UNIT-V Architecture dependent code improvement

Instruction scheduling for pipeline:

When instructions are handled in parallel, it is required to detect and resolve dependencies between instructions. It can generally discuss dependency detection and resolution as it associates to processor classes and the processing functions contained independently.

An instruction is resource-dependent on an issued instruction if it needed a hardware resource that can be utilized by a previously issued instruction. If, for instance, only a single non-pipelined division unit is accessible, as in general in ILP-processors, thus in the code sequence the second division instruction is resource-dependent on the first one and cannot be implemented in parallel.

Resource dependencies are constraints generated by multiple resources including EUs, buses, or buffers. They can lower the degree of parallelism that can be managed at multiple phases of execution, including instruction decoding, issue, renaming, and execution, etc.

Approaches of Instruction Scheduling

There are two basic approaches as static and dynamic. **Static detection** and resolution are adept by the compiler, which prevents dependencies by reordering the code. Thus the output of the compiler is reordered into dependency-free code. VLIW processors continually expect dependency-free code, whereas pipelined and superscalar processors generally do not.

In contrast, **dynamic detection** and resolution of dependencies are implemented by the processor. If dependencies have to be recognized in connection with instruction issues, the processor generally maintains two gliding windows.

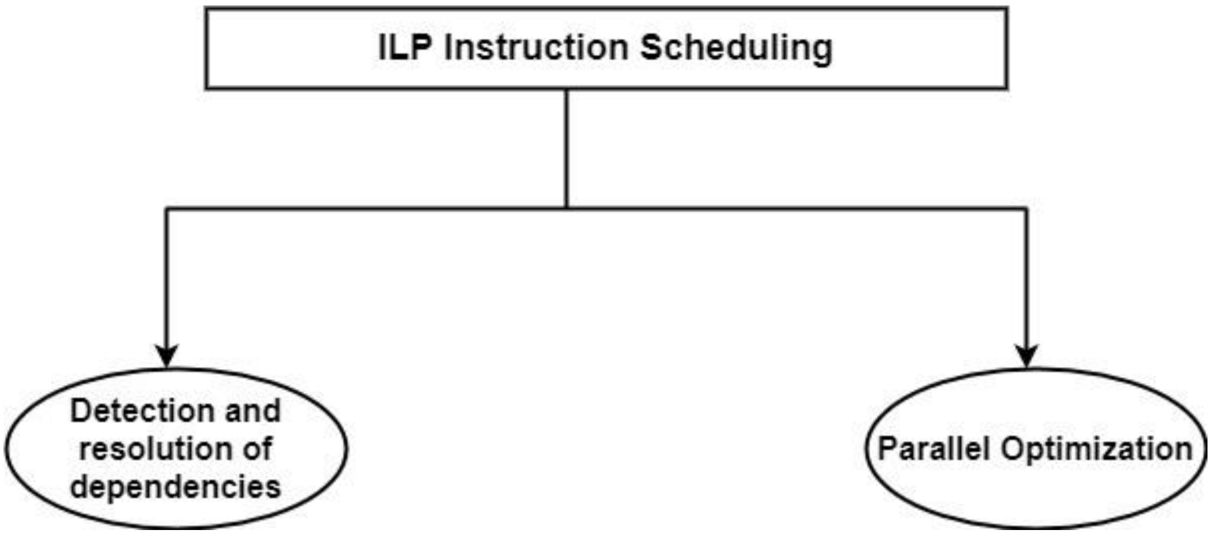
The issue window includes all fetched instructions which are predetermined for the issue in the next cycle, while instructions that are in execution and whose results have not been created are retained in an implementation window. In each cycle, all the instructions in the issue window are tested for data, control, and resource dependencies concerning the execution instructions.

Advantage of Dynamic Detection

- It allows handling some cases when dependencies are unknown at compile time.
- It facilitates the compiler.
- It enables code that was compiled with one pipeline in mind to run effectively on multiple pipelines.

The execution of an ILP-processor can be automatically raised by parallel code optimization. If dependencies are encountered and resolved statically by a compiler, certainly the compiler has to be improved to implement parallel code optimization as well.

If dependency detection and resolution are dynamic, code optimization has to be carried out previously by the compiler in connection with object code generation. Parallel optimization goes further than traditional sequential code optimization. It is managed by reordering the sequence of instructions by suitable code transformations for parallel execution.



In static scheduling promoted by parallel code optimization, it is the exclusive function of a compiler to detect information, control and resolve information, control, and resource dependencies during code generation. Further, the compiler also implements parallel code optimization. That is, the compiler makes dependency-free optimized code for parallel ILP-execution.

Loop optimization for cache memory:

Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.
Loop Optimization is a machine independent optimization.

Decreasing the number of instructions in an inner loop improves the running time of a program even if the amount of code outside that loop is increased.

Loop Optimization Techniques:

1. **Frequency Reduction (Code Motion):**
   In frequency reduction, the amount of code in loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.
   Example:
   Initial code:

```
while(i<100)
{
 a = Sin(x)/Cos(x) + i;
 i++;
}
```

   Optimized code:

```
t = Sin(x)/Cos(x);
while(i<100)
{
 a = t + i;
 i++;
}
```

2. **Loop Unrolling:**
   Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.
   Example:
   Initial code:

```
for (int i=0; i<5; i++)
    printf("Pankaj\n");
```

   Optimized code:

```
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
```

3. **Loop Jamming:**
   Loop jamming is the combining the two or more loops in a single loop. It reduces the time taken to compile the many number of loops.

   Example:
   Initial Code:

```
for(int i=0; i<5; i++)
    a = i + 5;
for(int i=0; i<5; i++)
    b = i + 10;

Optimized code:
for(int i=0; i<5; i++)
{
 a = i + 5;
 b = i + 10;
}
```

Register allocation:

Registers are the fastest locations in the memory hierarchy. But unfortunately, this resource is limited. It comes under the most constrained resources of the target processor. Register allocation is an NP-complete problem. However, this problem can be reduced to graph coloring to achieve allocation and assignment. Therefore a good register allocator computes an effective approximate solution to a hard problem.

