Chapter 1: Preliminaries

Programming domains:

- Scientific applications (Fortran, ALGOL 60)
- Business applications (COBOL)
- Artificial intelligence (LISP, Scheme, Prolog)
- Systems programming (PL/I, C)
- Scripting languages (ksh, tcl, awk, Perl, Javascript, PHP)
- Special-purpose languages (...)

Readability

Overall simplicity

- Languages with large number of basic components are more difficult to learn than languages with a small number. Problems occur when the program’s author learned a different subset than the subset with which the reader is familiar.
- Feature multiplicity. If there are too many ways to do the same thing, this can be confusing for the reader.
- Overloading. Programming can use this in a useful way but also non-sensible. E.g. using + to sum two arrays per element.

Orthogonality Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language.

An orthogonal language feature is independent of the context of its appearance in a program.

Control statements Poor readability can be caused by the inadequate control statements of some programming languages.

An example is goto which makes it hard for the reader to read the program in a top-to-bottom fashion.
**Data types and structures** The presence of adequate facilities for defining data types and data structures in a language is a significant aid to readability.

**Syntax considerations** The syntax of the elements of a language has a significant effect on the readability of programs. Some design choices:

**Identifier forms** Restricting identifiers to very short lengths detracts from readability (like Fortran 77 and (ANSI) BASIC did).

**Special words** Program readability is strongly influenced by the form of the language’s special words (e.g. blocks with \{ .. \} or \begin/end\)

**Form & meaning** Designing statements so that the appearance at least partially indicates their purpose helps readability. Semantics should follow directly from syntax. (e.g. multiple meaning of `static` in C is confusing).

**Writability**

**Simplicity and orthogonality** If the language has a large number of different constructs, most programmers won’t be familiar with every one of them. This can lead to misuse of some features and disuse of others that may be more elegant or efficient.

**Support for abstraction**

Definition Abstraction: The ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.

Abstraction is a key concept in contemporary programming language design. The degree of abstraction allowed by a language and the naturalness of its expression are very important to writability.

Examples of types of abstraction:

**Process abstraction** A subprogram which implements a sort algorithm. This code would have multiple occurrences in the program if abstraction wasn’t possible making it both less readable and writable.

**Data abstraction** A binary tree.

**Expressivity** This has different meanings.

It can mean that there are very powerful operators that allow a great deal of computation to be accomplished with a very small program (does affect readability though).

It can mean the language has a relatively convenient way of specifying computations (e.g. `count++` in C).
Reliability

Type checking Type checking is simply testing for type errors in a given program, either by the compiler or during program execution.

Java requires all variables to be type checked at compile time. K&R C didn’t do type checking on parameters on function calls, this was corrected in ANSI C.

Exception handling The ability of a program to intercept run-time errors (as well as other unusual conditions detected by the program), take corrective measures, and then continue. This is a great aid to reliability. Java and C++ have great exception handling, while C and Fortran have virtually none.

Aliasing Aliasing is having two or more distinct referencing methods or names, for the same memory cell.

In some languages aliasing is used to overcome deficiencies in the language’s data abstraction facilities, in other it is greatly restricted to increase their reliability.

Readability and writability Both influence reliability (indirectly).

Language design

There are three major influences on language design:

- Computer/target architecture (von Neumann, platform independent)
- Programming methodologies (top-down, data abstraction, OOP)
- Language category (imperative, functional, logical, OO)

Implementation methods

Programming languages can be implemented by any of the three general methods:

Compilation Translation of source to machine code/executable (running through linking and loading). Examples are C, C++, Pascal.

Pure interpretation Source interpretation. Examples: APL, SNOBOL, Perl, PHP

Hybrid implementation systems Translation to intermediate language which features easy interpretation. Examples: Perl, early Java.

Chapter 2: Evaluation of the Major Programming Languages

Fortran

Fortran was the first widely used high-level compiled language. At the time the first Fortran version was designed, computers were unreliable and much more expensive
than software. Fortran was build around those issues and also focused on scientific applications.

Fortran I featured:

- Three-way if statement
- Arithmetic expressions
- Iterative loop-statement
- No data-typing (everything float except variables I, J, K, L, M and N)

and was highly focused on the IBM 704 architecture/machine.

Fortran II brought many Fortran I bugfixes and separate compilation of subroutines.

Fortran IV became the most widely used programming language of its time. It added declaration of variables, a logical if-statement and passing subprograms as parameters to other subprograms.

Fortran IV was replaced by Fortran 77, adding character string handling, logical loop control statements and else-clause for the if-statement.

Fortran 90 is very different from Fortran 77 adding dynamically allocatable & dealloca-table arrays, array functions, recursivity, pointers and new control statements as case and exit. Also Fortran 90 is modular, similar to Ada.

Fortran changed forever the way computers are used!

**LISP**

Much used in the AI area. Uses lists or nested lists as data structures (e.g (A B C) or (A (B C) D (E (F G))).

LISP is the first to differ a lot from imperative languages, it is a functional program language. It uses a lambda-calculus syntax to express functionality.

LISP was used widely and had many dialects. This was resolved by creating COMMON LISP which is the standard now. Also Scheme is a widely used dialect.

Related languages are ML (MetaLanguage) from which came Miranda and Haskell.

**ALGOL 60**

ALGOL (ALGOritmic Language) 60 is an effort to design a universal language. Although partly based on the much used Fortran it was an attempt to design a elegant, machine-independent and formal-defined language.

ALGOL 60 features:

- Block structures.
- Passing parameters by name and by value.
- Recursive procedures (only LISP featured that until then)
- Stack-dynamic arrays were allowed.
It omitted input and output statements (and formatting), because they were thought to be too machine independent.

ALGOL 60 was a success in ways that most languages are direct or indirect descendants. It was the first language that was internationally designed. It was the first language to be machine independent and had a formal (BNF specified) syntax. It never was widely used (mostly in Europe) and in that way was wasn’t a successful language. This is also due to missing I/O statements.

COBOL

COBOL has no language descendants, mostly because there was no interest in the business sector.

COBOL has its origins in FLOW-MATIC and was made for people who couldn’t or were too naive to understand programming or computers. COBOL’s syntax looks like natural language (first in English, later on also available in French, German, etc.).

COBOL’s new concepts/features:

- DEFINE: high-level language construct for macros.
- Hierarchical data structures (later on included in virtually every language).
- Truly connotative names/identifiers.
- Strong data division (but weak procedural).

BASIC

BASIC was designed to make it easy for students to learn programming. Goals were: easy to learn, pleasant, friendly, free and private access and user time more important than computer time. The last goal was a revolutionary concept.

Old versions very small and not interactive, very limited. BASIC evolved into dialects like Quick Basic and extended versions like Visual BASIC .NET.

PL/I

PL/I was the first large-scale attempt to design a language that could be used for a broad spectrum of application areas.

It included what were then considered the best parts of ALGOL 60 (recursion and blocks structures), Fortran IV (separate compilation with communication through global data) and COBOL (data structures, I/O and report generation).

