- **Expressions and Assignment Statements: Ch. 7.1-7.2 (excluding 7.2.1.4-7.2.1.5), 7.3-7.8**
- **Statement-Level Control Structures: Ch. 8.1-8.4 (excluding 8.3.4)**

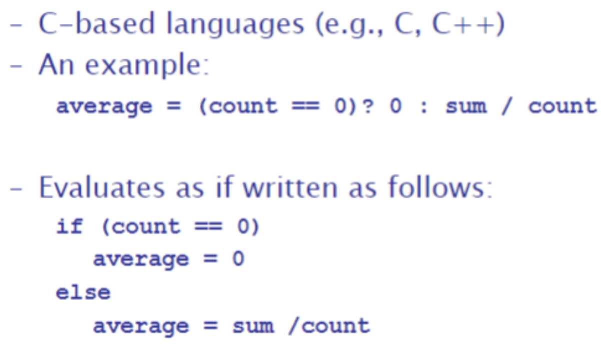
NOTE: MISSING DIFFERENCES BETWEEN LANGUAGES INFORMATION

Ch 7 and 8 Expression and statements
Expressions: Fundamental means of computations in a programming language. Operator and operand orders. Essence of imperative languages: is the dominant role of assignment statements.

Arithmetic Expressions: Motivation for developing early programming languages. Consists of operators/operands/parentheses and function calls. Ex: x+6.infix Some have + x 6. (prefix) Most unary operators have prefix, but -- ++ in c languages can be either.

Some design issues include precedence/associativity/evaluation side effects/overloading.
Unary Operator: one operand, Binary Operator 2 operands, ternary operator 3 operands
Operator Precedence rules: parentheses, unary operators, ** *,/ +,-

Left to right except ** which is right to left.. APL all operators are equal precedence.
Parentheses can override precedence



- C-based languages (e.g., C, C++)
- An example:

```
average = (count == 0)? 0 : sum / count
```

- Evaluates as if written as follows:

```
if (count == 0)
   average = 0
else
   average = sum /count
```

Variable: fetch Value from memory

Constants: Fetch from memory/machine language instruction

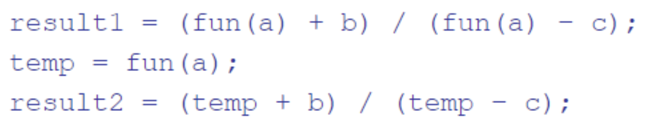
Parenthesized: evaluate all operands and operators

When Operand is function call is more interesting.

Function side effects: when a function changes a two way parameter or a non local variable. (can make it harder to track values)

Two ways to solve the problem: no two way parameters or non-local references. Advantage is that it works but becomes inflexible.

Or we write the definition to demand evaluation order is fixed, but it may limit some compiler optimizations.

Referential transparency: 

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

result1=result2 <- two expression in program have the same value submitted for them anywhere in program without affecting program.

Advantage: Semantics of a program is easier to understand.

Functions cannot have state, and outside values must be constant. So value of a function depends on parameters

Use of an operator for more than one purpose is called operator overloading.

Some are common (+ for in and float) some are trouble (* in c C++) will cause loss of compiler error detection. And perhaps readability. C+++ C# F# allow user defined overloaded operators. This can help operators to increase readability. But it can result in nonsense operations and readability may suffer

Type Conversions:

narrowing conversion: converts an object to a type that cannot include all values of original type ex. Float to int (value loss)

Widening conversion: is one which an object is converted to a type that can include at least approximations to all values of the original type. Ex: int to float (at the very least maintained values)

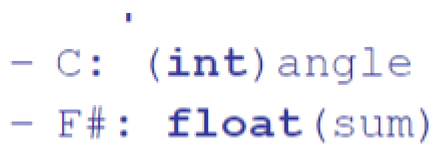
Mixed Mode expression: expression with operands of different types

Coercion: an implicit type of conversion. Decreases type error detection in compilers.

Most languages have all numeric types coerced into expressions using widening conversions.

ML and F# don't have coercions.