Figure – Input-Output

The register allocator determines which values will reside in the register and which register will hold each of those values. It takes as its input a program with an arbitrary number of registers and produces a program with a finite register set that can fit into the target machine. (See image)

Allocation vs Assignment:

Allocation –

Maps an unlimited namespace onto that register set of the target machine.

- **Reg. to Reg. Model:** Maps virtual registers to physical registers but spills excess amount to memory.
- **Mem. to Mem. Model:** Maps some subset of the memory location to a set of names that models the physical register set.

Allocation ensures that code will fit the target machine's reg. set at each instruction.

Assignment –
Maps an allocated name set to the physical register set of the target machine.

- Assumes allocation has been done so that code will fit into the set of physical registers.
- No more than **'k'** values are designated into the registers, where 'k' is the no. of physical registers.

General register allocation is an NP-complete problem:

- Solved in polynomial time, when (no. of required registers) <= (no. of available physical registers).
- An assignment can be produced in linear time using Interval-Graph Coloring.

Local Register Allocation And Assignment:
Allocation just inside a basic block is called Local Reg. Allocation. Two approaches for local reg. allocation: Top-down approach and bottom-up approach.
Top-Down Approach is a simple approach based on 'Frequency Count'. Identify the values which should be kept in registers and which should be kept in memory.

Algorithm:

1. Compute a priority for each virtual register.
2. Sort the registers into priority order.
3. Assign registers in priority order.
4. Rewrite the code.

Moving beyond single Blocks:

- More complicated because the control flow enters the picture.
- Liveness and Live Ranges: Live ranges consist of a set of definitions and uses that are related to each other as they i.e. no single register can be common in a such couple of instruction/data.

Following is a way to find out Live ranges in a block. A live range is represented as an interval [i,j], where i is the definition and j is the last use.

Global Register Allocation and Assignment:
1. The main issue of a register allocator is minimizing the impact of spill code;

- Execution time for spill code.
- Code space for spill operation.
- Data space for spilled values.

2. Global allocation can't guarantee an optimal solution for the execution time of spill code.
3. Prime differences between Local and Global Allocation:

- The structure of a global live range is naturally more complex than the local one.
- Within a global live range, distinct references may execute a different number of times. (When basic blocks form a loop)

4. To make the decision about allocation and assignments, the global allocator mostly uses graph coloring by building an interference graph.
5. Register allocator then attempts to construct a k-coloring for that graph where ‘k’ is the no. of physical registers.

- In case, the compiler can’t directly construct a k-coloring for that graph, it modifies the underlying code by spilling some values to memory and tries again.
- Spilling actually simplifies that graph which ensures that the algorithm will halt.

6. Global Allocator uses several approaches, however, we’ll see top-down and bottom-up allocations strategies. Subproblems associated with the above approaches.

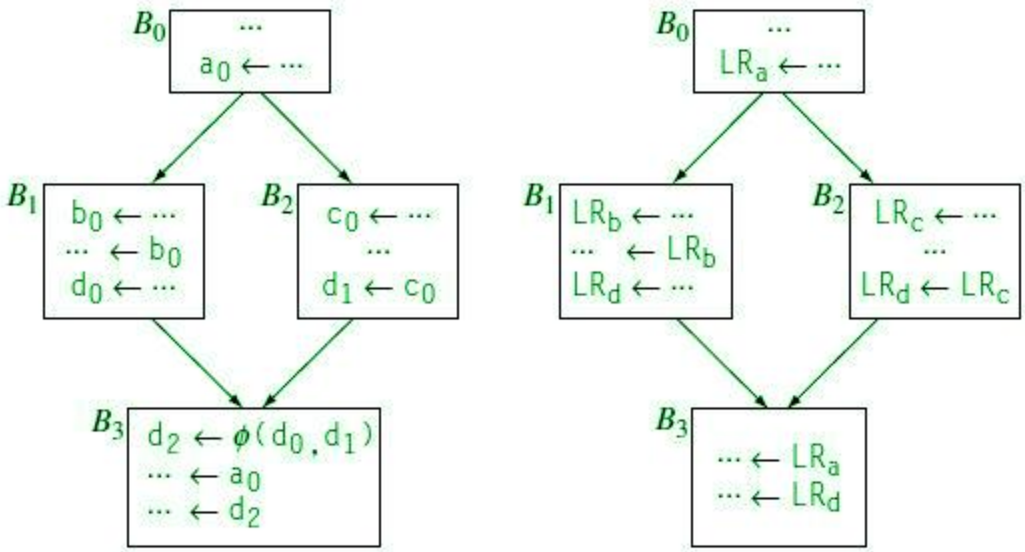
- Discovering Global live ranges.
- Estimating Spilling Costs.
- Building an Interference graph.

Discovering Global Live Ranges:
How to discover Live range for a variable?

```
1   loadI    ...        ⇒ r_arp
2   loadAI   r_arp,@a   ⇒ r_a
3   loadI    2          ⇒ r_2
4   loadAI   r_arp,@b   ⇒ r_b
5   loadAI   r_arp,@c   ⇒ r_c
6   loadAI   r_arp,@d   ⇒ r_x
7   mult     r_a,r_2    ⇒ r_a
8   mult     r_a,r_b    ⇒ r_a
9   mult     r_a,r_c    ⇒ r_a
10  mult     r_a,r_d    ⇒ r_a
11  storeAI  r_a        ⇒ r_arp,@a
```

	Register	Interval
1	r_{arp}	[1,11]
2	r_a	[2,7]
3	r_a	[7,8]
4	r_a	[8,9]
5	r_a	[9,10]
6	r_a	[10,11]
7	r_2	[3,7]
8	r_b	[4,8]
9	r_c	[5,9]
10	r_d	[6,10]

Figure – Discovering live ranges in a single block
The above diagram explains everything properly. Let’s take the example of Rarp, it’s been initialized at program point 1 and its last usage is at program point 11. Therefore, the Live Range of Rarp i.e. Larp is [1,11]. Similarly, others follow up.