PL/I’s new features were:

- Programs were allowed to create concurrently executing subprograms.
- Detecting and handling 23 different types of exceptions.
- Procedures could be recursive, but this could be disabled for efficiency.
- Pointers were included as a data type!
- Cross sections of arrays could be referenced.

PL/I was a very difficult language with many complicated constructs. It has some success in both scientific and business application, but hasn’t widely spread.
Simula 76

Simula is an extension of ALGOL 60. The difference was that subprograms could be restarted for simulation processes. It introduced a data structures which could be packaged with routines that manipulated that data structure. This definition is only a template and with this Simula brought the first classes, which was much later recognised.

ALGOL 68

ALGOL 68 was another huge language design effort based on ALGOL 60 and featured a lot of new ideas that appear in subsequent languages. ALGOL 68 had a highly orthogonal structure. The most important new feature was that there were primitive types and structures which the user could combine into a large number of different structures. Also introduced was a type of arrays that will be named *implicit heap-dynamic* (array declarations without subscript bounds).

Pascal

Pascal is descendant language of ALGOL-W and had a large impact on the teaching of programming. It was designed to be more elegant, simpler and easier than the in that time popularised ALGOL 68.

C

C’s ancestors include CPL, BCPL and B. C’s adequate control statements and data-structuring facilities allow its use in many application areas. C is liked and disliked for its lack of complete type-checking.

Modula/Modula-2

Modula grew out of concurrency experiments and is based on Pascal. Differences between Pascal/Modula and Modula-2 are modules which provided support for abstract data types, procedures as types, low-level facilities for systems programming and coroutines. The module structure paved the way for OO, which was implemented in Oberon.

Prolog

Programming language meant for the use of a formal logic notation to communicate computational processes to a computer. Predicate calculus is the used notation. Prolog is a totally different language from imperative and functional languages. It features a data structure built of facts and rules and can be queried with goal statements. Prolog++ is a dialect of Prolog that supports OOP.
Ada

Ada was developed by the Department of Defence (DoD) to get rid of the hundreds of different programming languages used within the department (especially embedded systems). Ada development took several iterations to arrive at the Ada 80 standard.

Major features of Ada are:

- Packages provide means for encapsulating data objects, data type specifications and procedures.
- Extensive exception handling.
- Generic program units (e.g. polymorphic functions) that are adapted/generated compile-time.
- Concurrent execution and intertask communication and synchronising methods.

Ada95 was developed to focus on four areas of the language:

- Interfacing (graphical user interfaces)
- Support for OO
- Flexible libraries
- Better control mechanisms for shared data.

Key features of Ada 95 were components derived from parent types (inheritance) and dynamic binding of subprogram calls, making it very different from Ada 83.

Smalltalk

Smalltalk is a fully object-oriented language with three fundamental characteristics:

- Abstraction
- Inheritance
- Dynamic binding

In Smalltalk everything is an object. This concept is based on Simula’s object simulation by sending messsages to objects and getting response-messages.

C++

C++ evolved from C with Classes. It features two contracts that defines types: classes and structs. C++ also allows both functions and methods and also overloading of both. Dynamic binding is done by providing virtual methods. Also note that C++ is aimed to be backward compatible with C, leaving most of C’s un-safeness in the language.

Both methods and classes can be templated like in Ada. Exception handling is done different from that of Ada.
Java

Java is an attempt to clean-up the un-safeness and dual-structure of C++ (both function and methods). Things as enumerations, union and record types were “removed” in respect to C++ in favour of classes.

Java features both value types and objects and uses an intermediate language (byte-code) to create platform-independence. Java features extensive type-checking and range-checking of array bounds compile-time. “New” in Java is the garbage collector making deallocation of objects implicit.

C#

C# means to be an advance over both C++ and Java as general purpose language. Although bringing changes to some statements as switch, returning the enumeration, structs and pointers.

Also added were function pointers and delegates (type-safe and object-oriented method references).

C++ and Java have two typing systems. This creates the continuous need to convert values between the two systems. In C# this is solved by implicitly making the conversion between values of the two typing systems.

Chapter 3: Describing Syntax and Semantics

Backus-Naur Form (BNF) and context-free grammars: Equivalent meta-languages suited for the task of describing the syntax of programming languages. Not only descriptive, but also the associated parse trees that can be associated with their generative actions give graphical evidence of the underlying syntactic structures.

Attribute grammar: A descriptive formalism that can describe both the syntax and static semantics of a language (type-checks, subscripts). This is an extension of the context-free grammars.

Operational semantics: A method of describing the meaning of the language constructs in terms of their effects on an ideal machine (dynamic semantics).

Axiomatic semantics: A tool, based on formal logic, for proving the correctness of programs.

Denotational semantics: The usage of mathematical objects to represent the meanings of language constructs. Language entities are converted to these mathematical objects with recursive functions.

---

1This is just an implementation feature, not really a language feature.
Chapter 5: Names, Bindings, Type Checking and Scopes

Binding

**Definition** Binding: A binding is an association, such as between an attribute and an entity or between an operation and a symbol.

**Definition** Binding time: The time at which a binding takes place.

Possible binding times are (and examples):

- Language design time (choosing \* for multiply)
- Language implementation time (possible values of an **int**)
- Compile time (variable to type binding (strong typed language))
- Load time (binding a variable to a memory cell (constants, etc.))
- Link time (binding a library subprogram to the subprogram code)
- Run time (binding a variable to a memory cell (malloc, etc.))

**Definition** Static binding: A binding is static if it first occurs before runtime and remains unchanged throughout program execution.

**Definition** Dynamic binding: A binding is dynamic if it first occurs during runtime or can change in the course of program execution.

Type bindings

**Definition** Explicit variable declaration: An explicit declaration is a statement in a program that lists variable names and specifies that they are a particular type.

**Definition** Implicit variable declaration: An implicit declaration is a means of associating variables with types through default conventions instead of declaration statements.

**Definition** Dynamic type binding: With dynamic type binding a variable is bound to a type when it is assigned a value in an assignment statement.

The primary advantage of dynamic binding of variables to types is that it provides a great deal of programming flexibility. The two main disadvantages are:

- The error checking capabilities of the compiler are diminished compared to static type bindings.
- The cost of implementing dynamic attribute binding is considerable, particularly in execution time.

Dynamic binding is only possible in purely interpreted languages.
Storage Bindings and Lifetime

**Definition** Allocation: To take the memory cell to which a variable is to be bound from a pool of available memory.

**Definition** Deallocation: The process of placing a memory cell that has been unbound from a variable back into the pool of available memory.

**Definition** Lifetime (of a variable): The time during which a variable is bound to a specific memory location.

We can divide scalar (unstructured) variables into four categories, based on their lifetime:

1. Static
2. Stack-dynamic
3. Explicit heap dynamic
4. Implicit heap dynamic

**Definition** Static variables: Variables that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates.

Advantages:

- Static variables declared in subprograms are history sensitive, meaning that they retain their values during separate executions of the subprogram.
- They are efficient, because they can be addressed directly and have no runtime overhead for (de)allocation.