Explicit Type conversion: called casting in C languages. Some languages check certain casts during runtime.



```
– C:  (int)angle
– F#:  float(sum)
```

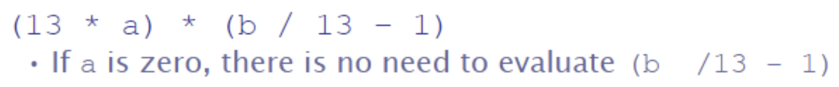
<-example of casting

Errors in expressions: Inherent limitations of arithmetic (you cannot divide by 0) and limited by computer arithmetic eg overflow. It's often ignored by the run time system.

Relational Expressions: Use relational operators and operands. Evaluates to some boolean representation. Operator symbols vary among languages. Java and PHP both have ===, and !== which are similar to == != but don't coerce.

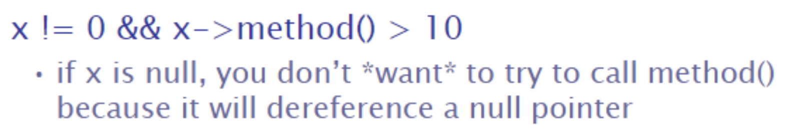
Boolean expressions: Operands are boolean/result is boolean.

Short circuit evaluation: an expression in which result is determined without looking at all operands/operators.



```
(13 * a) * (b / 13 - 1)
```

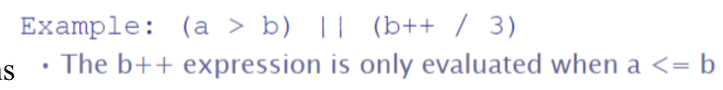
- If a is zero, there is no need to evaluate `(b  /13 - 1)`



x != 0 && x->method() > 10

- if x is null, you don't *want* to try to call method() because it will dereference a null pointer

Used by C C++ and java, good for boolean operators, but also bitwise bool operators. Short Circuit has side effects in expressions



```
Example: (a > b) || (b++ / 3)
```

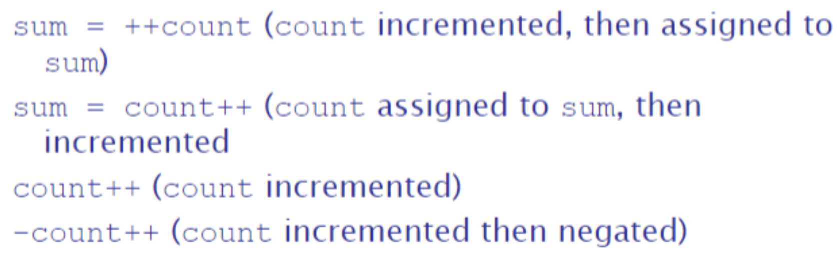
- The b++ expression is only evaluated when a <= b

Assignment Statements: The general syntax: <target_var> <assign_operator> <expression>, C uses = sign, can be bad when it is overloaded for relational operator for equality (which is why c language uses == for relational operator and not =)

Compound Assignment Operators: shorthand method of specifying a needed form of assignment.

Ex: a = a + b can be written as a += b

Unary Assignment Operators: combines increment and decrements operations with assignments.



```
sum = ++count (count incremented, then assigned to
  sum)
sum = count++ (count assigned to sum, then
  incremented
count++ (count incremented)
-count++ (count incremented then negated)
```

Assignment as expression: assignment statement produces result and can be used as operand. But can cause side effects.

Mixed Mode Assignments:C++ and others: any numeric type of value can be assigned to any numeric type variable, Java and C# only have widening assignment coercions done.

Chapter 8

Levels of control flow: within expressions, program units and program statements.

Program statements for this chapter

Control statements were based on IBM hardware, a lot of debate but ir proved algorithms represented by flowcharts can be coded with only two way selection and pretest logical loops.

Sequence of statements: the semantics of a sequence of several statements, is that they are executed in order. Each statement may change the contents of memory "State of the program", blocks of statements (compound statements enclosed in {} or begin/end are treated as a single statement in most grammars.

