Chapter 6

Data Types

Topics

- Introduction
- Primitive Data Types
- Character String Types
- Enumeration Types
- Array Types
- Associative Arrays
- Record Types
- Tuple Types
- List Types
- Pointer Types
- Reference Types
- Type Checking
- Type Equivalence

Introduction

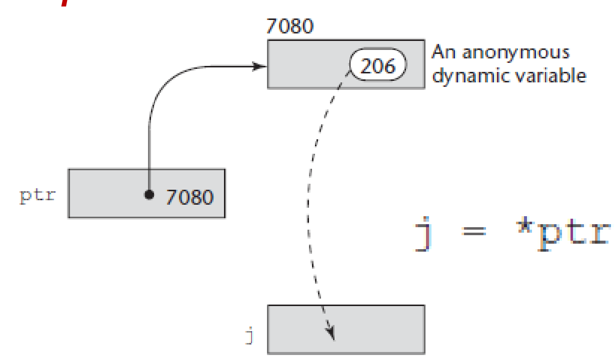
- A *data type* defines a *collection of data objects* and a *set of predefined operations* on those objects.
- A *descriptor* is the collection of the attributes of a variable.
- An *object* represents an *instance* of a user-defined (abstract data) type.
- One design issue for all data types:

  What *operations* are defined and

  how are they *specified*?

Pointer and Reference Types

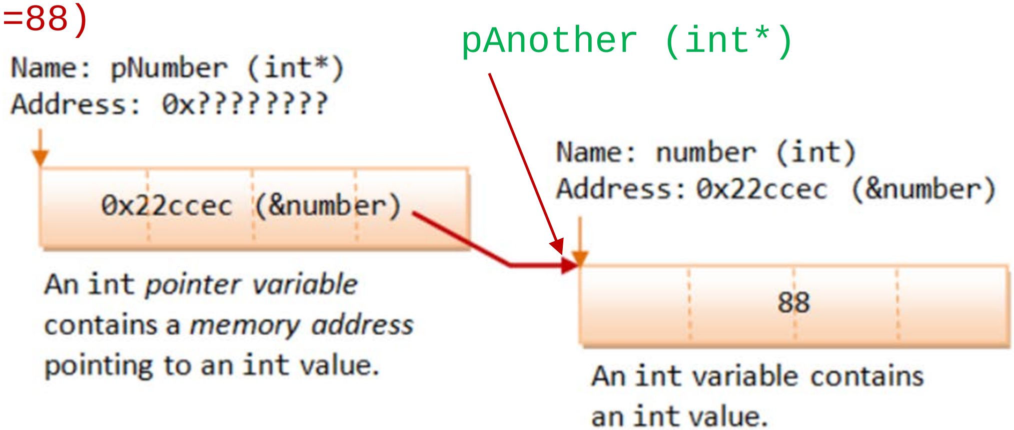
- cf) a Value type: a type of variable that stores data.
- A *pointer* type variable has a range of values that consists of *memory addresses* and a special value, *nil*.
  - Nil -- a pointer can't currently be used to reference a memory cell.
- Uses:
  - Power of *Indirect Addressing* – addressing flexibility
  - *Dynamic Memory Management*.
- Dynamic Memory Management:
  - A pointer can be used to access a location in the area where *storage is dynamically allocated,* called a *heap.*
  - *Heap dynamic variable*:
    - Dynamically allocated from the heap.
    - *anonymous variable* - no name.
    - only referenced by *pointer* or *reference type* variable without an identifier.



Pointer and Reference Types (cont.)

- Indirect addressing (= indirect reference):
  - A Pointers are used to *reference* other variables which are reference type.
  - The result of *dereferencing* the pointer.
  - E.g.) C++:

```
int number = 88;               // An int variable with a value
int *pNumber;                  // Declare a point variable pNumber  pointing to an int.
pNumber = &number;             // Assign the address of ‘number’ to pointer var ‘pNumber’.
int *pAnother = &number;  // Declare another int pointer and initiate to address
                                   of the variable ‘number’
j= *pAnother                    // Dereferencing of *pAnother  for the stored value
(i.e. j=88)
```



Pointer and Reference Types (cont.)

Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00	sum	int (4 bytes)	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFFF ($4.45015E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

A = 1010
B = 1011
C = 1100
D = 1101
E = 1110
F = 1111

```
int sum = 255;
int *ptrSum;
ptrSum = &sum;
```

Design Issues of Pointers

- What are the *scope* of and *lifetime* of a pointer variable?
- What is the *lifetime* of a heap-dynamic variable?
- Are pointers restricted to pointing at a particular type?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

Pointer Operations

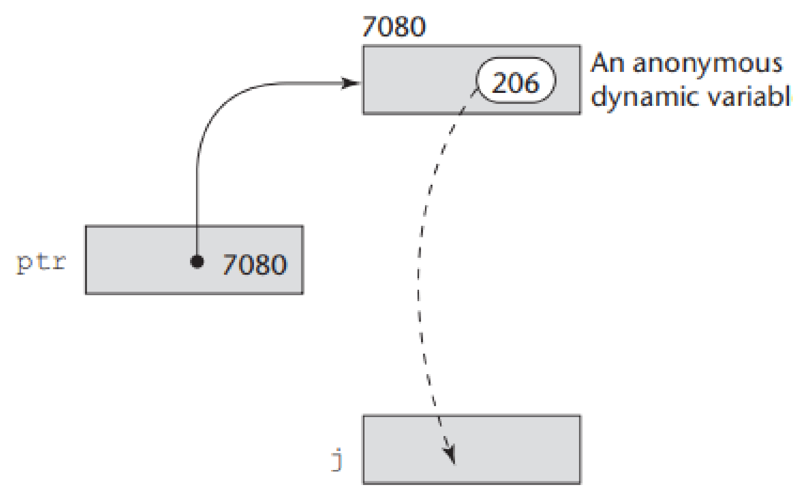
- Fundamental operations:
  - Referencing: assignment of an address to a pointer
  - Dereferencing: explicit vs. implicit

```
int *pNumber; // Decl
pNumber = &number;
```

- Referencing:
  - assignment is used to set a pointer variable’s value to some address.
- Dereferencing:
  - retrieve the value stored at the location represented by the pointer’s value.
  - C++ uses an explicit operation via *

    `j = *ptr`

  sets `j` to the *value* located at `*ptr`.



- Language with a pointer requires (de)allocation operation:
  - C : `malloc, free`
  - OOP: `new, delete`

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a *heap-dynamic variable* that has been *deallocated*.
  - Created when two pointers are aliases:

1. Allocate a new heap-dynamic variable and set a pointer (p1) to point to it.
2. Set a 2nd pointer (p2) to the value of 1st pointer (p1).
3. Deallocate the heap-dynamic variable using the 1st pointer (p1) but p2 is not changed by the operation → P2 is now a dangling pointer. -- p1 and p2 are aliases.

```
int *p2;
int *p1 = new int[100];
p2 = p1;
delete p1;
```

Problems with Pointers (cont.)

- Lost heap-dynamic variable (≈ dangling space/object)
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
  - Created when:
    - Pointer p1 is set to point to a newly created heap-dynamic variable.
    - Pointer p1 is later set to point to another newly created heap-dynamic variable.
  - The process of losing heap-dynamic variables is called *memory leakage.*

```
e.g.) int *p1 = new int[10];
      *p1 = new int[100];
```

Example: Pascal & Ada

- Pascal: used for dynamic memory management only
  - Explicit dereferencing
  - Dangling pointers are possible.
  - Dangling objects are also possible.
- Ada: a little better than Pascal and Modula-2
  - Some dangling pointers are disallowed because *dynamic objects can be automatically deallocated* at the end of pointer's scope.
  - All pointers are initialized to null.
  - Similar dangling object problem (but rarely happens)

Example: 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 memory management and addressing.
- *Pointer arithmetic* is possible.
- *Explicit dereferencing* (*) and *address-of* (&) operators.
- Void pointer: domain type need not be fixed (**`void *`**)
  - **`void *`** can point to any type and can be type checked.
  - Void pointer cannot be dereferenced.
- Example:

```
int *ptr;
int count, init;
. . .
ptr = &init;
count = *ptr;
```

i.e. `count = init`

// dereference `ptr`

Pointer Arithmetic in C and C++

```c
int stuff[100];
int *p;
p = stuff;
```

`p = stuff;` Assign the address of `stuff[0]` (i.e. offset) to `p`.

- `*(p+5)` is equivalent to `stuff[5]` and `p[5]`
- `*(p + index)` is equivalent to `stuff[index]` and `p[index]`
- `p[index]` is equivalent to `stuff[index]`.

Reference Types

- Pointer type vs. Reference type:
  - A pointer refers to an *address* in memory, vs.
  - A reference refers to an *object* or a *value* in memory.
- A reference is an alias, another name for an existing variable.
  - Implemented by storing the.
  - E.g.) `int i = 3;` *address of an object*

```
int b = 6;
int *ptr = &i;        - ptr is a pointer type.
ptr = &b;             - reassignment of pointer variable.
int &ref = i;         - ref is a reference type.
```

- A reference cannot be re-assigned after initialization.
  - E.g.)

```
int i = 3;
int b = 6;
int &ref = i;         - ref is a reference type.
int &ref = b;         - an error of multiple declaration.
int &q = ref;         - i, b, q are referring the same variable.
```

Reference Types (cont.)