Disadvantages:

- Reduced flexibility, for example in languages that only have statically bound variables, recursion is not possible.
- Storage can not be shared amongst variables, two subprograms that use large arrays and are never active at the same time, cannot share the same storage for their arrays.

**Definition** Stack-dynamic variables: Variables whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound.

Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached (runtime).

Advantages:
• They allow recursion
• All subprograms share the memory space for their locals.

Disadvantages:

• Runtime overhead for (de)allocation.
• Slower access because of indirect addressing.
• Subprograms cannot be history sensitive.

Definition Explicit heap-dynamic variables: Variables that are nameless (abstract) memory cells that are (de)allocated by explicit run-time instructions that are specified by the programmer.

Definition Heap: A collection of storage cells whose organisation is highly disorganised because of the unpredictability of its use.

These variables can only be referred through pointers and reference variables.

Advantage:
They allow dynamic structures like linked list and trees that need to grow and/or shrink during execution.

Disadvantages:

• Using pointers and reference variables correctly is very difficult.
• Extra cost because of the reference to the variables, allocations and deallocations.

Definition Implicit heap-dynamic variables: Variables that are bound to heap storage only when they are assigned values.

In fact, all their attributes are bound every time they are assigned.

Advantage:
Highest amount of flexibility, they allow highly generic code to be written.

Disadvantages:

• The (possibly high) runtime overhead of maintaining all the dynamic attributes.
• The loss of some error detection in the compiler.

(Strong) Typing

Definition Strongly typed language: A language is strongly typed if type errors are always detected.
This requires that the types of all operands can be determined, either at compile time or at run time.

“Fortran 95” is not strongly typed, because the use of EQUIVALENCE between variables of different types allows a variable of one type to refer to a value of a different type.

“Ada” is nearly strongly typed, because it allows programmers to breach the type checking rules by specifically requesting that type checking be suspended for a particular type of conversion.

“C and C++” are not strongly typed because both include union types, which are not type checked.

“Java and C#” are strongly typed in the same sense as Ada.

Scope

Definition Scope (of a variable): The range of statements in which a variable is visible. □

A variable is visible in a statement if it can be referenced in that statement.

Definition Scope rules: These determine how references to variables declared outside the currently executing subprogram or block are associated with their declarations and thus their attributes. □

Definition Nonlocal variables: Those variables that are visible within a program or block but are not declared there. □

Static Scope

Definition Static scoping: When the scope of a variable can be statically determined, that is, prior to execution. □

There are two categories of static scoped languages:

1. Those in which subprograms can be nested, which creates nested scopes.
2. Those in which subprograms cannot be nested.

Ada and Javascript allow nested subprograms, but the C-bases languages do not.

Definition Block: A new static scope that can be defined in the midst of executable code. □

For problems with static scoping read 5.8.3 on page 214-217.
Dynamic Scope

**Definition** Dynamic scoping: Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus the scope can only be determined at run time.

Programming problems with dynamic scoping:

- During the time span beginning when a subprogram begins its execution and ending when its execution ends, the local variables of the subprogram are all visible to any other executing subprogram.
- It’s impossible to statically type check references to non-locals. This results from the inability to statically determine the declaration for a variable referenced as a non-local.
- Reduced readability, because the calling sequence of subprograms must be known to determine the meaning of references to non-local variables.
- Accesses to non-local variables in dynamic-scoped languages take far longer than with static scoping.

Advantage:
Parameters that are passed between subprograms are often variables that are defined in the caller. These parameters are not needed with dynamic scoping, because they are implicitly visible in the called subprogram.

Chapter 6: Data Types

**Definition** Data type: A collection of data objects and a set of predefined operations on those objects.

**Definition** Descriptor: The collection of attributes of a variable.

If the attributes are all static, descriptors are only required at compile time. For dynamic attributes, however, part or all of the descriptors must be maintained during execution.

Primitive Data Types

**Definition** Primitive data types: Data types that are not defined in terms of other types.

Numeric types

Numeric types:

- Integer
- Floating-point
- Decimal
**Definition** Decimal type: These types store a fixed number of decimal digits, with the decimal point at a fixed position in the value. (Implemented in COBOL and C#)

Advantage:
They are capable of precisely storing decimal values, at least those within a restricted range, which cannot be done in floating-point.

Disadvantages:
- The range of values is restricted because no exponents are allowed.
- Their representation in memory is wasteful.

**Character types**

**Definition** Character string type: A type in which the values consist of sequences of characters.

Design Issues:
- Should strings be simply a special kind of character array or a primitive type (with no array-style subscripting operations)?
- Should strings have static or dynamic length?

“C and C++”: Character array
“Java”: STRING class is a primitive type, StringBuffer class is like a character array.
“C#” : Java alike classes

**Definition** Substring reference: A reference to a substring of a given string.

**Definition** Static length string: A string with a static length, set when the string is created.

**Definition** Limited dynamic length strings: Strings which can have a varying length up to a declared and fixed maximum set by the variable’s definition. (C and C-style strings in C++)

**Definition** Dynamic length strings: Strings which can have a varying length without any maximum. (Javascript/Perl)

Ada95 supports all 3 types.
User Defined Ordinal Types

**Definition** Ordinal type: A type in which the range of possible values can be easily associated with the set of positive integers.

**Definition** Enumeration type: A type in which all of the possible values, which are named constants, are provided in the definition.

Design issues (all related to type checking):

1. Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
2. Are enumeration values coerced to integer?
3. Are any other types coerced to an enumeration type?

Problem with point 2:
There is little control over its range of legal operations or its range of values.

Problem with point 3:
If an int type is coerced to an enumeration type, an enumeration type variable could be assigned any integer value whether is represented an enumeration constant or not.

Enumerated types enhance readability and can enhance reliability if enumerations are not coerced to integers. This is because they can not be assigned a value outside of their range or have any arithmetic operation performed on them (adding a day to the week for example). Coercion to int happens in C for example, but not in Ada and C#.

**Definition** Subrange type: A contiguous\(^2\) subsequence of an ordinal type.

Subrange types enhance readability by making it clear to readers that variables of subtypes can story only certain ranges of values. Reliability is increased, because assigning a subrange variable that is outside the specified range is detected as an error. Odd: No contemporary language except Ada 95 has subrange types.

**Array types**

An array is an homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Design issues:

- What are types for legal subscripts (== indices)?
- Are subscripting expressions in element references range-checked? (No: C, C++, Perl, Fortran; Yes: Java, ML, C#, Ada)
- When are subscript ranges bound?

\(^2\)aaneengesloten
• When does array allocation take place?
• Are ragged or rectangular multi-dimensioned arrays allowed, or both?
• Can arrays be initialised when they have their storage allocated?
• What kinds of slices are allowed and why?