Control structure: control statement and the statements whose execution it controls.

Selection statements: provides the means of a choice between two or more paths of execution.

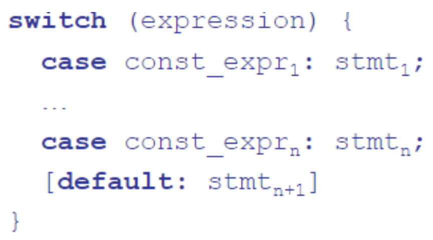
Two types: two way or multiple way selectors.

Two way generally use if (control expression) then (first path) else (path)

Control Expression:reserved word for the clauses placed in parentheses. Some languages require it to be parenthesized.

Clause Form: then and else claus can be single or compound statements in many languages.

Multiple way selection Statements: allow the selection of one of any number of statements or statement groups. EX:



```
switch (expression) {
  case const_expr_1: stmt_1;
  ...
  case const_expr_n: stmt_n;
  [default: stmt_n+1]
}
```

Design Choice for switch statement: Control expression can only be an integer, electable segments can be statement sequences/blocks/compound statements. Any number of segments can be executed in one execution. Default clauss is for unrepresented values it is necessary for the statement to function.

Implementing multiple selectors: multiple condition breaks, store case value sin a table and use linear search of the table, use a hash table of case values for 10+ cases, an array for small number of cases with most being represented.

Iterative Statements: the repeated execution of a statement or compound statement via iteration or recursion.

Counter Controlled Loops: A counting iterative statement has a loop variable and a means of specifying the initial/terminal/stepsize values. Can be whole statements/statement sequences for C languages. The value of a multiple statement expression is the value of the last statement in expression. If second expression is absent : infinite loop.
Design choice: no explicit loop variable, everything can be changed in loop, first expression is evaluated once, the others are evaluated each iteration. You can branch into the body of a for loop(in c). In C++ the control expression can be boolean and the initial expression can include variable definitions.
Logically Controlled Loops:Repetition based on Boolean Expression. C and C++ have both pretest and posttest forms in which control expression can be arithmetic. Java must be boolean.
User Locate Loop control Mechanisms: sometimes convenient for programmers to decide location for loop control. Simple design for simple loops (ex: break). Some also have (continue) to skip the remainder of iteration but doesn't exit loop.
Iteration based on data structures: # of elements in data structures controls loop iteration, .Control Mechanism is a call to an iterator function: that returns the next element in a chosen order if there is one or the loop ends.
Unconditional Branching: Transfers execution control to a specified place in program, major concern with readability, some languages do not support and loop exit statements are restricted.

CHAPTER 9 AND 10

Subprograms and their implementation

Subprograms are fundamental blocks in programming, sometimes called functions, subroutines, or methods.

Two fundamental Abstraction facilities: process Abstraction, and Data Abstraction

Each subprogram has a single entry point, the calling program is suspended during execution of the called subprogram. Control always returns to the caller when the called subprogram's execution terminates. Only one subprogram is running at a time. Calling a function involves an interaction between caller of function and called function.

Subprogram definition: describes the interface and actions of the subprogram abstraction. In python function definitions are executable, in all others they are not.

A subprogram call is an explicit request that the subprogram executed

Subprogram header: first part of the definition, including the name kind of subprogram and parameters.

Parameter profile (also known as signature) of a program is its number/order/and type of its parameters.

Protocol: subprogram's parameter file, if its a function then its return type.

Functions declarations in c C++ are often called prototypes.

Subprogram Declaration: provides protocol but not body of subprogram.

Formal parameter: a dummy variable listed in subprogram header and used in subprogram.

Actual Parameter: value or address used in subprogram call statement

Actual/Formal Parameter Correspondence: Positional: the binding of actual parameters to formal parameters is by position, the first actual parameter bound to first formal parameter and so on. Safe and effective. Keyword: The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter. Advantage is that parameters can appear in any order, reducing the parameter correspondence errors. The con is that the user must know the formal parameter's names.

