Data types

- Definitions
- Aggregate constructors
- User-defined type definitions
- Types and storage
- Types and subtypes
- Type systems and type checking
Data types

- **Definition**: a set of values + set of operations, e.g., Boolean, integer
- Program entities (or objects) are **instances** of types:
  
  \[ x, y, z : T; \quad \text{or} \quad T \ x, y, z; \]
Primitive and composite types

- **Primitive** (atomic) types are atomic types (values usually defined by language/machine)
- **Compound** types have components (values are formed from components)
- **Built-in** types are provided by language (NB: built-in types may be not primitive, e.g. string)
- **Type constructors** are used to define new types
Purpose of types

- Classification of the data manipulated by program
  - c: centigrade;
  - f: fahrenheit;
- Protection from illegal operations
  - f := c and f; -- should not be allowed
Advantages of types (1)

- Hiding of underlying representation
  - name versus structure
  - specification versus implementation
  - independence of client from server: change implementation without changing interface and use
  - ⇒ modifiability and portability
Advantages of types (2)

- Correct use of variables can be checked at compile time (if type is known)
  \[ x := i/j \] can be checked for type correctness but not \( j \neq 0 \)
- Resolution of overloaded operators at compile-time (if type is known)
  \[ x := y * z; \]
Defining the *values* of a new type

- Enumeration (also subrange):
  ```
  type color = (white, yellow, red, green)
  ```

- What about operations?
  ```
  succ, pred, < (ordered elements, by position and value)
  ```
Aggregates

- Set of values, one name
- Homogeneous versus heterogeneous aggregates
- Constructors for compound values and types
- Operations on entire aggregate
- Selection of components possible
Aggregate constructors

• **Cartesian product:**
  \[ A_1 \times A_2 \times \ldots \times A_n \]

• Ordered n-tuples (a_1, a_2, ..., a_n)

• Examples:
  - complex numbers
  - point (x, y)

• record, struct, ...

Aggregate constructors

- **Finite mapping** *(domain is finite)*
  - map: DT → RT

- Arrays of most languages

- Is DT fixed at compile-time or instantiation time or object manipulation time?

- Is DT only subrange of integers?

- Is DT extensible at runtime?
Aggregate constructors

• (Discriminated) union
  \[ A_1 \cup A_2 \cup A_3 \]
• Disjunction of values
• What is the type of the object at runtime?
• Requires runtime type checking for correct operation
  
  union u (int i, float f);
Aggregate constructors

- Recursive structures (unbound size)

```c
struct int_list {
  int val;
  int_list* next;
};
int_list* head;
```
Problems with pointers

• Pointers are often used to implement recursive structures
• Pointers establish a dependency between two objects. It is important for the object to exist as long as the pointer points to it
• *Memory leaks* (object exists, but can’t be reached)
• *Dangling pointers* (pointer to deallocated area)
Dangling pointers

```c
int* px;
void trouble ();
{
    int x;
    ...px= &x; ...
    return;
}
main(){
trouble ();
/* now x deallocated ⇒ px dangling */
```
Pointer examples in C++

- int j = 3; const int k = 99;
- int*p = &j;
- *p = 0;
- const int*q = &k; --pointer to a constant
- *q = 10; illegal
- q = &j; legal --questionable but true
- const int*const pc = &k; --const ptr to const
- pc = &j; illegal
References in C++

- A variable declared to be a “T&” (reference to type T) must be initialized by an object of type T (or convertible to a T)
  - int i;
  - int& r = i; //r and i share the same object, i.e.the l-value is the same
  - r=1  //changes value of i to 1
  - int& rr = r; //same object shared by rr, r, i

- a reference cannot be changed after initialization
Type constructor: routine

• int foo (int n, float f)
  { ... }
• Signature: foo : int × float → int
• Can you have variables of type, e.g..
  int × float → int ? (first class objects)
• or pointers to such objects?
  – int (*ps)(int, float);
  – ps = &foo;
• Can you express generic procedures?
Parameter passing in C++

```c++
void f(int val, int& ref) {
    val++;  
    ref++; 
}
```