We can divide arrays into five categories, based on the binding to subscript value ranges and the binding to storage.

1. Static array
2. Fixed stack-dynamic array
3. Stack-dynamic array
4. Fixed heap dynamic array
5. Heap dynamic array

**Definition** Static array: Subscript ranges are statically bound and storage allocation is static.

Advantage:
Time efficiency.

**Definition** Fixed stack-dynamic array: Subscript ranges are statically bound and allocation is done at declaration elaboration time during execution.

Advantage over 1:
Space efficiency.

**Definition** Stack-dynamic array: Subscript ranges are dynamically bound and storage allocation is dynamic. After binding and allocation they remain fixed however during the lifetime of the variable.

Advantage over 1 and 2:
Flexibility, the size of the array need not be known until the array is bound to be used.

**Definition** Fixed heap dynamic array: Similar to Stack-dynamic array, but the storage is allocated on the heap and bindings are done when the program requests them (malloc, etc).

**Definition** Heap dynamic array: Subscript ranges are dynamically bound and storage allocation is dynamic and both can change any number of times during the array’s lifetime.

Advantage over all others:
Flexibility, arrays can grow and shrink during program execution as the need for storage changes.
**Definition** Rectangular array: A multidimensional array in which all of the rows have the same number of elements, all of the columns have the same number of elements, etc.

**Definition** Jagged array: A multidimensional array in which the rows/columns/etc. need not have the same number of elements.

**Definition** Associative array: An unordered collection of data elements that are indexed by an equal number of values called keys.

In an associative array the user-defined keys must be stored in the structure, unlike in non-associative arrays.

When to use which array type?

Associative: If searches of the elements are required.
Non-associative: If every element must be processed.

**Record Types**

**Definition** Record: A possibly heterogeneous aggregate of data elements in which the individual elements are identified by name.

Design issues:

- What is the syntactic form of references to fields?
- Are elliptical references allowed?

**Definition** Fully qualified reference: A reference to a record field in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference.

**Definition** Elliptical reference: In this type of reference the field is named, but any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous in the referencing environment.

**Union types**

**Definition** Union: A type that may store different type values at different times during program execution.

**Pointer and Reference Types**

**Definition** Pointer type: A type in which the variables have a range of values that consists of memory addresses and a special value nil.

Pointers can be used to access memory cells located on the heap.

Advantage:
They add writeability, by allowing the implementation of dynamic structures.

Design issues:
• What are the scope and lifetime of a pointer variable?
• What is the lifetime of a heap-dynamic variable?
• Are pointers restricted as to the type of value to which they can point?
• Are pointers used for dynamic storage management, indirect addressing, or both?
• Should the language support pointer types, reference types, or both?

Pointer problems:

• Dangling pointer, meaning a pointer containing the address of a deallocated heap-dynamic variable.
• Memory leakage, when allocated heap-dynamic memory can no longer be accessed by the user program.

**Pointers in Ada** Fixing the main problems:
Dangling pointers are mostly fixed by implicitly deallocating a heap-dynamic variable at the end of the scope of its pointer type, thus dramatically lessening the need for explicit deallocation. It’s not foolproof because Ada still has an explicit deallocator, though its use is discouraged.

Memory leakage is not fixed.

Special notes:
Pointers in Ada can only point at the heap.

**Pointers in C and C++** Fixing the main problems:
None

Special notes:
The fact that pointers in C and C++ allow pointer arithmetics make their pointers much more interesting than those in other programming languages.

Pointers in C and C++ can point at virtually any variable anywhere in memory.

**Pointers in Fortran 95** Fixing the main problems:
None, pointers can become dangling very easy, because the `DEALLOCATE` statement takes a pointer as an argument, deallocates the memory that’s pointed to, but makes no attempt to check whether other pointers are pointing to the same memory.

**Reference Types**

**Definition** Referency type: A reference type is a variable is a constant pointer that is always implicitly dereferenced.

Reference types are included in C++, C# and Java.
Evaluation of pointers and references:

- A known danger to every programmer
- Formal methods are not very satisfactory here
- Garbage collection doesn’t solve all problems
- Benefits:
  - Flexible data structures
  - Currently the only way to build efficient data structures
  - References often appear in the domain (“real world”), so modelling is easier; this is one of the claims to fame of OO development.

Chapter 7: Expressions and Assignment statements

Arithmetic expressions are derived from those in mathematics. They consist of operators (unary, binary or ternary), operands (the constants, variables or subexpressions that operators act on), parentheses and function calls. Their purpose is to specify an arithmetic computation. This is done by fetching the operands from memory and performing the operation on them.

Precedence and Associativity

**Definition** Operator precedence: The order in which operators are evaluated, with those that have highest precedence first. For most imperative languages (based on mathematics): exponentiation (highest), then multiplication and division, then addition and subtraction (lowest).

**Definition** Associativity rules: The rules that define, for consecutive occurrences of the same operator, whether the left- or the rightmost occurrence is evaluated first. Mostly (for imperative languages) left to right, except for exponentiation, which is right to left.

Example: \(a+b+c = (a+b)+c\), but \(a**b**c = a**(b**c)\). But in Ada exponentiation is nonassociative, so \(a**b**c\) is illegal without braces.

In APL, all operators have same precedence and associativity is right to left for all operators.

Precedence and associativity rules can always be overridden with the use of parentheses.
Operand Evaluation Order

If the expression $a + b$ is to be evaluated, does it matter whether $a$ or $b$ is evaluated first? Normally not, only when evaluation of $a$ or $b$ has side effects.

**Definition** Functional side effect: A side effect that occurs when a function changes either one of its parameters or a global variable.

Example: If $fun(a)$ changes the value of $a$, $a + fun(a)$ will have a different outcome when $a$ is evaluated first than when $fun(a)$ is evaluated first.

Two solutions for this problem:

1. Disallow side effects in functions (but difficult for compiler, and eliminates programmer flexibility)
2. State evaluation order in language definition (done by Java (left to right)) (but less possibilities for compiler optimization)

Operator Overloading

**Definition** Operator overloading: Using an operator for more than one purpose.

Possible ways:

For different types $float_1 + float_2$ is used for adding floats, $int_1 + int_2$ for adding integers. Problem: in $float_1 = int_1/int_2$, integer division is performed, while this is probably not what was meant.

With different arities In $-a$, the “$-$” is a unary operator, in $a - b$ it is binary. Problem: when accidentally forgetting the first operand, the expression is still legal.

For abstract data types In some languages, operators can be overloaded to perform some useful function. Problem: there is no guarantee that the user-defined function is a sensible one.

Type Conversions

**Definition** Narrowing conversion: A type conversion that converts a value to a type that cannot store even approximations of all of the values of the original type. Example: converting `double` to `float`. Problem: loss of precision.