Certain languages allow formal parameters to have default values. Variable numbers of parameters can be accepted as long as they are of the same type, the corresponding formal parameter is an array preceded by params.

Two categories of subprograms: procedures are a collection of statements that define parameterized computations . Functions structurally resemble procedures but are semantically modeled on mathematical functions. They tend to not have side effects, program functions have side effects.

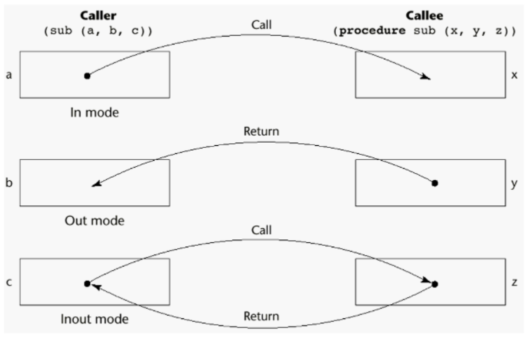
Local variables can be stack Dynamic which support recursion and storage for locals is shared among some sub programs, but allocation and deallocation/initializing time becomes a factor, indirect addressing also happens, subprograms also can no longer be history sensitive.

Local variables can also be static, and this has the exact opposite pro/cons of stack dynamic.

In most languages locals are stack dynamic.

Semantic models of parameter passing: inmode: copy parameter into a function at beginning.

Outmode: copy a parameter out of a function when it completes inout mode: does both.



Conceptual models of transfer: physically move a value, move an access path to a value (via pointer or reference parameter). Implementation models of parameter passing (7 models for the three basic transmission models)

Pass by pass value(in mode): the value of the actual parameter is used to initialize corresponding formal parameters. Normally implemented by copying, can be implemented by transmitting an access path but not recommended(write protection not easy)
Disadvantage is that if physically moved then additional storage is required and the actual move can be costly for large parameters and if its moved by access path it must write-protect in the called subprogram and access costs more.
Pass By result (out mode): in many languages the result of executing a function is passed out of the function by using "return". Void return means nothing, prototype for function has return type. In languages that support outmode parameters: an outmode parameter passes no value to subprog, the corresponding formal parameter acts as a local variable, its value is transmitted to callers actual parameter when control is returned to called by physical move. Requires extra storage location and copy operation, with potential problems when parameter copied back.
Pass by Value-Result(inout)also called pass by copy, combination of both above, formal parameters have local storage, has the disadvantage of pass by result and passby value
Pass by Reference (inout): pass an access path (pointer/reference) also called point by sharing, passing process is efficient (no copying and duplication) but its slower access to formal parameters and possible side effect collisions, as well as unwanted aliases can cause bad readability.
Pass by name(inout): by text substitution, actual parameter is text substituted for corresponding formal parameter in all occurrences. Does not correspond to a single implementation model. Formals are bound to an access method at the time of the call but actual binding to a value or address takes place at the time of reference/assignment. Allows flexibility, requires subprogram

and the referencing environment of calls are passed with parameters so actual parameters can be calculated and accessed.

Most languages communicate during runtime stack, pass by reference is the simplest to implement only an address placed in stack.

C – Pass-by-value – Pass-by-reference is achieved by using pointers as

parameters

C++ – A special pointer type called reference type for pass-by-reference

Java – All non-object parameters are passed are passed by value So, no method can change any of these parameters – Object parameters are passed by reference

Type Checking parameters: considered very important for reliability.

Multi-dimensional array as parameter: if passed to subprogram and subprogram is separately compiled, the compiler needs to know the declaration size of array to build storage mapping function. Requires programer to include declared sizes of all but first subscript in the actual parameter. They are arrays of arrays stored in row major order. Disallows flexible subprograms due to matrix. Passing a pointer to array fixes this.

Design Considerations for parameter passing: Efficiency and oneway/twoway data transfer, but these are in conflict as good programing suggest limited access meaning use one way, but pass by reference is more efficient to pass structures of significant size.