- Only *one level of indirection*, no extra level of indirection.
  - E.g.) in pointer type

```
int a = 10;
int *ptr;
int **q;          - valid in pointer type.
ptr = &a;
q = &ptr;         - indirect pointer to pointer.
int &ref = i;     - ref is a reference type.
```

  - E.g.) in reference type

```
int a = 10;
int &ptr = a;      - valid in reference type.
int &&q = ptr;     - reference to reference – ERROR!!
```

- No reference arithmetic

Reference Types (cont.)

- C++ Reference Types:
  - A *constant pointer* that is implicitly dereferenced.
    - Reference typed variable stores the *address* of an object.
      -- Initialized with the *address* of a variable.
    - Then, it *canno*t be set to reference *any other variable*.
    - *lvalue* reference : refer to a named variable
      - Specified by preceding & in the name
    - *rvalue* reference: refer to a temporary object
      - Specified by preceding && in the name
      - A temporary object is an unnamed object created by the compiler to store a temporary value.
  - Two reference typed variables can reference the same object; thus, operations on one variable can affect the object referenced by the other variable

Reference Types (cont.)

- C++ Reference Types:
  - Specified by preceding & in the name.

```
int result = 0;
 int &ref_result = result;
. . .
ref_result = 100;
```

    - `result` and `ref_result` are aliases.
  - Used for parameters:
    - Two-way communication b/t a caller and a callee.
    - a special kind of *reference type* that is used for *formal parameters* in a function definition.
    - Advantages of both *pass-by-reference* and *pass-by-value.*
    - The compiler passes the address to reference parameters.

Reference Types (cont.)

- In Java
  - extends C++'s reference variables
  - Reference variable can be assigned to refer to *different class instances*, not constant.
  - References are *references to objects*, not addresses.
  - E.g.)

```
String str1;
. . .
str1 = "This is a Java literal string";
```

    `str1` is defined to be a reference to a String class instance.
    the assignment sets `str1` to reference the String object.

  - All Java class instances are referenced by reference variables.
- In C#:
  - includes both the references of Java and the pointers of C++.

Evaluation of Pointers

- Dangling pointers and dangling objects (i.e. garbage) are problems as is 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.

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments.
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types.
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type.
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type.

Type Checking (cont.)

- If all type bindings are static, nearly all type checking can be static.
- If type bindings are dynamic, type checking must be dynamic.
- A programming language is *strongly typed* if type errors are always detected.
  - Advantage: allows the detection of the misuses of variables that result in type errors.

Type Equivalence

- *Name type equivalence:*

  two variables have equivalent types if they are

  in either the same declaration

  or in declarations that use the same typed name.

- Easy to implement but highly restrictive:
  - Subranges of integer types are not equivalent with integer types.

    E.g.) In Ada: `count` and `index` are not equivalent below.

    ```
    type Indextype is 1..100;
    count : Integer;
    index : Indextype;
    ```

  - Formal parameters must be the same type as their corresponding actual parameters.

Structure Type Equivalence

- *Structure type equivalence:*

  Two variables have equivalent types if their types have *identical structures*.

- More flexible, but harder to implement.
  - it disallows differentiating between types with the same structure.
  - e.g.) In Ada:

```
type Celsius = Float;
     Fahrenheit = Float;
```

    `Celsius` and `Fahrenheit` are structure type equivalent, so they can be mixed in the expression – undesirable.

  → Derived type:

```
type Celsius is new Float;
type Fahrenheit is new Float;
```

    - new type based on the previously defined type with which it is not equivalent, though identical structure.
    - It inherits all the properties of their parent type `Float`.
  - Subtype: range-constrained version of an existing type.

```
subtype Small_type is Integer range 0..99;
```

Structure Type Equivalence

- More flexible, but harder to implement.
  - Subtype: range-constrained version of an existing type.

    ```
    subtype Small_type is Integer range 0..99;
    ```

  - `Small_type` is equivalent to the type `Integer`.
- Example:

```
type Derived_Small_Int is new Integer range 1..100;
subtype Subrange_Small_Int is Integer range 1..100;
```

  - Both `Derived_Small_Int` and `Subrange_Small_Int`, have the same range of legal values and both inherit the operations of Integer.
  - Variables of type `Derived_Small_Int` are not compatible with any Integer type. vs.
  - Variables of type `Subrange_Small_Int` are compatible with variables and constants of Integer type and any subtype of Integer.

Type Equivalence (continued)

```
type
record-name = record
   field-1: field-type1;
   field-2: field-type2;
   ...
   field-n: field-typen;
end;
```

- Problem of two structured types:
  - Are two *record types* equivalent if they are structurally the same but use different field names?
  - Are two *array types* equivalent if they are the same except that the subscripts are different?
    - e.g.) A[1..10] and A[0..9]
  - Are two *enumeration types* equivalent if their components are spelled differently?
  - With structural type equivalence, you cannot differentiate between types of the same structure
    - e.g. different units of speed, both float

Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management