- `val` is by value ⇒ only local copy is incremented
- `ref` is by reference ⇒ actual parameter is affected
- an argument of type T[] converted to a T* when passed (arrays cannot be passed by copy)
User-defined types

- struct complex {
    float real_part, imaginary_part;
}
...
complex a, b, y;
- variable can be declared and instantiated
- no protection from illegal (undesired) operations this way, attributes are public
Abstract data types

A C++ example

class point {
    private:
        float x, y;
    public:
        point (float a, float b) {x = a; y = b;}
        void x_move (float a) {x += a;}
        void y_move (float b) {y += b;}
        void reset () { x = y = 0.0;}
};
ADTs

point p1 (1.3, 3.7);
point p2 (53.3, 0.0);
point * p3 = new point(0.0, 0.0);
point p4 = point(1.3, 3.7);

p1.x_move(9.3); *p3 = p2;

- Public versus private part
- Implementation versus specification
- Hidden representation
First class types

• Is point a first class type?
• Can I pass a point as argument? Can I write a function to return a copy of a point?

point copy (point p) {
    point temp;
    temp = p;
    return temp;
};
...
• p2 = copy (p1)
The class construct in C++

• Definition and implementation can be compiled separately
  – WARNING: definition ≠ interface!
• Clients must be compiled after servers (interfaces)
• Names local to a class implicitly private, unless declared public
class stack {
private
    int* p;
    int* top;
    int size;
public:
    stack (int n) {top = p = new int[size = n];}
    ~stack() {delete [] p;}  \textit{destructor}
    void push(int i) {(*top++) = i;}
    int pop() {return *--top;}
    ....
}
Object lifetime

- At construction: Object allocated, then attributes initialized as specified by constructor
  - allocation automatic for stack objects or explicit (via new)
- At desctruction: Object attributes cleaned, then object deallocated
  - deallocation automatic for stack objects or explicit (via delete)
Constructor

- Same name as type being defined
- There can be many, with different signatures
- Default constructors are provided if none explicitly defined
Copy constructor

• Special kind of constructor
• Differs from assignment (object does not exist yet)
• Builds an object from an existing one, whose copy is constructed
• Used also for parameter passing (by value)

```cpp
point(const point& p) {
    /* x and y private parts of the object itself */
    x = p.x;
    y = p.y;
}
```
Copy constructor

• Can use the two notations
  – point p1 = p2;
  or
  – point p1(p2)
• Signature is
  – point(const point&)

Ghezzi & Jazayeri: Ch 3
Destructor

• Name of class prefixed by ~
• Performs cleanup actions after last use of an object
• There can be only one destructor for a class
• Default destructor exists if not provided
Assignment for classes

- C++ uses memberwise copy of attributes by default; copy is shallow
- Works okay for point but ...
Memberwise copy and pointers

```c
void f()
{
    stack s1(20), s2(30);
    s1 = s2;  // array referred to by s1 is lost
               // it is not deleted upon exit from block
}
```

*Predefined assignment operator performs memberwise assignment also for pointers*

⇒ *Need a customized assignment operator*
Assignment operation for objects

class stack {

    ...

    public:

    stack (int n) {top = p = new int[size = n];}
    ~stack() {delete [ ] p;}

    stack& operator = (const stack&);

    an assignment operator with this signature is typically present

    ...