(a) Code Fragment in Pruned SSA Form

(b) Rewritten in Terms of Live Ranges

Figure – Discovering Live Ranges

Estimating Global Spill Cost:

- Essential for taking a spill decision which includes – address computation, memory operation cost, and estimated execution frequency.
- For performance benefits, these spilled values are kept typically for the Activation records.
- Some embedded processors offer ScratchPad Memory to hold such spilled values.
- **Negative Spill Cost**: Consecutive load-store for a single address needs to be removed as it increases the burden, hence incurs negative spill cost.
- **Infinite Spill Cost**: A live range should have infinite spill cost if no other live range ends between its definition and its use.

Interference and Interference Graph:

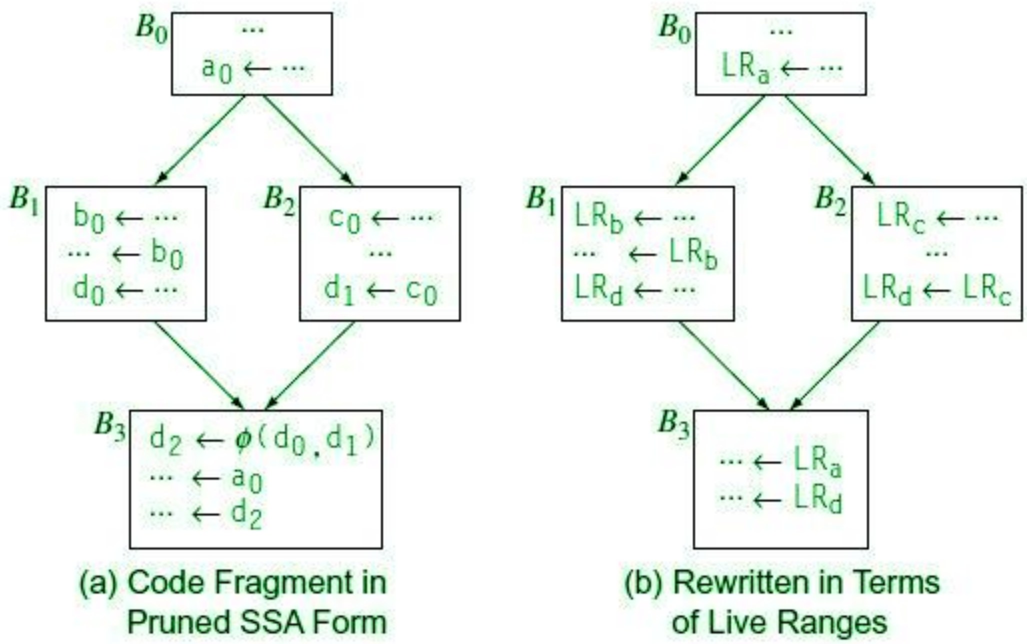


Figure – Building Interference Graph from Live Ranges

From the above diagram, it can be observed that the live range LRA starts in the first basic block and ends in the last basic block. Therefore it will share an edge with every other live Range i.e. Lrb, Lrc,Lrd. However, Lrb, Lrc, Lrd doesn't overlap with any other live range except Lra so they are only sharing an edge with Lra.

Building an Allocator:

- Note that a k-colorable graph finding is an NP-complete problem, so we need an approximation for this.
- Try with live range splitting into some non-trivial chunks (most used ones).

Top-Down Colouring:

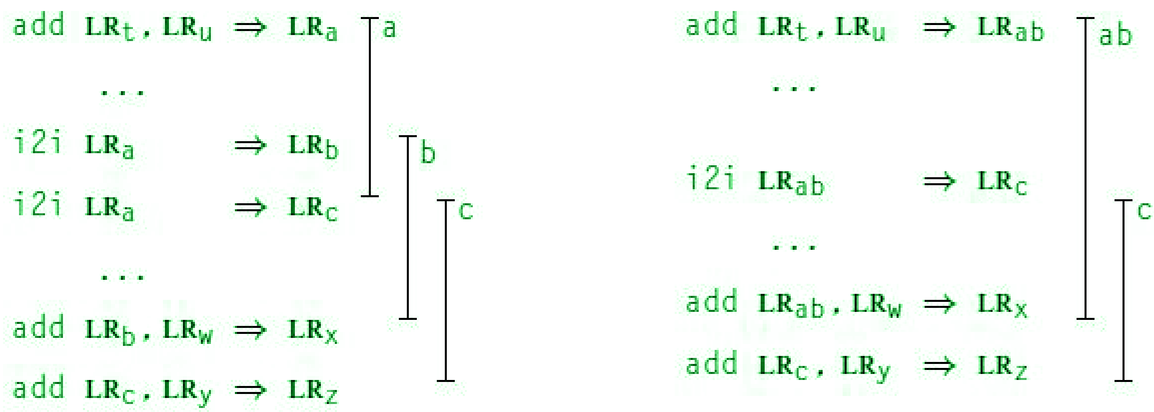
1. Tries to color live range in an order determined by some ranking functions i.e. priority based.
2. If no color is available for a live range, the allocator invokes either spilling or splitting to handle uncolored ones.
3. Live ranges having k or more neighbors are called constrained nodes and are difficult to handle.
4. The unconstrained nodes are comparatively easy to handle.
5. **Handling Spills:** When no color is found for some live ranges, spilling is needed to be done, but this may not be a final/ultimate solution of course.
6. **Live Range Splitting:** For uncolored ones, split the live range into sub-ranges, those may have fewer interferences than the original one so that some of them can be colored at least.