**Definition** Widening conversion: The converse of a narrowing conversion (suchs as `float` to `double`).

**Definition** Mixed-mode expression: An expression that has operands of different types in it.

Operands in a mixed-mode expression are implicitly converted to the same type. Such a conversion is called a coercion. The operand of the narrower type is converted to the operand of the wider type, except in assignments (assigning a floating-point expression to an integer variable, for example) where the right-hand side is always converted to the type of the left-hand side.

Operands can also be explicitly converted to another type; this is called a cast.
Relational and Boolean Expressions

**Definition** Relational operator: An operator that compares the values of its two operands, with a Boolean result (except in a language with no Booleans, where 0 is used for false and 1 for true). □

Relational operators always have lower precedence than arithmetic operators.

Boolean expressions consist of Boolean variables and constants, relational expressions and Boolean operators like “and”, “or” and “not” (and sometimes “xor” and equivalence). In languages without Booleans, a zero numeral means false and a nonzero one means true.

Short-Circuit Evaluation

**Definition** Short-circuit evaluation: The evaluation of an expression in which the result is determined without evaluating all of the operands and/or operators. □

Example: in \((a \geq 0) \land (b < 10)\), the \((b < 10)\) part doesn’t have to be evaluated if \(a < 0\), because \(false \land x = false\) for all \(x\).

When using operators with side effects, short-circuit evaluation can give trouble: \((a > b) \land ((b++) \div 3)\) will change \(b\) when \(a \leq b\), but not when \(a > b\).

Ada provides both “normal” and short-circuit versions of “and” and “or” operators, which is the best design, because it is most flexible.

Assignment Statements

Different forms of assignments:

- **Simplest form** `<target_variable> <assignment_operator> <expression>`
  Evaluates `<expression>` and assigns the result to `<target_variable>`. In C and C++, this assignment can be used in an expression, where it evaluates to the right-hand side of the assignment.

- **Conditional target** In C-based languages:
  `<boolean_expression> ? <var1> : <var2> = <expression>`
  Assigns the result of `<expression>` to `<var1>` if `<boolean_expression>` evaluates to true, or else to `<var2>`.

- **Compound assignment** For example, \(a += 3\) as a shorthand for \(a = a + 3\).

- **Unary assignment** For example \(a++\) as a shorthand for \(a = a + 1\) in C-based languages. Both \(a++\) and \(++a\) increase \(a\) by one, but \(a++\) in an expression evaluates to \(a\), while \(++a\) evaluates to \(a + 1\).

In C and C++, \((x = y)\) is often accidentally written in expressions instead of \((x == y)\), which almost certainly gives the wrong result and is difficult to detect.
Chapter 8: Statement-level Control Structures

**Definition** Control structure: A statement that provides the capability of selecting among alternative control flow paths or causing the repeated execution of certain collections of statements.

In principle, the only control structure that is necessary is a selectable goto, but a language without goto (which is a better choice) can work with only an if statement and a while statement. To this, the for statement was added, because it is often more practical to use than the while statement.

Control statements with multiple entries (that do not always begin with the first statement of the code segment) provide a little bit more flexibility, but a lot less readability.

**Selection Statements**

**Definition** Selection statement: A means of choosing between two or more execution paths in a program.

Basic form of two-way selection:

```plaintext
if control_expression
    then clause
else clause
```

Problem if clauses do not have to be surrounded by braces or `begin-end` keywords or similar:

```plaintext
if (sum == 0)
    if (count == 0)
        result = 0;
else
    result = 1;
```

Does the else-clause belong to the first or second if? Solution: make explicit by inserting braces or similar.

Multiple selection can be done by multiple nested if-statements, or by a `switch/case` statement.

**Iterative Statements**

**Definition** Iterative statement: A statement that causes a statement or collection of statements to be executed zero, one or more times.

Iteration in functional languages is accomplished by recursion, in imperative languages by `while` and `for` statements.

Two classes of iterative statements:

- **counting** (for statement) A loop variable is initialised to an initial value, and each time the loop body is executed the variable is increased with the step size. The repetition ends when the loop variable reaches some terminal value.
logical (while or do statement) The statement body is executed as long as a certain
Boolean expression is true. Available in two flavors, pretest:
while (expression) { statements }
and posttest:
do { statements } while expression.

In most languages, there is a statement to break out of an iteration (break or
last, possibly with a label to specify which nesting level to break out of), and to
break the current execution of the body and to start the next one (continue).

Some languages (Perl, JavaScript, PHP and C#) also have iteration not based on
counting or a Boolean expression, but on iterating through a data structure (such
as a list).

Unconditional branching

Definition Unconditional branch statement: A statement that transfers execution control to a specified place in the program (i.e., a goto statement). □

Because the goto is so powerful and flexible, it is easy to overuse it and to make an
unreadable and unreliable mess of one’s program.

Chapter 9: Fundamentals of Subprograms

Subprograms provide process abstraction, hiding details of the subprogram from the
calling code. It allows the reuse of code by calling the subprogram from different
locations.

Subprogram characteristics

• single entry point (there are exceptions, but they are not covered here).
• calling unit is suspended during the execution of the called subprogram, so
there always is only one subprogram in execution.
• control returns to caller after termination.

Basic definitions

Subprogram definition Describes the protocol (interface) and the actions (body)
of the subprogram.

Subprogram call Request to execute the subprogram.

Active An active subprogram is a subprogram that is currently executing.

Parameter profile Number, order, types of formal parameters.
Subprogram header  First part of definition; indication, name, parameter profile.

Protocol  Parameter profile + return type. Also called interface.

Subprogram declaration/prototype  Defines protocol, but not body. Necessary when subprogram is used before compiler has seen the definition.

Parameters

Accessed data  Nonlocal variables, parameters.

Formal parameters  The parameters in the subprogram header.

Actual parameters  The parameters with which the calling code calls the subprogram. Actual parameters are mapped to formal parameters on subprogram call. In nearly all languages, this mapping occurs based on position. Such parameters are called positional parameters. Some other languages provide keyword parameters for this mapping, in which the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter.

Default value  In some languages, a formal parameter can have a default value. If there is no actual parameter for that formal parameter in a subprogram call, then the default value is used.

Procedures and functions

Procedure  Type of subprogram, collection of statements that define parameterized computations. Does not return values, resembles a statement.

Function  Type of subprogram, semantically modeled on mathematical functions. Returns a value, resembles an expression.

Side effects  Modification of nonlocal variables or parameters by a subprogram. Can cause problems if the caller of the subprogram does not expect this.

Design issues

- What parameter-passing method or methods are used?
- Are the types of the actual parameters checked against the types of the formal parameters?
- Are local variables statically or dynamically allocated?
- Can subprogram definitions appear in other subprogram definitions (subprogram nesting)?
• If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?

• Can subprograms be overloaded?

• Can subprograms be generic?

Local referencing environments

Local variables Variables defined inside a subprogram. These are called local variables, because the scope is usually limited to the subprogram in which they are defined.

Local variables can be stack-dynamic or static. Stack-dynamic variables are allocated at each invocation of the subprogram. Static variables are allocated only once.