    

Ghezzi&Jazayeri: Ch 3
Assignment operation for objects

stack& stack::operator=(const stack& s)
{
    if (this != &s) { //to cover s = s;
        delete[ ] p;
        p = s.p;
        top = s.top;
        size = s.size;
        return *this;  //this: pointer to current object
    }
}

This is the typical predefined assignment but for stacks you need actual cloning
How to assign by cloning?

stack & stack::operator=(const stack & s)
{
    if (this != &s) { //to cover s = s;
        delete [] p;
        p = new int[size= s.size]; // NB: a new one is allocated
        top = p + s.top - s.p;
        for (int i = 0; i < (top-p); ++i) {p[i] = s.p[i];}
        return *this;
    }
}

Copy constructor for stack

```cpp
stack::stack(const stack& s) {
    p = new int[size= s.size];
    top = p + s.top - s.p;
    for (int i = 0; i<(top-p); ++i) {
        p[i] = s.p[i];
    }
}
```
ADTs in C++: summing up

- Constructor: to initialize the object
- Destructor: to clean up after object
- Public part (excluding method bodies): spec
- Private data and functions (and method bodies): implementation (default is private)
Friend functions in C++

- Class vector and matrix are given
- We wish to define operation to multiply a vector and a matrix: where should it be placed?
  - Does it belong to matrix?
  - Does it belong to vector?
- If global, it has no access to private attributes
- Can be declared friend of both
Given classes

class matrix {
    ...
    friend vector mult (const matrix&, const vector&);
    //can be placed in either private or public part
}

class vector {
    ...
    friend vector mult (const matrix&, const vector&);
    //can be placed in either private or public part
}
Friend function

vector mult (const matrix& m, const vector& v);
{ //assume 0..3 matrix and vector
    vector r;
    for (int i = 0; i<3, i++) { //r[i] = m[i] * v
        r.elem(i) = 0
        for (int j = 0, j<3; j++)
            r.elem(i) += m.elem(i, j) * v.elem(j);
    }
    return r;
}
Objects and visibility

• stack s1(20);
  – can perform push and pop
    • s1.push(5);
    • int i = s1.pop();
  – top, size, and p are hidden

• stack * s1 = new stack (20);
  – can dereference and then use push, pop
    • *s1.push(5); abbreviated as
    • s1->push(5)
Generic (or parameterized) ADTs

template<class T> class Stack{
public:
    Stack(int sz) {top = s = new T[size = sz];}
    ~Stack() {delete[] s;}  // destructor
    void push(T el) {*top++ = el;}
    T pop() {return *--top;}
    int length() {return top - s;}

private:
    int size;
    T* top;
    T* s;
};
Use of generic ADT

```cpp
void foo() {
    Stack<int> int_st(30);
    Stack<item> item_st(100);
    ...
    int_st.push(9);
    ...
}
```
C++: not only classes

- C++ modularizes a system also based on functions, not only classes
- Functions may be generic
  - generic algorithms
A generic C++ function

template <class T>
void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

int i, j;
char x, y;
pair<int,string> p1, p2;
//pair is a generic class
//p1, p2 are non-generic pairs
...
swap(i, j); //swap integers
swap(x, y); //swap characters
swap(p1, p2); //swap pairs
Java Generics (1)

class gStack<T> {
    private List<T> stack = new ArrayList<T>();

    public void push(T a) { stack.add(a); }
    public T top() { return stack.get(stack.size()-1); }
    public void pop() { stack.remove(stack.size()-1); }

    public void print() {
        System.out.print("[");
        Iterator<T> iterator = stack.iterator();
        for (T el : stack) {
            System.out.print(el + " ");
            System.out.println(" ");
        }
    }
}
Java Generics (2)

class Example
{
    public static void main(String[] args)
    {
        gStack<Integer> IntegerStack = new gStack<Integer>();

        IntegerStack.push(3);
        IntegerStack.push(4);
        IntegerStack.push(5);
        IntegerStack.print();
        System.out.println("top = " + IntegerStack.top());
        IntegerStack.pop();
        IntegerStack.print();
    }
}

Java Generics - Implementation

• Generics are implemented by the Java compiler as a front-end conversion called erasure
• Erasure is the process of translating or rewriting code that uses generics into non-generic code
• Thus erasure maps the new syntax to the current JVM specification
• This conversion erases all generic type information
  – all information between angle brackets is erased.
    • For example, LinkedList<Integer> will become LinkedList.
  – when the resulting code is not type correct, a cast to the appropriate type is inserted.
Java Generics vs. C++ templates

• Java generics look like the C++ templates, but they are not the same.