Chaitin's Idea:

- Choose an arbitrary node of (degree < k) and put it in the stack.
- Remove that node and all its edges from the graph. (This may decrease the degree of some other nodes and cause some more nodes to have degree = k, some node has to be spilled.
- If no vertex needs to be spilled, successively pop vertices off the stack and color them in a color not used by neighbors. (reuse colors as far as possible).

Coalescing copies to reduce degree:

The compiler can use the interference graph to coalesce two live ranges. So by coalescing, what type of benefits can you get?



(a) Before Coalescing

(b) After Coalescing LR_a and LR_b

Figure – Coalescing Live Ranges

Comparing Top-Down and Bottom-Up allocator:

- Top-down allocator could adopt the 'spill and iterate' philosophy used in bottom-up ones.
- 'Spill and iterate' trades additional compile time for an allocation that potentially, uses less spill code.
- Top-Down uses priority ranking to order all the constrained nodes. (However, it colors the unconstrained nodes in arbitrary order)
- Bottom-up constructs an order in which most nodes are colored in a graph where they are unconstrained.

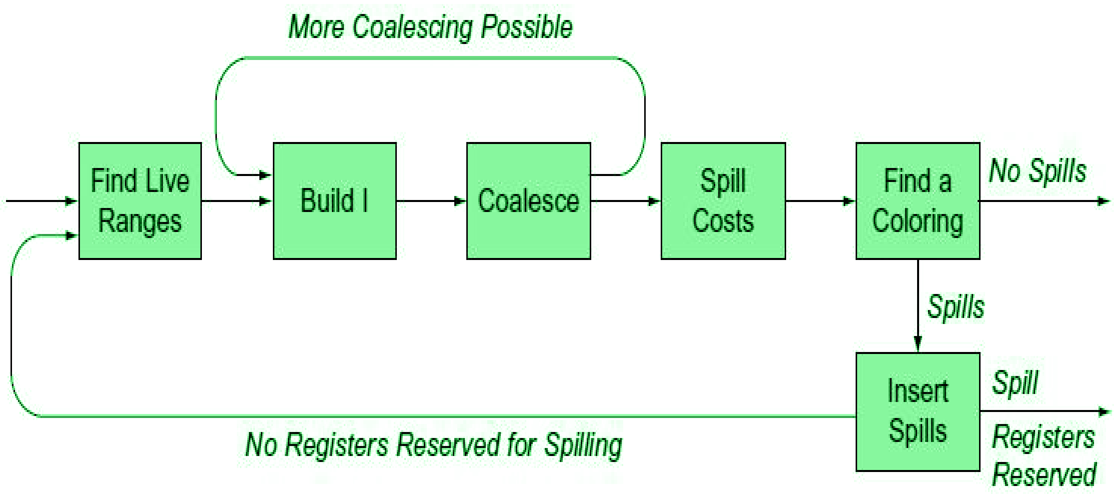


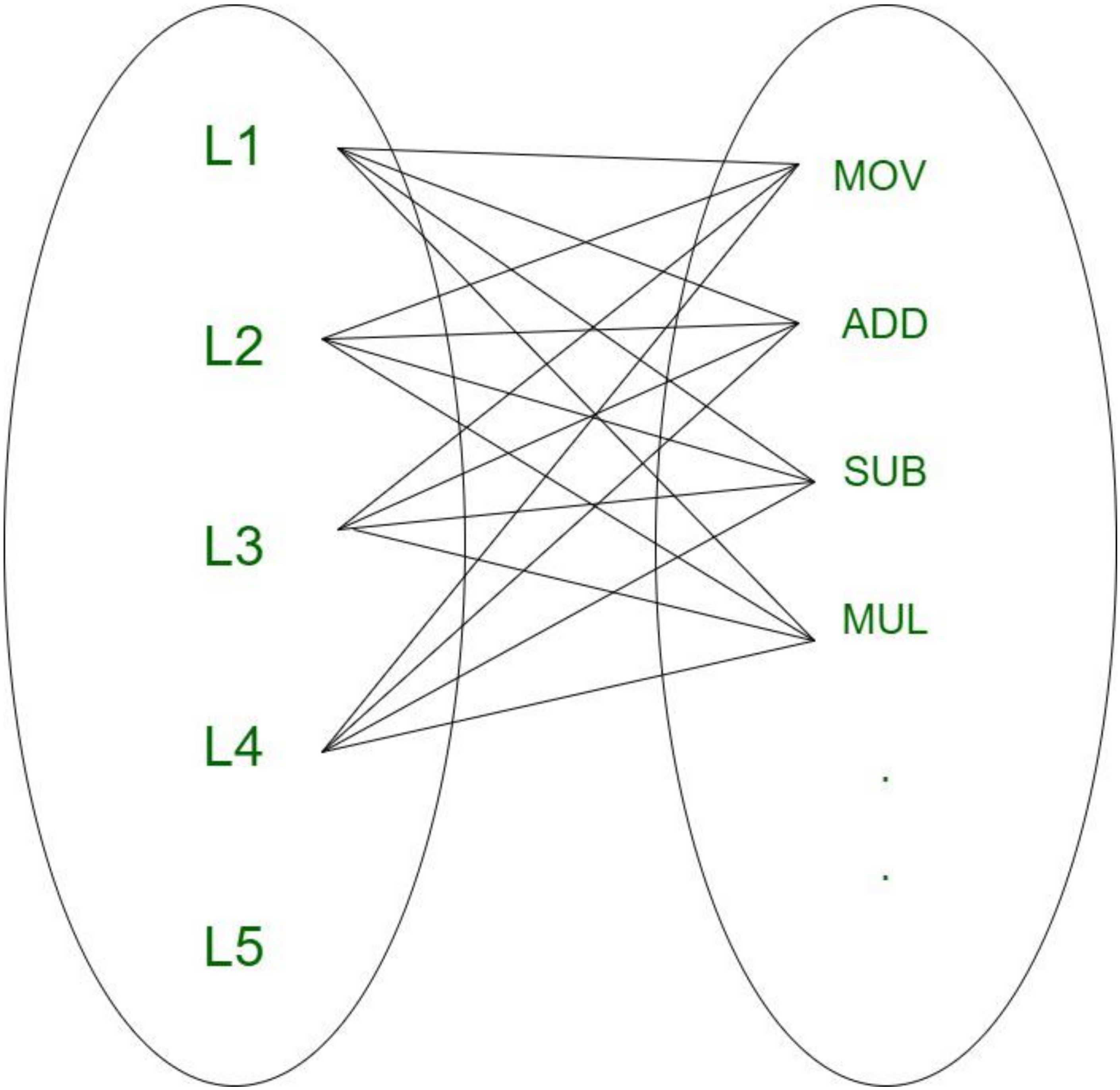
Figure – Coalescing Live Ranges

Target code generation:

Target code generation is the final Phase of Compiler.