It is sometimes convenient to pass subprograms as parameters.

Shallow Binding: Environment of call statment that enacts the subprogram is the referenced enviroment (most natural for dyanmic scope)

Deep BindingL environment of definition of subprogram (most natural for static scope)

Ad Hoc Binding: environment of the call statement that passed subprogram

When multiple subprogs are called to be run it is unknown which until execution.

Overloaded subprograms: has same name as another subprog in same environment

Generic/polymorphic subprogram: takes parameters of different types on different activations.

Overloaded subprogs provide ad hoc polymorphism.

Subtype Polymorphism: a variable of type T can access any object of type T or derived from it.

Parametric Polymorphism: subprog takes generic parameter used in type expression that describes type of parameters in subprog.

General semantics of calls and returns : are together subprogram linkage

Genreal semantics of calls to subprog: parameter passing,stack dynamic allocation, save execution status, transfer of control and arrange for return. General semantics of subprog returns: outmode inoutmode parameters returned by copy, deallocation of stack dynamic locals, restore execution status and return control to caller.

“Simple” subprogs: Subprogs cannot be nested, all local variables static, doesn’t support recursion. Save execution status of caller, Compute and pass parameter, pass return address to called, transfer control to called.

The format/layout of a none code part of an executing subprog is called an activation record, activation record instance is concrete example of activation record.

All activation records can be attached to code segments. Subprog units may be compiled separately.

For implementing subprogs with stack sdyanmic more complex activation records needed. Recursion must be supported. Activation record instance dynamically created when subprog is called, and resides on runtime stack.

Environment Pointer: must be maintained by runtime system. Always points at base of activation record instance currently executing programing unit. EP is not stored in runtime stack.

Caller actions: creat activation record instance save execution status of prog unit.

Dynamic chain and local offset: collection of dynamic links in the stack at a given ime is called the dynamic chain or call chian, local variables can be access by their offset from the beginning of activation record. Determined by compiler at compile time.

Blocks are user specific local scopes. Advantage in temp is that it cannot interfere with any other variable with same name.

Implement blocks in two ways: parameter-less subprog that are always called from same location, or since the max storage for block can be determined, this amount of space can be allocated after the local variables in the activation record.

CHAPTER 11

Abstract data types and encapsulating constructs

Abstraction: a view or representation of an entity that includes only the most significant attributes. This is fundamental to programming and cs.

ADT: a data structure (ex: struct) which includes subprograms that manipulate its data. An instance of an ADT is called an object. All built in data types are abstract. User Defined ADT provides the same characteristics of language defined types.

An abstract data type: -user defined data type that is a representation of objects of the type is hidden from prog units that use those objects, so only operations provided in definition are possible. -Declarations of the type and protocols of operations on objects of type are contained in a single syntactic unit.

Advantage of first condition: reliability by hiding representations and reduces range of code and variables of which programmers must be aware. Less name conflicts.

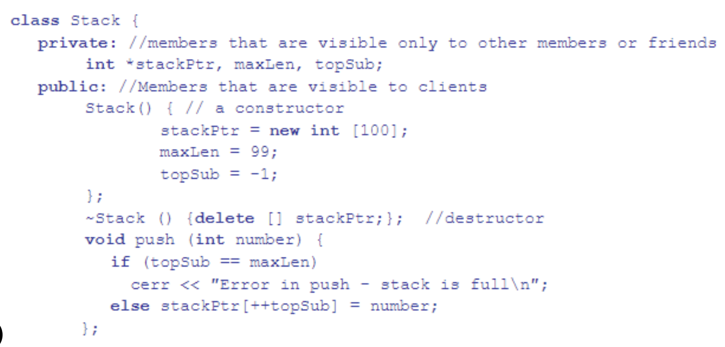
Advantage of second: provides a method of programing organization and aids modifiability. Wit seperate compilation

Language requirements for ADT's: a syntactic unit for type definition. A method of making names and headers visible to client while hiding actual def. Some primitive operations must be built onto processor.