Stack-dynamic variable properties

• Essential for recursive subprograms.

• Storage for variables can be shared with other (inactive) subprograms.

• Cost of (de)allocation on each subprogram invocation.

• Indirect access to variables, usually more expensive.

• Not history sensitive.

Static variable properties

• More efficient.

• History sensitivity.

• No support for recursion.

• No storage sharing.

Most languages default to stack-dynamic for local variables. In a number of languages, the user can choose which variable model he wants. In some languages, it is not defined which is used by default.

Semantic models of parameter passing

Formal parameters can receive information, transmit information, or do both. These semantic models are called \textit{in mode}, \textit{out mode}, and \textit{inout mode} respectively.

Implementation models of parameter passing

Pass-by-value In mode. The value of the actual parameter is used to initialize the formal parameter, which acts as a local variable. It is normally implemented as a copy. Disadvantage: storage required for formal parameter, cost of copying.
Pass-by-result  Out mode. No value transmitted to the subprogram. The formal parameter acts as a local variable. On termination of the subprogram, the value of the formal parameter is transmitted back to the actual parameter of the caller (which must be a variable). Disadvantage: like pass-by-value. Also risk of parameter collision. Another problem that can occur with pass-by-result is that the implementor may be able to choose between two different times to evaluate the addresses of the actual parameters: at the time of the call or at the time of the return. For example, suppose a subprogram has the parameter list[index]. If index is changed by the subprogram, either through global access or as a formal parameter, then the address of list[index] will change between the call and the return.

Pass-by-value-result  Inout mode. Sometimes also called pass-by-copy. Implements pass-by-value and pass-by-result in one. The disadvantages and design issues are the same as for pass-by-value and pass-by-result combined.

Pass-by-reference  Inout mode. The pass-by-reference method passes an access path (usually a pointer or a reference). Through the provided access path, the subprogram operates on the actual parameters directly, rather than on local copies. Advantages: efficient. No extra storage or copying required. Disadvantages: slower access to parameters (because of indirection), inadvertent modifications of actual parameters possible. Also, using pass-by-reference, aliases can be created, reducing readability and reliability.

Pass-by-name  Inout mode. Does not correspond to a single implementation model. When parameters are passed by name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. It resembles late binding. Because it is relatively very expensive, it is not used in any widely used language.

Parameter-passing methods of the major languages

- Fortran uses inout, without specifying whether pass-by-reference of pass-by-result-value is used.
- C uses pass-by-value. Pass-by-reference can be achieved by passing pointers. Passed pointers can be declared const to avoid modifications.
- C++ often uses reference parameters, which can also be declared const.
- Java uses pass-by-value. Objects can only be accessed through reference variables, they are in effect passed by reference. Scalars can not be passed by reference.
- Ada knows in, out and inout models.
- Fortran 95 is like Ada.
- C# uses pass-by-value. Pass-by-reference can be specified through keywords. C# also supports out-mode parameters.
- PHP is like C#.
- Perl is call-by-reference.
Type-checking parameters

It is widely accepted that the types of the actual parameters should be checked against the types of the formal parameters of the called function. Without such checking, hard-to-find bugs are easy to introduce. Type checking greatly enhances reliability.

Implementing parameter-passing methods

In most languages, parameters are passed through the run-time stack.

- Pass-by-value parameters have their values copied onto the stack, where they are used as the corresponding formal parameters.
- Pass-by-result parameters are placed onto the stack by the called subprogram. The calling subprogram may retrieve them from the stack.
- Pass-by-value-result parameters have their values copied onto the stack, and retrieved back by the calling subprogram.
- Pass-by-reference parameters have their addresses placed on the stack, so that the called subprogram can dereference them.

Multidimensional arrays as parameters

In C and C++, the dimensions of a passed multidimensional array are unknown. The solution is to also pass the dimensions of the array as parameters. Because multidimensional arrays in C and C++ are arrays of arrays, which are in effect one long array, the index can be computed as follows: array[num_cols * row + col].

In Java and C#, arrays are objects. They are all single-dimensioned, but the elements can be arrays. Each arrays inherits a named constant that is set to the length of the array.

Design considerations

Two important considerations are:

- Efficiency.
- The requirement of one-way or two-way data transfer.

Access by subprogram code to data outside the subprogram should be minimized. Inout mode parameter should only be used when two-way data transfer is necessary. However, sometimes it is considerably more efficient to pass by reference than to pass by value (for example, when passing a big array).

Examples of parameter passing

(omitted)
Parameters that are subprogram names

In C and C++, functions cannot be passed as parameters, but pointers to functions can. The type of a pointer to a function is the functions protocol.

In languages that allow nested subprograms, such as JavaScript, there is another issue related to subprogram names that are passed as parameters. The question is what referencing environment for executing the passed subprograms should be used. The three choices are:

- The environment of the call statement that enacts the passed subprogram (shallow binding)
- The environment of the definition of the passed subprogram (deep binding)
- The environment of the call statement that passed the subprogram as an actual parameter (ad hoc binding)

Overloaded subprograms

An overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing environment. Every version of an overloaded subprogram must have a unique protocol. C++, Java, Ada, and C# allow overloaded subprograms. Constructors are commonly overloaded. Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls.

Generic subprograms

A generic or polymorphic subprogram takes parameters of different types on different activations. Parametric polymorphism is provided by a subprogram that takes a generic parameter that is used in a type expression that describes the types of the parameters of the subprogram.

Generic subprograms in Ada

(omitted)

Generic functions in C++

Generic functions in C++ have the descriptive name of template functions. The definition of a template function has the general form

```cpp
template <template parameters>
    -- a function definition that may include the template parameters
```

C++ template functions produce no actual code. They must be instantiated for each type they are to be used with.
Generic subprograms in other languages

At this moment, neither Java nor C# supports generic methods.

Design issues for functions

- Are side effects allowed?
- What types of values can be returned?

Functional side effects

Function parameters should be in-mode, to prevent unexpected side-effects. This makes functions behave more like mathematical functions. Some languages enforce this to increase reliability.

Types of returned values

- C allows any type to be returned except arrays and functions (but allows pointers to both).
- C++ is like C but also allows classes to be returned.
- Java and C# don’t have functions. They have methods, which can behave similar. In both, methods can return any type or class.

User-defined overloaded operators

In Ada and C++, the user can overload operators, so operators perform new or different operations on the variables they are used on. For example, + could be overloaded to do string concatenation if used on two strings.

Operator overloading can be useful, but it can also significantly reduce readability, when overloaded operators behave different than the user would expect.

Coroutines

Rather than the master-slave relationship between a caller and a called subprogram that exists with conventional subprograms, caller and called coroutines are on a more equal basis. In fact, the coroutine control mechanism is often called the symmetric unit control model.