1. **Input :** Optimized Intermediate Representation.
2. **Output :** Target Code.
3. **Task Performed :** Register allocation methods and optimization, assembly level code.
4. **Method :** Three popular strategies for register allocation and optimization.
5. **Implementation :** Algorithms.

Target code generation deals with assembly language to convert optimized code into machine understandable format. Target code can be machine readable code or assembly code. Each line in optimized code may map to one or more lines in machine (or) assembly code, hence there is a 1:N mapping associated with them .



1 : N Mapping

Computations are generally assumed to be performed on high speed memory locations, known as registers. Performing various operations on registers is efficient as registers are faster than cache memory. This feature is effectively used by compilers, However registers are not available in large amount and they are costly. Therefore we should try to use minimum number of registers to incur overall low cost.

Optimized code :

```
Example 1 :

L1: a = b + c * d


optimization :

t0 = c * d

a  = b + t0

Example 2 :

L2: e = f - g / d


optimization :

t0 = g / d

e  = f - t0
```

Register Allocation :

Register allocation is the process of assigning program variables to registers and reducing the number of swaps in and out of the registers. Movement of variables across memory is time consuming and this is the main reason why registers are used as they available within the memory and they are the fastest accessible storage location.

```
Example 1:

R1<--- a

R2<--- b

R3<--- c

R4<--- d


MOV R3, c

MOV R4, d

MUL R3, R4

MOV R2, b

ADD R2, R3
```

```
MOV R1, R2
MOV a, R1
Example 2:
R1<--- e
R2<--- f
R3<--- g
R4<--- h

MOV R3, g
MOV R4, h
DIV R3, R4
MOV R2, f
SUB R2, R3
MOV R1, R2
MOV e, R1
```

Advantages :

- Fast accessible storage
- Allows computations to be performed on them
- Deterministic as it incurs no miss
- Reduce memory traffic
- Reduces overall computation time

Disadvantages :

- Registers are generally available in small amount (up to few hundred Kb)
- Register sizes are fixed and it varies from one processor to another
- Registers are complicated
- Need to save and restore changes during context switch and procedure calls

Type systems:

Type Systems

SEMANTIC CHECKING. A compiler must perform many semantic checks on a source program.

- Many of these are straightforward and are usually done in conjunction with syntax-analysis.
- Checks done during compilation are called *static checks* to distinguish from the *dynamic checks* done during the execution of the target program.
- Static checks include:

  Type checks.
  The compiler checks that names and values are used in accordance with type rules of the language.

  Type conversions.

  Detection of implicit type conversions.

  Dereferencing checks.

  The compiler checks that dereferencing is applied only to a pointer.

  Indexing checks.

  The compiler checks that indexing is applied only to an array.

  Function call checks.

  The compiler checks that a function (or procedure) is applied to the correct number and type of arguments.

  Uniqueness checks.

  In many cases an identifier must be declared exactly once.

  Flow-of-control checks.

  The compiler checks that if a statement causes the flow of control to leave a construct, then there is a place to transfer this flow. For instance when using **break** in C.

In this chapter we will focus on type checking and type conversions. Most of the other static checks are easy to implement.

TYPE INFORMATION may be needed for the code generation. For instance, for the translation of *overloaded symbols* like +. Indeed, overloading may be accompanied by coercion of types, where a compiler supplies an operator to convert an operand into the type expected by the context.

OVERLOADED SYMBOLS AND POLYMORPHIC FUNCTIONS. An *overloaded symbol* is a symbol that represent different operations in different contexts. This is different from a *polymorphic function* which is a function whose body can be executed with arguments of several types.

TYPES HAVE STRUCTURE. Generally types are either basic or structured.

- Basic types are the atomic types with no internal structure. For instance, in C: `char`, `int`, `double`, `float`, etc.
- Structured types are the types of constructs like arrays, sets, (typed) pointers, structs, classes, and functions.

TYPE EQUIVALENCE. Because of overloaded symbols and structured types, we will be concerned with the non trivial problem of type equivalence.

- That is, given two language constructs, decide whether they have the same type.
- We will use the notion of a *type expression*.

TYPE EXPRESSION. The idea is to associate each language construct with an expression describing its type and so called *type expression.* Type expressions are defined inductively from basic types and constants using *type constructors.* Here are the type constructors that we shall consider in the remaining of this chapter. They are close to type constructors of languages like C or PASCAL.

Basic types.

`char`, `int`, `double`, `float` are type expressions.

Type names.

It is convenient to consider that names of types are type expressions.

Arrays.

If T is a type expression and n is an `int` then

$$\texttt{array}(n, T)$$

is a type expression denoting the type of an array with elements in T and indicies in the range $0 \cdots n - 1$.

Products.

If T_1 and T_2 are type expressions then

$$T_1 \times T_2$$

is a type expression denoting the type of an element of the Cartesian product of T_1 and T_2. We assume that products of more than two types are defined in a similar way and that this product-operation is left-associative. Hence

$$(T_1 \times T_2) \times T_3 \text{ and } T_1 \times T_2 \times T_3$$

are equivalent.