C++: based on C struct type, class is type, all of class instances of a class chare a single copy of member. Each instance of class has own copy. Instance can be static, stack dynamic, heap dynamic. Private claus for hidden intenties, Public clause for interface entities, protected clause for inheritance.

Constructors: Functions to initialize the data members of instances (do not create though). May also allocate storage if part of object is heap dynamic. Can include parameters for parameterization. Can be explicitly called, name is same as class name.

Destructers: functions to cleanup after an instance is destroyed, usually just reclaim heap storage. Implicitly called when object's lifetime ends. Can be explicitly called. Name is class name by a tilde.(~)



```
class Stack {
   private: //members that are visible only to other members or friends
        int *stackPtr, maxLen, topSub;
   public: //Members that are visible to clients
        Stack() { // a constructor
                stackPtr = new int [100];
                maxLen = 99;
                topSub = -1;
        };
        ~Stack () {delete [] stackPtr;};  //destructor
        void push (int number) {
          if (topSub == maxLen)
            cerr << "Error in push - stack is full\n";
          else stackPtr[++topSub] = number;
        };
```

Stack is construct ~stack is destruct

Friend functions/classes: provide access to private members to some units/functions, needed in c++

Java is similar but all user defined types are classes.

Differences between languages.

Parameterized ADT's allow designing an ADT that can store any type elements, only an issue for static typed languages. Known as generic classes. Classes can be somewhat generic by writing **parameterized constructor function**s.

Stack element type can be parameterized.Generic Parameters must be classes. Java has Most generic types are the collection types. Eliminate need to cast objects or multiple types in structure. Generic collection class cannot store primitives and no indexing. C# similar but no wildcard and predefined array list stack queue.

Encapsulating constructs: two special needs for large progs. Some means of organization other then diving subprogs, some means of partial compilation (smaller units in the whole prog).

A solution to this is grouping subprogs that are related into a unit that can be separately compiled. This is known as encapsulating.

Nested subprogs: organizing progs by nesting subprogs definitions inside the logically larger subprogs that use them.

Encapsulation in C: files containing one or more subprogs can be independently compiled, interface placed in header file. (can't check types and problem with pointers)

Encaping in C++: can define header and code files similar to C, classes can be used for encapsulation. Friends provide a way to grant access to private members of class.

Naming encapsulations: Large programs define global names, need a way to diving into groups.

A naming encapsulation is used to create new scope for names. C++ has namespaces.

CHAPTER 12

Support for object oriented programming

Many object oriented programming (OOP) languages.

Some support procedural and data oriented programming. Some support functional program newer languages do not support paradigms but use their imperative structures. Some are pure oop language like ruby.

Three language features: Abstract data types, inheritance, dynamic binding of methods to call method definitions.

Inheritance: reuse results in productivity increases. ADT's are difficult to reuse. Inheritance allows new classes defined in terms of existing ones by allowing them to inherit common parts. Address both concerns of reuse and adt independence with a hierarchy.

ADT's are usually classes, class instances are objects, class that inherits is a derived class or a subclass. The class from which another class inherits is a parent/super class.

Subprogs defined operations on objects are called methods.

Call to methods are called messages.

An entire collection of methods are called message protocol/message interface.

Messages have two parts- a method name and the destination object.

In the simplest cas a class inherits all of the parent's entities.

Inheritance can be complicated by access control to encapsulated enttiies. A class can hide entities form subclasses/clients. Or just clients. Class can also modify an inherited method, new one overrides inherited one. Method of parent is overridden.

Three ways a class can differ from its parent: subclass can add variable and/or methods to those inherited from the parent. Subclass can modify the behavior of one or more its inherited methods. Parent can have some of its variables/methods have private access, no longer visible to subclass.

There are two kinds of variables in a class: class variables one/class and instance variables one/object. Two kinds of methods class methods (messages to class) and instance methods (accept messages to objects)

Single verse Multiple inheritance: disadvantage of inheritance for reuse-creates interdependencies among classes that complicate maintenance.

Dynamic binding: a poly morphic variable can be defined of a class type that is able to reference object of that class and objects of any descendants.