Coroutines (which must be history-aware) constantly transfer control to each other. When control is transferred to a coroutine, it resumes operation after the statement it used to transfer control to another coroutine. For example, coroutine A may transfer control to B, which transfers control to C. C, in turn may transfer control to A, which transfers it to B again, etc.

Coroutines resemble a set of tasks doing cooperative multitasking.
Chapter 11: Abstract Data Types and Encapsulation Constructs

Introduction to data abstraction

An abstract data type is an encapsulation that includes only the data representation of one specific data type and the subprograms that provide operations for that type. Through access control, unnecessary details of the type can be hidden from units outside of that encapsulation. An instance of an abstract data type is called an object.

For example, consider floating point variables. The language specifies means to creating them, and operations on them, such as addition and multiplication. However, the actual representation of the floating point variable in the memory of the computer is hidden from the user.

User-defined abstract data types

An abstract data type is a data type that satisfies the following two conditions:

- The declarations of the type and the operations on objects of the type, which provide the type’s interface, are contained in a single syntactic unit. The implementation of the type and its operations may be in the same unit, or may be given in a separate syntactic unit. Also, other program units are allowed to create variables of the defined type.

- The representation of objects of the type is hidden from the program units that use the type, so the only direct operations possible on those objects are those provided in the types definition.

Program units that use a specific abstract data type are called clients of that type.

The advantage to hiding the implementation details of an abstract data type from clients is that clients cannot manipulate objects directly, increasing their integrity. Also, clients cannot depend on the implementation, only on the specification.

Design issues for abstract data types

- A facility for defining abstract data types in a language must provide a syntactic unit that can encapsulate the type definition and subprogram definitions of the abstraction operations.

- Few, if any, built-in operations should be provided. Candidates for built-ins are assignment and comparison for (in)equality.

- Can the abstract data types be parameterized?

Abstract data types in Ada

(omitted)
Abstract data types in C++

C++ provides structs and classes. Structs are not discussed here. The data defined in a C++ class are called data members; the functions (methods) defined in a class are called member functions. All of the instances of a class share a single set of member functions, but each instance gets its own set of the class’ data members. Class instances can be either stack dynamic or heap dynamic. If stack dynamic, they are referenced directly with value variables. If heap dynamic, they are referenced through pointers.

A C++ class can contain both hidden and visible (to clients of the class) entities. Hidden entities are placed in a private clause, and visible or public entities are written in a public clause. The public clause therefore describes the interface to class objects.

C++ allows the user to include the constructor and destructor functions in class definitions. The constructor is implicitly called on object creation, and may initialize data members. The destructor is called implicitly on object end-of-life, and may deinitialize data members. The constructor function has the same name as the class whose objects it initialises. There may be more than one constructor, which enables overloading. The destructor has the same name as the constructor, but preceded by a tilde (~).

In C++, objects are instantiated by the operator new, and destroyed by the operator delete. Neither constructors nor destructors have return types. Both can be called explicitly.

Abstract data types in Java

Java support for abstract data types is similar to that of C++. The key differences are:

- Java does not include structs, only classes.
- All objects are allocated from the heap and accessed through references.
- C++ abstract data types can be generic, Java’s cannot (yet).
- In Java, both the declaration and definition are placed in a single syntactic unit. Definitions are hidden from clients by making them private.
- C++ has private and public clauses. Java has private and public modifiers which can be attached to each method and variable definition.
- Java has garbage collection, obviating the need of most destructors.

Abstract data types in C#

C#’s abstract data type support closely resembles that of Java.
Parameterized abstract data types

It is often convenient to parameterize abstract data types. For example, for a class that implements a stack, it would be desirable to specify the allowed number of elements on the stack at the instantiation.

C++ supports parameterized, or generic, abstract data types. A constructor for the class `stack` that implements a stack could be the following:

```c++
stack(int size) {
    stkPtr = new int [size];
    maxLen = size - 1;
    top = -1;
}
```

A declaration for a stack object would then look like:

```c++
stack stk(150);
```

Parameterization in other languages  
C# and Java do not provide support for parameterized abstract data types. Ada does, but it is not discussed here.

Encapsulation constructs

An encapsulation is a collection of logically related code and data. Encapsulations are often placed in libraries and made available for reuse in other programs.

Nested subprograms  
Nested subprograms can be used for encapsulation by defining them in the logically larger subprograms that use them. This is far from ideal.

Encapsulation in C

C does not provide strong support for abstract data types. Encapsulation is achieved by placing the implementation of subprograms and data in a file, and placing the interface to these subprograms and data files in a header file. This is not completely safe, but adequate if used properly.

Encapsulation in C++

Encapsulation in C++ is similar to that in C.

Encapsulation in other languages

Ada and C# provide packages and assemblies respectively. They are similar in spirit to the other encapsulation methods, but differ in detail.
Naming encapsulations

C++, C#, Java and Ada include naming encapsulation. For Java, they are named packages, for C++ and C#, they are namespaces. In Ada, packages can be used as naming encapsulations.

Chapter 12: Support for Object-Oriented Programming

A Object Oriented language must have three key language features:

- Abstract data types (Called classes or objects in OO)
- Inheritance
- Dynamic binding of method calls to methods

In OO languages operations/functions on classes are called methods and method calls are called messages. The entire collection of messages is called message protocol or message interface.

Inheritance

**Definition** Inheritance: New classes can inherit the data and functionality from an existing class.

**Definition** Derived class or subclass: A class that is defined through inheritance from another class.

**Definition** Parent class or superclass: A class from which the new class is derived.

A new class can derive from a single parent (called single inheritance) or from multiple (called multiple inheritance). Java and smalltalk only allow single inheritance, C++ allows multiple inheritance.

Multiple inheritance is more complex. For example two parent classes could define the same method!

**Definition** Override: A subclass defines a method with the same name (and often the same protocol) as one of it’s superclasses. The method is then called an overridden method.

Classes can have two kinds of methods and variables:

- Instance methods/variables: These belong to one instance of the class and only describe/act on that instance
- Class methods/variables: These belong to the whole class rather then one instance, so there is only one copy of them!

33
**Dynamic binding**

This is needed to be able to allow a variables of a parent class to reference an object of the subclass (These kind of variables are called polymorphic). The parent class can define methods that are overriden by its subclasses. When such a method is called on a parent class, the calls is dynamically bound to the method in the proper class.

In Java and Smalltalk all methods are dynamically bound, in C++ all methods are statically bound be default. Methods need to be defined virtual to be dynamically bound. Static binding is obviously a lot faster then dynamic.

---

**Definition** Abstract method: A method which has a protocol definition, but no implementation.  

**Definition** Abstract class: A class with at least one abstract method.

---

Although Java only allows single inheritance. One can define a kind of abstract class, called an *interface*, with only abstract methods. A Java class can inherit from multiple interfaces!

**Definition** Subtype: A derived class is a subtype, when a variable of a subclass can appear anywhere a variable of the parent class type was legal, without causing a type error.