Records.

If $name_1$ and $name_2$ are two type names, if T_1 and T_2 are type expressions then

$$\texttt{record}(name_1\text{: } T_1,\ name_2\text{: } T_2)$$

is a type expression denoting the type of an element of the Cartesian product of T_1 and T_2. The only difference between a record and a product is that the fields of a record have names.

Pointers.

If T is a type expression, then

$$\texttt{pointer}(T)$$

is a type expression denoting the type *pointer to an object of type T*.

Function types.

If T_1 and T_2 are type expressions then

$$T_1 \longrightarrow T_2$$

is a type expression denoting the type of the functions associating an element from T_1 with an element from T_2.

<u>GRAPHICAL REPRESENTATIONS FOR TYPE EXPRESSIONS.</u> Let Σ_{val} be the alphabet consisting of

- the basic types,
- the possible values of an array index, say $\mathbb{N}$,
- the possible names for a type.

Let Σ_{op} be the alphabet consisting of the keywords `array`, ×, `record`, `pointer` and $\longrightarrow$. Then it is not hard to see that the set of type expressions can be viewed as a language of expressions in the sense of the section about syntax trees in the chapter dedicated to syntax-directed definitions. Indeed we could write

(`array` n T), (`record` $name_1$ T_1 $name_2$ T_2) and (`pointer` T)

instead of

`array`(n, T), `record`($name_1$: T_1, $name_2$: T_2) and `pointer(T)`.

That way we would satisfy the definition of a language of (arithmetic) expressions. It follows that we can use syntax trees and DAGs to represent type expressions graphically.

TYPE SYSTEMS. A *type system* is a set of rules assigning type expressions to different parts of the program.

- Type systems can (usually) be implemented in a syntax-directed way.
- The implementation of a type system is called a type checker.

A LANGUAGE IS STRONGLY TYPED if the compiler can guarantee that the accepted programs will **execute without type errors.** Lisp languages are usually weakly typed:

```
CannaLisp listener 3.5 beta 2
-> (defun f (l) (car l))
f
-> (f '(+ 1 2))
+
-> (f (+ 1 2))
Bad arg to car 3
```

Data abstraction:

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this −

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

int main() {
   cout << "Hello C++" <<endl;
   return 0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels −

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages −

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example −

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

class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }

    // interface to outside world
    void addNum(int number) {
      total += number;
    }

    // interface to outside world
    int getTotal() {
      return total;
    };

  private:
    // hidden data from outside world
    int total;
};

int main() {
  Adder a;

  a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

Designing Strategy

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

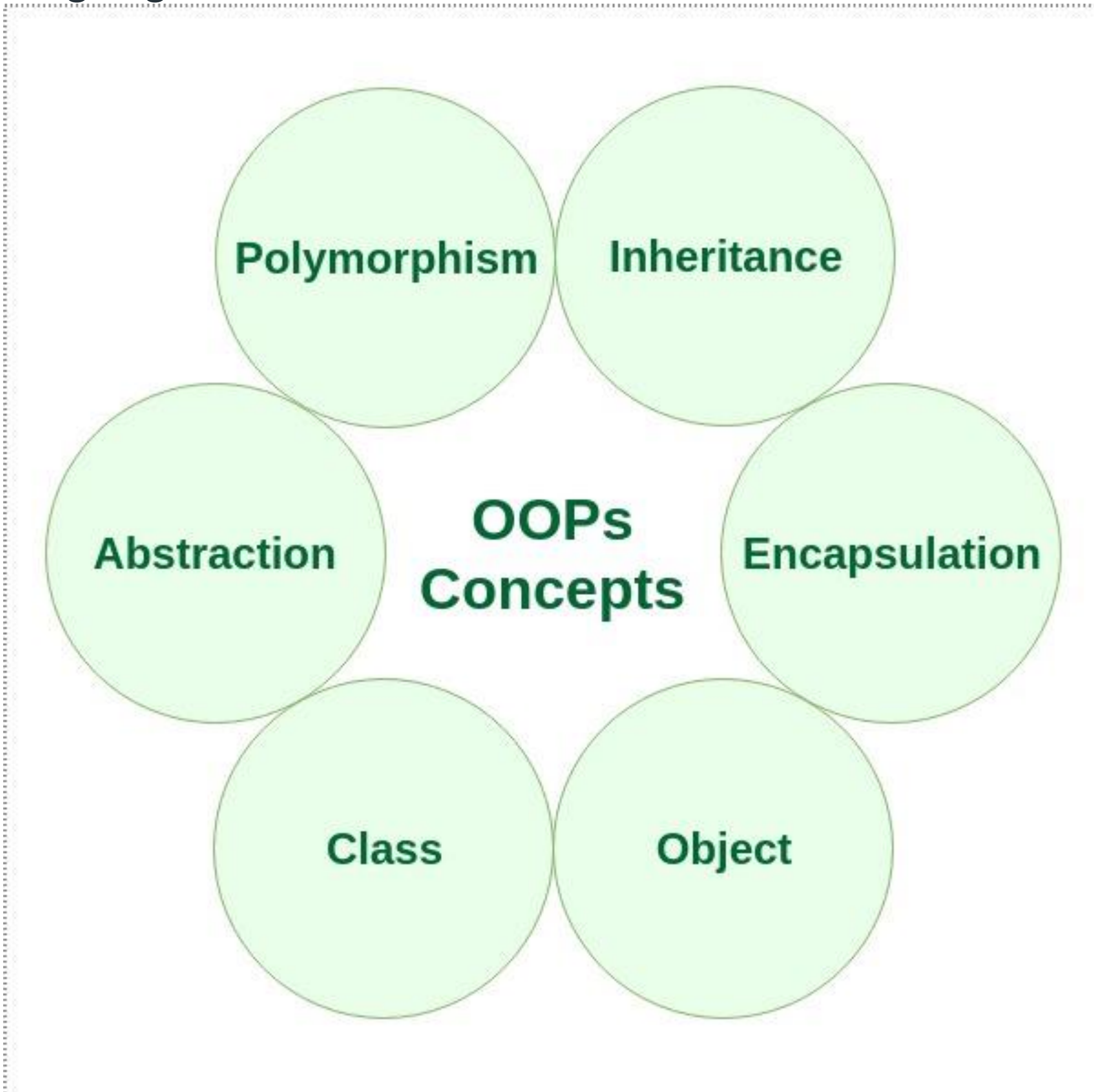
Compilation of Object Oriented features:

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

There are some basic concepts that act as the building blocks of OOPs i.e.

