Enumeration Types

• All possible values, which are named constants, are provided in the definition

• C# example
  ```csharp
  enum days {mon, tue, wed, thu, fri, sat, sun};
  ```

• Design issues
  o 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 checked?
  o Are enumeration values coerced to integer?
  o Any other type coerced to an enumeration type?
Evaluation of Enumerated Type

• Aid to **readability**, e.g., no need to code a color as a number

• Aid to **reliability**, e.g., compiler can check:
  
  o operations (don’t allow colors to be added)
  
  o No enumeration variable can be assigned a value outside its defined range

  o Ada, C#, and Java 5.0 provide better support for enumeration than C++ because **enumeration type variables** in these languages are **not coerced** into integer types
Subrange Types

• An ordered contiguous subsequence of an ordinal type
  o Example: 12..18 is a subrange of integer type

• Ada’s design

```ada
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekdays;
Day2 := Day1;  -- Assignment is legal unless Day1 = sat/sun
```
Subrange Evaluation

• Aid to readability
  o Make it clear to the readers that variables of subrange can store only certain range of values

• Reliability
  o Assigning a value to a subrange variable that is outside the specified range is detected as an error
Implementation of User-Defined Ordinal Types

- Enumeration types are implemented as integers
- Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables
Pointer and Reference Types
Pointer and Reference Types

• A pointer type variable has a range of values that consists of memory addresses and a special value, *nil*

• Provide the power of indirect addressing

• Provide a way to manage dynamic memory

• A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)
Design Issues of Pointers

• What are the scope and lifetime of a pointer variable?

• What is the lifetime of a heap-dynamic variable?
Design Issues of Pointers

- 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 types point to the data
  - Reference types explicitly store references to the data
Pointer Operations

• Two fundamental operations:
  o assignment
  o dereferencing

• Assignment is used to set a pointer variable’s value to some useful address
  o \texttt{a} = \&b;
Pointer Operations

• Dereferencing yields the value stored at the location represented by the pointer’s value
  o Dereferencing can be explicit or implicit
  o C++ uses an explicit operation via *

\[
j = *ptr
\]

sets \( j \) to the value located at \( ptr \)
Pointer Assignment

\[ \text{ptr} = \&j \]

\[ \text{ptr} = \text{(int*)malloc( sizeof(int))} \]
The assignment operation $j = *ptr$
Problems with Pointers

• **Dangling pointers** *(dangerous)*
  
  o A pointer points to a heap-dynamic variable that has been deallocated
  
  o Many different ways to cause a dangling pointer

```c
delete [] array1;
return mysort (array1); // array1 is a dangling pointer
```
Problems with Pointers

• Lost heap-dynamic variable
  o An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    1. Pointer $p_1$ is set to point to a newly created heap-dynamic variable
    2. Pointer $p_1$ is later set to point to another newly created heap-dynamic variable
    3. The process of losing heap-dynamic variables is called *memory leakage*
Pointer Problems

A 25

B .4

A 25

B .4

garbage

dangling pointer

free B

Copyright © 2009 Addison-Wesley. All rights reserved.
Pointers in Ada

• Some dangling pointers are **disallowed** because dynamic objects can be automatically deallocated at the end of pointer's type scope
• The lost heap-dynamic variable problem is not eliminated by Ada
Pointers in C and C++

• Extremely flexible but must be used with care
• Pointers can point at any variable regardless of when or where it was allocated
• Used for dynamic storage management and addressing
Pointers in C and C++

• Pointer arithmetic is possible
• Explicit dereferencing and address-of operators
• Domain type need not be fixed (\texttt{void *} )
  \texttt{void *} can point to any type and can be type checked (cannot be de-referenced)
float stuff[100];
float *p;
p = stuff;

*(p+5) is equivalent to stuff[5] and p[5]

*(p+i) is equivalent to stuff[i] and p[i]
Reference Types

- C/C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++’s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than addresses
- C# includes both the references of Java and the pointers of C++
Evaluation of Pointers

- Dangling pointers and dangling objects are problems when using heap management.
- Pointers are like `goto`'s -- they widen the range of cells that can be accessed by a variable.
- Pointers or references are necessary for dynamic data structures -- so we can't design a language without them.