---

**Smalltalk**

Smalltalk was the first full Object Oriented language, everything in smalltalk is an Object (even the constant int 2). Smalltalk never became very popular, because it was focussed on language elegancy instead of practical use.

---

**Eiffel**

Design by contract. Assertions. Exploits the open/close principle: objects should be closed for users in the sense that the implementation is hidden. They should be open for inheritors that extend the behaviour. Other languages follow these principles, although often implementation details can be hidden for inheritors also.

---

**Java**

Why it is revolutionary: *Driven by security*, a basically succesful attempt has been made to shield the programmer completely from the underlying hardware and software platform. Consequences:

- quite complete and precise language definition, especially for an industrial language that is not just a research toy
- few issues in the language definition that are left to the implementor (which could boil down to “left to the platform”)

---

34
• “dangerous” language constructs are removed or as much as possible “disarmed” (pointers, array subscriptions, procedure passing, )

• independence of software platform implies the development of a complete and relatively abstract standard library, which is more and more considered as part of the language.

Chapter 14: Exception Handling

Definition Exception: An unusual event, erroneous or not, that is detectable either by software or hardware and that may require special processing. □

The special processing is called exception handling, which is done by a piece of code called an exception handler. An exception is raised when its associated event occurs (in C-based languages this is called thrown instead of raised)

An exceptions can have both user defined and predefined exceptions. Predefined exceptions are always implicitly raised, whereas user-defined exceptions are explicitly raised by user code.

Exception handling has the following advantages:

• Without exception handling the code required to detect and handle error conditions can clutter the code.

• With exception propagation errors can be handled dynamically in an other programming unit.

• A language with exceptions handling encourages its users to consider all events that could occur during program execution and how they can be handled

• Programs dealing with nonerroneous, but unusual situation can be simplified.

C++

there are no predefined exceptions in C++. All types can be thrown as an exception. C++ uses the following construct to catch exceptions:

```cpp
try {
    <statements>
} catch (formal parameter) {
    exception handler
}
```

The catch will catch any exception that matches it’s formal type (i.e. same type or one of it’s subclasses). An ellipsis formal parameter will match any throw. Only the first catch with a matching type will be executed.

If the exception is not caught in the try block, it will propagate to the caller of the function. When no handler is found, the program is terminated.

C++ exceptions can not be disabled!. A throw without args in the exception handler reraises the exception. A C++ function may list all exceptions that it can throw but not handle in a throw clause. When there is an throw clause, propagating an exception not in the clause is a fatal error!
Java

Java has predefined exceptions which can be thrown implicitly by the JVM. Java exception handling is based on C++ so only the differences are noted here.

All exceptions in Java subclasses of Throwable. There are two predefined subclasses of Throwable:

- Error, which is only thrown by the JVM. A user program should not handle these!
- Exception, which has two predefined subtypes.
  - IOException, which is thrown on IO errors
  - RuntimeException which is thrown in most cases by the JVM when a user program causes an error.

User programs can define there own exception classes, which by convention are a subclass of Exception.

Java uses the same construct as C++ to catch exceptions, but instead of an ellipsis as wildcard a catch for an Exception object can be used.

Error and RuntimeException are called unchecked exceptions. All others are checked exceptions. All methods that can have a checked exceptions must either handle it or list it in their throw clause. The Java compiler checks this!

An method can’t declare more exceptions then the method it overrides, though it can declare less. Exceptions can not be disabled!

Java has a finally clause wich can be added after the last catch. This is always executed!

Huizings remarks:

- Exception handling in Java. Interesting issues:
  - hierarchy in exceptions
  - visible in API and enforced to comply with API, as a consequence of Java’s nature
  - connection with OO: objects are relatively autonomous and hence exceptional situations can not be predicted by the caller

Chapter 16: Logic Programming

Programming using a form of symbolic logic as a programming language is called logic programming, languages based on this are logic programming languages or declarative languages.

Logic programming use propositions which are logical statemens which may or may not be true. The simplest proposistins, which are called atomic propositions, consist out of compound terms.

**Definition** compound term: One element of a mathematical relationship. It consists out of two parts, a functor and an ordered lists of parameters. for example: parent(bob, jake) in which parent is the functor and bob and jake parameters.
The semantic of logic programming languages is called *declarative semantics*. This is because in logic programming, programs exist out of declarations rather than assignments and control flow statements. The program declares what the result should look like, not how it is to be computed.

All propositions can be expressed in causal form, which has the following general syntax: \( B_1 \cup B_1 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m \)

**Definition** resolution: this is an inference rule that allows inferred propositions to be computed from given propositions. Resolution is devised to be applied to propositions in clausal form. The concept is as follows: when \( P_1 \subset P_2 \) and \( Q_1 \subset Q_2 \) then if \( Q_2 \) is identical to \( P_1 \) the two propositions can be rewritten as \( Q_1 \subset P_2 \)

When the propositions contain variables the resolution process must determine useful variables for them, this is called *unification*. Temporary assignment of values to variables for unification is called *instantiation*

**Definition** Horn clauses: A subset of clausal form which has at most one single atomic proposition on the left side. The left side of a clausal form is sometimes called the head. Horn clauses with left sides are called *headed*, without are called *headless*. Most, but not all, propositions can be stated in this form

**Prolog**

Prolog is the most widely used logic programming language. Prolog has two basic statements forms; these correspond to the headless and headed Horn clauses.

Headless horn clauses is interpreted as an unconditional assertion, or fact.

Headed horn clauses are called rules in Prolog and have the following form:

\[
\text{consequence} :- \text{antecedent}
\]

The consequence can be concluded if the antecedent is true. The antecedent can be a conjunction of terms, which are separated by a , (comma).

A headless horn clause can be used as a goal or query. On which Prolog will respond either true or false. Prolog tries to instantiate variables in a goal in a way that results in a true value for the goal.

**Definition** bottom-up resolution or forward chaining: All the fact and rules are used in an attempt to find a sequence that lead to the given goal.

**Definition** top-down resolution or backward chaining: The system begins with the given goal and attempts to find a sequence that matches a set of facts in the original database

Forward chaining is better when the number of possibly correct answers is large; in this case backward chaining would require a very large number of matches to get an answer. Prolog uses backward chaining.

A solution for goals that consist of multiple subgoals can be found by a *depth-first* search (all subgoals one by one) or a *breadth-first* search (all subgoals in parallel). Prolog uses depth-first, which uses less computer resources.
**Definition** backtracking: When, during resolution, multiple subgoals are being processed and the system fails on one subgoal. The system tries to find an alternative solution for a previous subgoal (after which the failed subgoal will be retried).

Prolog uses the same list notation as Haskell (and thus Hugs). For example: [apple, prune, grape, kumquat]. [X|Y] denotes a list with head X and tail Y.

Prolog works on the basis of the *Closed-world assumption* (innocent until proven guilty). If there is insufficient information in its database to prove the goal, it is assumed to be false.

Prolog can and has been used in RDBMS systems, Expert systems and natural language processing.