1. Class
2. Objects
3. Encapsulation
4. Abstraction
5. Polymorphism
6. Inheritance
7. Dynamic Binding
8. Message Passing

Characteristics of an Object-Oriented Programming Language



Class

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage, etc and member functions can apply brakes, increase speed, etc.

We can say that a **Class in C++** is a blueprint representing a group of objects which shares some common properties and behaviors.

Object

An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

- C++

```
// C++ Program to show the syntax/working of Objects as a

// part of Object Oriented PProgramming

#include <iostream>

using namespace std;



class person {

    char name[20];

    int id;



public:

```

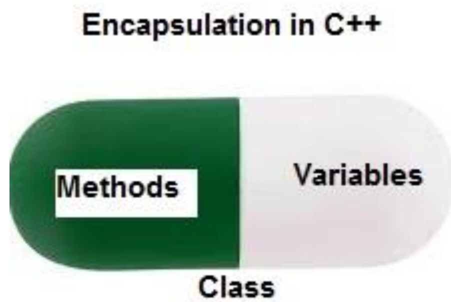
```
    void getdetails() {}

};



int main()

{



    person p1; // p1 is a object

    return 0;

}
```

Objects take up space in memory and have an associated address like a record in pascal or structure or union. When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other’s data or code, it is sufficient to know the type of message accepted and the type of response returned by the objects.

To know more about C++ Objects and Classes, refer to this article – [C++ Classes and Objects]

Encapsulation

In normal terms, Encapsulation is defined as wrapping up data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them. Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.



Encapsulation in C++

Encapsulation also leads to *data abstraction or data hiding*. Using encapsulation also hides the data. In the above example, the data of any of the sections like sales, finance, or accounts are hidden from any other section.
To know more about encapsulation, refer to this article – [Encapsulation in C++]

Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of an accelerator, brakes, etc. in the car. This is what abstraction is.

- ***Abstraction using Classes***: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- ***Abstraction in Header files***: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

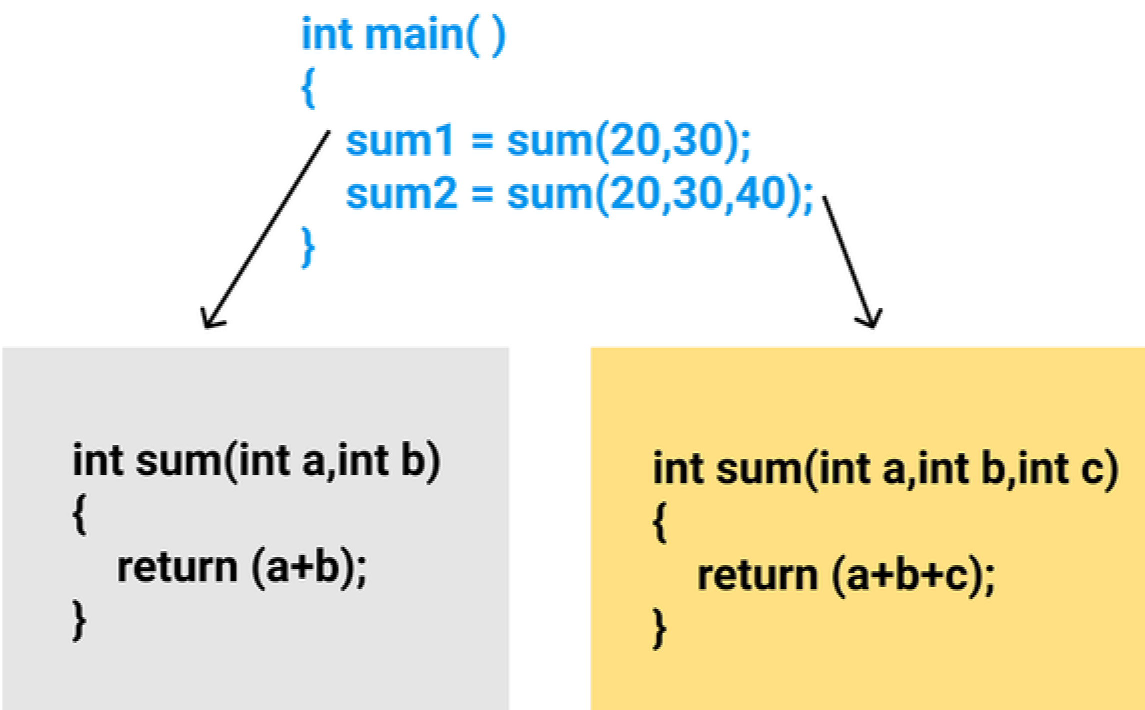
To know more about C++ abstraction, refer to this article – [Abstraction in C++]

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A person at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ supports operator overloading and function overloading.