When class hierarchy includes classes that override methods and such methods are called via poly-variable, the binding to the correct method is dynamic. Allows software systems to more easily extend during both development and maintenance.

Dynamic binding concepts: abstract method is one that does not include a definition

An abstract class is one that defines at least one abstract method

Abstract class can not be instantiated.

Design issues for OOP: Exclusivity/single multiple inheritance/object allocation decallation/dynamic and static binding/nested class/ initialization of objects.

Exclusivity of objects: everything is an object, allowing elegance and purity, but slow operation on simple objects.

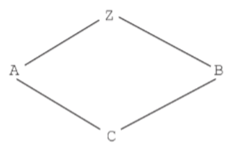
Add objects to complete typing system: fast operations on simple objects, but may result in confusing type system.

Include an imperative style typing system for primitive, but make all structure type objects, faast operations on simple objects and relatively small typing system. Results in fast operations on simple objects and relatively small typing system, but has confusion because of two systems.

Are subclasses subtypes: does an is-a relationship hold?

If a derived class is a parent class then objects of class must behave the same parent object. A derived class ia subtype if it has an is-a relationship with its parent class, subclass can only add variables and methods and overide inherited methods in ocmpatiable ways.subclasses inherit implementation, subtypes inherit interface and behavior.

Multiple inheritance allows a new class to inherit from two or more classes. But causes language and implementation complexity with potential inefficiency. It can be very convenient and valuable.

Diamond Inheritance: 

z to ab ab to c

From where are objects allocated?

If they behave like adts then allocated anywhere, if they are heap dynamic then references can be uniform through a pointer or reference variable. Objects that are stack dynamic: object slicing becomes a problem bc copying of object value must be copied to target object space.

If all messages to methods are dynamic binded its inefficient, if none are you lose its advantages. Allow user to specify.

If a new class is needed by only one class: there is no reason to define so it can be seen by other classes. In some cases, new class is nested inside a subprog rather then another class.

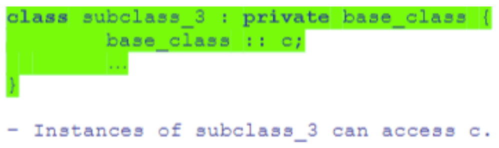
Support for OOP in C++: most widely used oop language.

Inheritance: class not need be the subclass of any class. Subclass can be declared with access controls wich define potential changes.

Private derivation: inherited public and protected members are private in subclass

Public Derivation: public and protected members are also public and protected in subclass.

ReExportation in C++: a member not accessible in subclass can be declared to be visible using



```
class subclass_3 : private base_class {
         base_class :: c;
         ...
}
```

- Instances of subclass_3 can access c.

scope resolution operator.(::)

Motivation for using private derivation: class provides members that must be visible os they are defined to be public; a derived class adds some new members but does not let clients see members of parent class, even though they had to be public by parent class definition

Multiple inheritance is supported in C++: if there are two inherited members with the same name, they can both be referenced using ::

Dynamic Binding: a method that can be defined to be virtual, which means that they can be called via polyvariables and be dynamically bound to messages

-pure virtual function has no definition, a class that has at least one pure virtual function is an abstract class, C++ does not allow a value variable to be polymorphic- pointer variable having type of class can be used to point to heap dynamic objects of any class derived publicly from base class making it polymorphic variable.

If objects are allocated from stack result is different.

Implementing OO constructs: two challenges, storage structures for instance variables and dynamic binding of messages to methods.

Class instance records (CIRs): store state of an object, they are static so built at compile time.

If a class has a parent teh subclass instance variables are added to parent CIR, because CIR is static, access to all instance variables is done as it is in records (efficient)
Dynamic Binding of methods call: methods in class are statically bound need no be involved in CIR, method dynamically bound must have entries in CIr. Call to dynamic bound methods can be connected via pointer in CIR. Storage Structure sometimes called Virtual method tables (vtables). Method calls can be represented as offsets from beginning of the vtable.