• Java Generics
  – They provide compile-time type safety and eliminate the need for casts.
  – The main difference is encapsulation: errors are flagged where they occur and not later at some use site, and source code is not exposed to clients.
  – Use type erasure: the compiler keeps track of the generics internally, and all instances use the same class file at compile/run time.

• C++ template
  – They are macros
  – Whenever a template class is instantiated with a new class, the entire code for the class is reproduced and recompiled for the new class.
Type systems (1)

- *Type system* is set of rules used by language to structure and organize its types
- Objects are *instances* of types
- Incorrect use of an object is a *type error*
- A program is *type safe* if it has no type errors
Type systems (2)

• A type system is *strong* if it guarantees type safety
  – Programs written according to the rules of the type system are guaranteed not to contain type errors

• A language with strong type system is *strongly typed*

• If a language is strongly typed the compiler can guarantee type safety (*static checking*)
Static vs dynamic type checking

• Static checking: done by compiler before program execution
• Dynamic checking: requires program execution
• Static type checking advantageous in terms of program correctness (independent of input data) and efficiency of checking (dynamic checks slow down execution)
Static vs strong typing

- Static type system: expressions bound to a type at compile time
- Dynamic type system → polymorphic
- A static type system is strong
- A strong type system is NOT necessarily static (do not confuse a static type system with static type checking!)
- There are languages that are polymorphic and yet strongly typed (we will see the case of inheritance in OO languages)
Issues defined by type systems

• Type compatibility
  – When can an object of type T1 be used instead of an expected object of type T2?

• Type conversions
  – How/when an object of type T1 can be transformed into an object of type T2?

• Types and subtypes
  – What kind of compatibility rules between type and subtype?
Type issues: compatibility
(or conformance, or equivalence)

• Type T1 is compatible with type T2 with respect to operation O if it can be used in place of T2 in operation O

x: T1; y: T2; f(T2) {...}
y = x;
...

f(x);
Type compatibility rules

• Within the context of an operation defined for an object of type T, only type T is acceptable
  – Also called name compatibility (a type name is compatible only with itself)

• Structural compatibility
  – two different struct types with fields having the same types (e.g., COMPLEX and COORDINATE) are compatible
Type issues: conversions

- Conversions (casting)
  
  \[
  \text{fun: } T_1 \rightarrow R_1 \\
  x: T_2; \ y: R_2; \\
  \]

  How to call: \( y = \text{fun}(x) \)\

  - Use of conversion routines \( t_{21} \) and \( r_{12} \):
    
    \[
    t_{21}: T_2 \rightarrow T_1 \\
    r_{12}: R_1 \rightarrow R_2 \\
    y = r_{12} (\text{fun} (t_{21} (x))) ; \\
    \]

- Called \textit{coercion} when performed automatically and implicitly by the compiler

- Warning: interaction between coercion and overloading may lead to obscure programs and unexpected behavior
Types and subtypes

• type natural = 0..maxint;
  digit = 0..9;
  small = -9 .. 9;

• Compatibility rules? Substitutability.
  x: integer; n: natural; d: digit; s: small;
  ...
  x := n;  --okay
  n := x;  --? requires runtime check
Monomorphic vs polymorphic TS

- **Monomorphic type systems**
  - All program entities are typed
  - Every operation accepts operands only of exactly the defined type
  - \(\Rightarrow\) strongly typed but too strict, no flexibility

- **Polymorphic type systems**
  - Some program objects can belong to more than one type

- All “practical” prog. lang. provide some polymorphism

- Goal: combine polymorphism (\(\Rightarrow\) flexibility) + strong typing + efficient, preferably static type checking
Classification of polymorphism

(the case of routines)

- Ad hoc: finite (and small) set of types
- Universal: infinite set of types