- ***Operator Overloading***: The process of making an operator exhibit different behaviors in different instances is known as operator overloading.
- ***Function Overloading***: Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, sometimes there are 2 integers, and sometimes there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



Polymorphism in C++

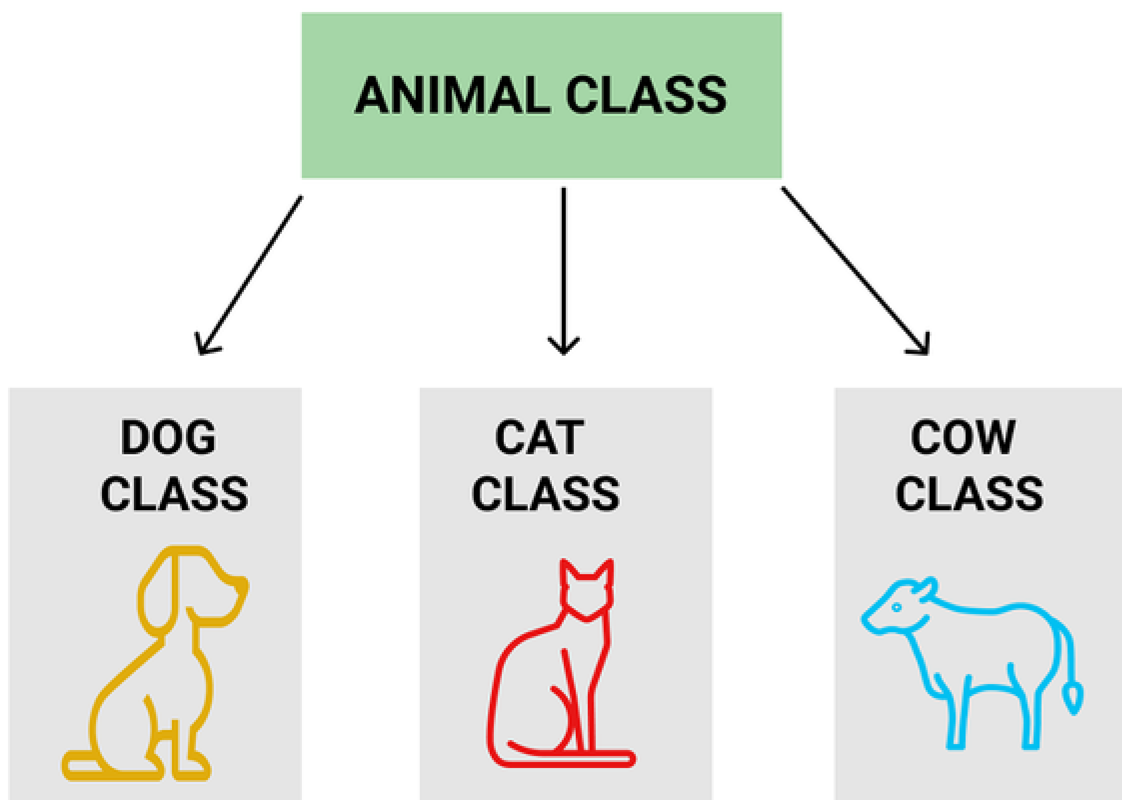
To know more about polymorphism, refer to this article – [Polymorphism in C++]

Inheritance

The capability of a class to derive properties and characteristics from another class is called [Inheritance](). Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**: The class whose properties are inherited by a sub-class is called Base Class or Superclass.
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Inheritance in C++

To know more about Inheritance, refer to this article – Inheritance in C++

Dynamic Binding

In dynamic binding, the code to be executed in response to the function call is decided at runtime. C++ has virtual functions to support this. Because dynamic binding is flexible, it avoids the drawbacks of static binding, which connected the function call and definition at build time.

Example:

- C++

```
// C++ Program to Demonstrate the Concept of Dynamic binding

// with the help of virtual function

#include <iostream>

using namespace std;



class GFG {

public:

    void call_Function() // function that call print

    {

        print();

    }

    void print() // the display function

    {

        cout << "Printing the Base class Content" << endl;

    }

};

class GFG2 : public GFG // GFG2 inherit a publicly

{
```

```
public:

    void print() // GFG2's display

    {

        cout << "Printing the Derived class Content"

             << endl;

    }

};

int main()

{

    GFG geeksforgeeks; // Creating GFG's pbject

    geeksforgeeks.call_Function(); // Calling call_Function

    GFG2 geeksforgeeks2; // creating GFG2 object

    geeksforgeeks2.call_Function(); // calling call_Function

                                    // for GFG2 object

    return 0;

}
```

Output

```
Printing the Base class Content

Printing the Base class Content
```

As we can see, the print() function of the parent class is called even from the derived class object. To resolve this we use virtual functions.

Message Passing

Objects communicate with one another by sending and receiving information. A message for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

Non-imperative programming languages:

DIFFERENCE BETWEEN IMPERATIVE AND NON-IMPERATIVE PROGRAMMING

The major difference between imperative and non-imperative programming is that the imperative language focus on how the program should follow its tasks, while the non-imperative, functional or logical language tells the program what it needs to do as opposed to how to do it.

Imperative programming in computer science “is a programming paradigm that describes computation in terms of statements that change a program state” (Libflow, nd.). Imperative programs define the sequence of commands for the computer to perform. This is in contrast to declarative which expresses what the program should accomplish without prescribing how to do it in terms of sequences of action to be taken.

The non-imperative programming language would be better in functional and logical programming. A pure functional language like Haskell has state changes that are only represented as functions that transform the state (Foss, 2015). As with functional programming language, logical programming consists of logical statements and programs executed by searching for proofs of the statement. Examples include Prolog and database query languages such as SQL.

On the other hand, procedural language is an imperative language in which the program is built from one or more procedures (subroutines or functions). Imperative programming can also be used in structured programming and modular programming.

WHY THE SELECTD LANGUAGE MAKES SENSE

Selected imperative programming language makes sense because the hardware implementation of almost all computers is imperative as nearly all computers is designed to execute machine code, which is native to the computer, and written in imperative style.

Assignment statements in imperative paradigm perform an operation on information located in memory and store the results in memory for later use. High level imperative language permits evaluation of complex expressions consisting of a combination of arithmetic operation and function evaluation.

Language processing (translation, style checking), pattern matching (vision, genetics), and process optimization are among examples (scheduling). Non Imperative programming languages include Ruby, R, Haskell, Scheme, and ML.