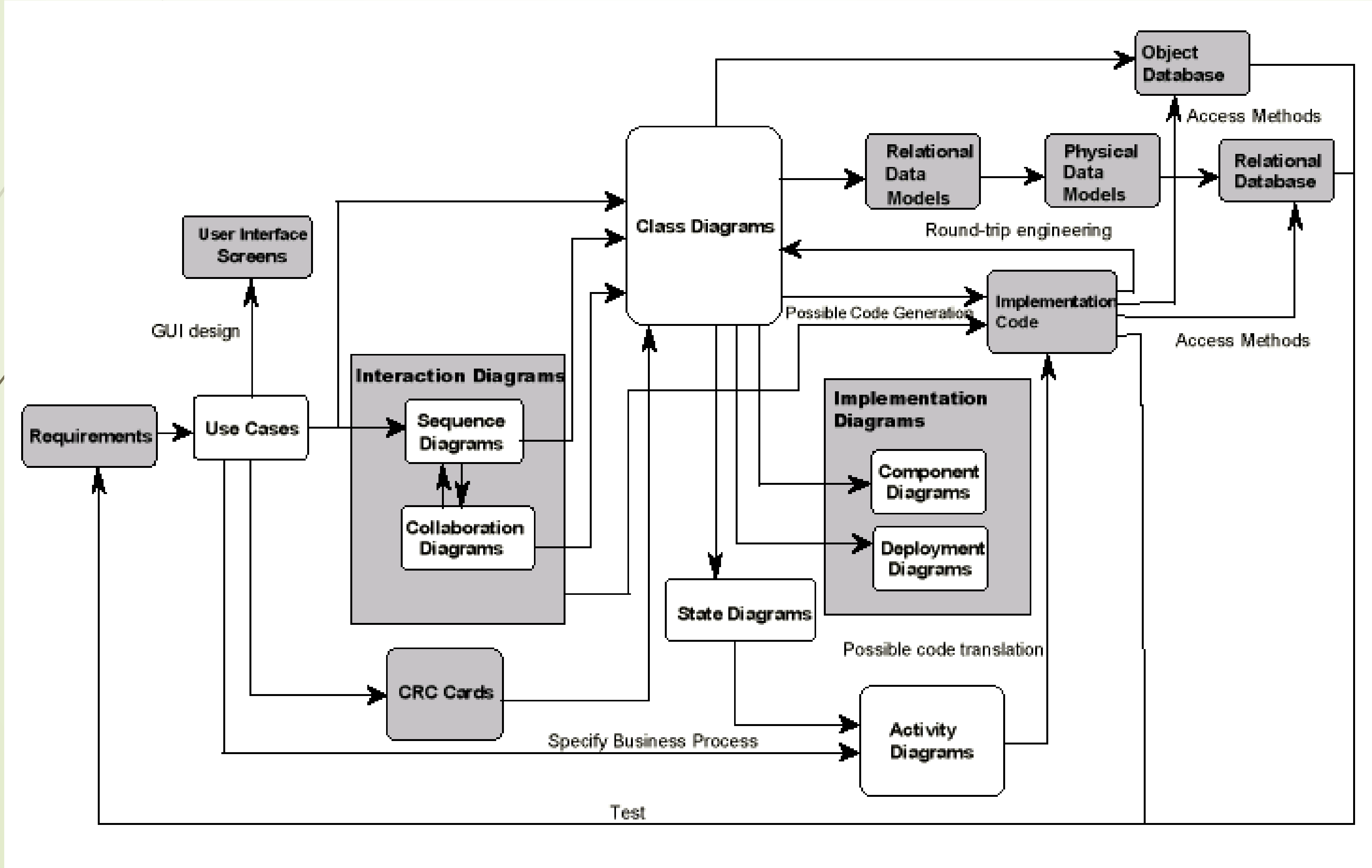
Class diagrams (object models) are the mainstay of OO modeling

- They are used to show both what the system will be able to do (analysis) and how it will be built (design)
- Class diagrams show the classes of the system and their interrelationships
  - Inheritance
  - Aggregation
  - Associations

Class Diagram



UML Models



Object Database
Access Methods
Relational Data Models
Physical Data Models
Relational Database
Class Diagrams
Round-trip engineering
User Interface Screens
Implementation Code
Possible Code Generation
Access Methods
GUI design
Interaction Diagrams
Sequence Diagrams
Collaboration Diagrams
Implementation Diagrams
Component Diagrams
Deployment Diagrams
Requirements
Use Cases
State Diagrams
Possible code translation
CRC Cards
Activity Diagrams
Specify Business Process
Test

Annexes

- "!" exclamation mark or exclamation point
apostrophe (’ ')
brackets ([ ], ( ), { }, ⟨ ⟩)
colon (:)
comma (, ، 、)
dash (‒, –, —, ―)
ellipsis (…, ..., . . .)
exclamation mark (!)
full stop/period (.)
guillemets (« »)
hyphen (‐)
hyphen-minus (-)
question mark (?)
quotation marks (‘ ’, “ ”, ' ', " ")
semicolon (;)
slash/stroke/solidus (/, ⁄)ampersand (&)
asterisk (*)
at sign (@)
backslash (\ )
bullet (•)
caret (^)
dagger (†, ‡)
degree (°)
ditto mark (〃)
inverted exclamation mark (¡)
inverted question mark (¿)
number sign/pound/hash (#)
numero sign (№)
obelus (÷)
ordinal indicator (º, ª)
percent, per mil (%, ‰)
basis point (‱)
pilcrow (¶)
prime (′, ″, ‴)
section sign (§)
tilde (~)
underscore/understrike (_)
vertical bar/broken bar/pipe (¦, |)