Programming Methodology

Spring 2009

Types

교육과학기술부
국가지정연구실
National Research Lab

Topics

Definition of a type

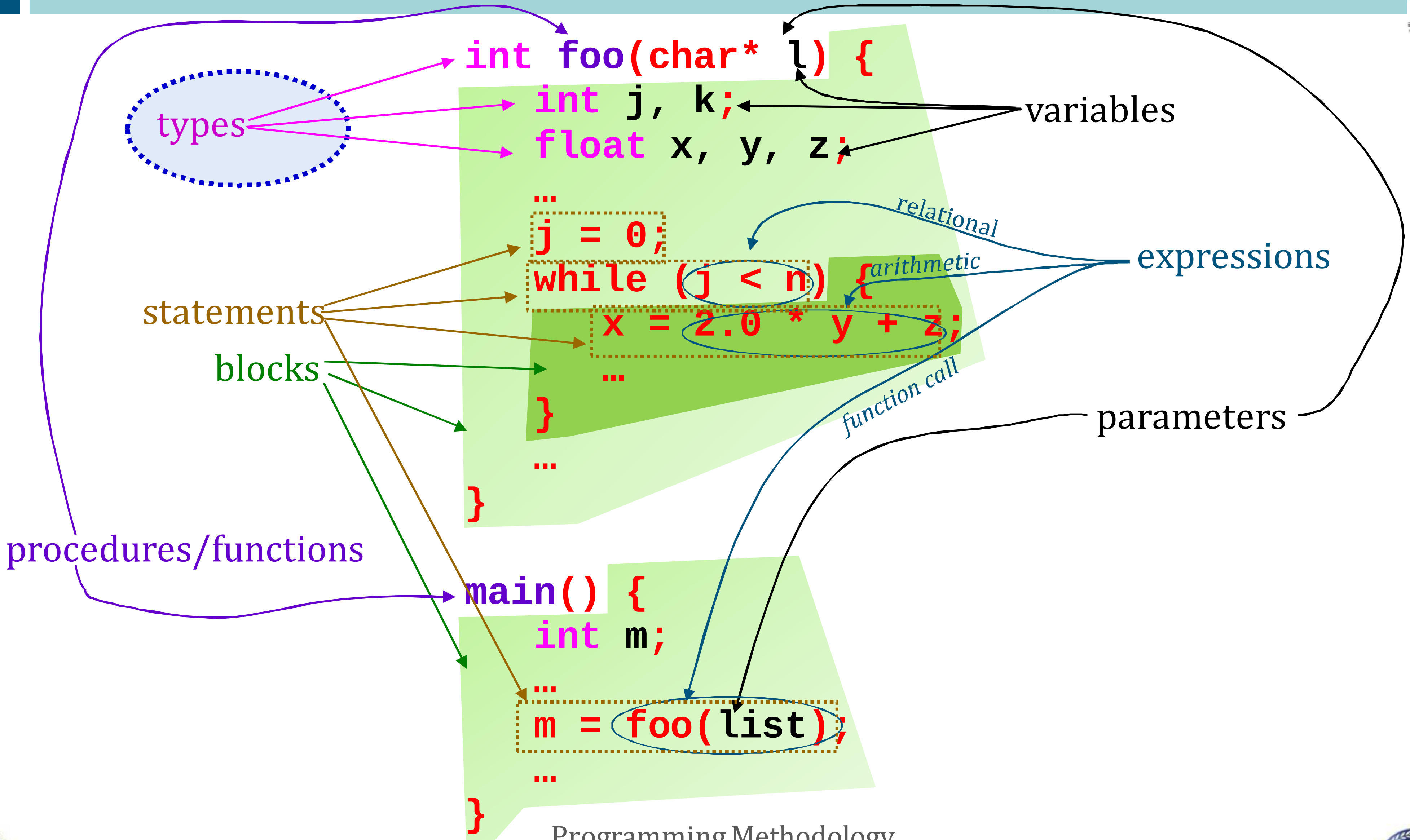
Kinds of types

Type binding and checking

Type conversion

Polymorphism

Programming language constructs



```
int foo(char* l) {
    int j, k;
    float x, y, z;
    …
    j = 0;
    while (j < n) {
        x = 2.0 * y + z;
        …
    }
    …
}

main() {
    int m;
    …
    m = foo(list);
    …
}
```

types

variables

relational

arithmetic

expressions

statements

blocks

function call

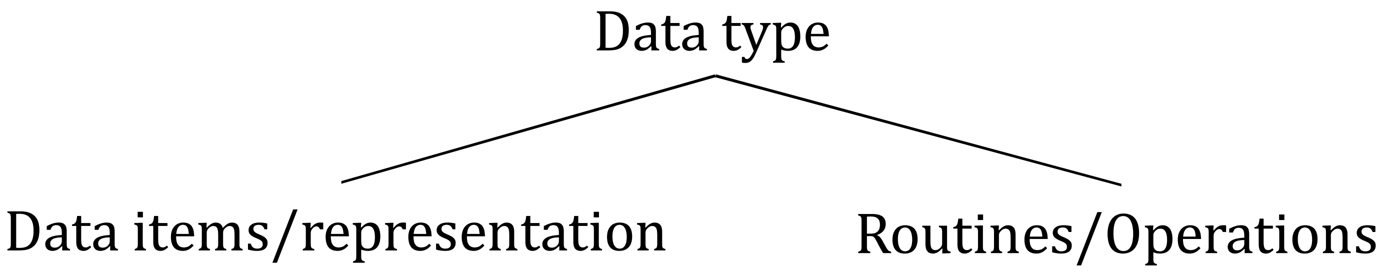
parameters

procedures/functions

Types

- Types correspond to *naturally occurring categories*, such as those found in the real world.
  - Ex: student, university, car, integer, character, string, stack, heap, …
- Types enable us to represent/operate-on bundles of descriptive data items/attributes for each category with a single name.
  - Student has name, GPA... University has departments… String has length…
  - GPA has a real value that is recalculated every semester.
  - Departments are newly added/removed, or have new names.
  - Stack length is increased or decreased at time goes by.
- Once a new type is defined, you can construct any number of class objects that belong to that class
  - student objects, university objects, integer objects, stack objects, …

Components of a data type



Data type

Data items/representation

Routines/Operations

- a set of data items/representation that model a collection of abstract objects in the real world

  ex: in C language

  - **`int`** ← integers, student id, # of departments, stack length ...
  - **`char[]`** ← letters, names, ...

- a set of operations/routines that can be applied to the objects
  - **`int`**: **`+ - * /`** ← add, subtract, multiply, divide for integers
  - **`char[]`**: append, copy, concatenate

Using types …

- improves readability and writability.

Ex:
```
char* student_name;
struct employee_records {
        char* name;
        int salary;
          …
}
```

- reduces programming errors.

Ex:
```
student_name / 5
```

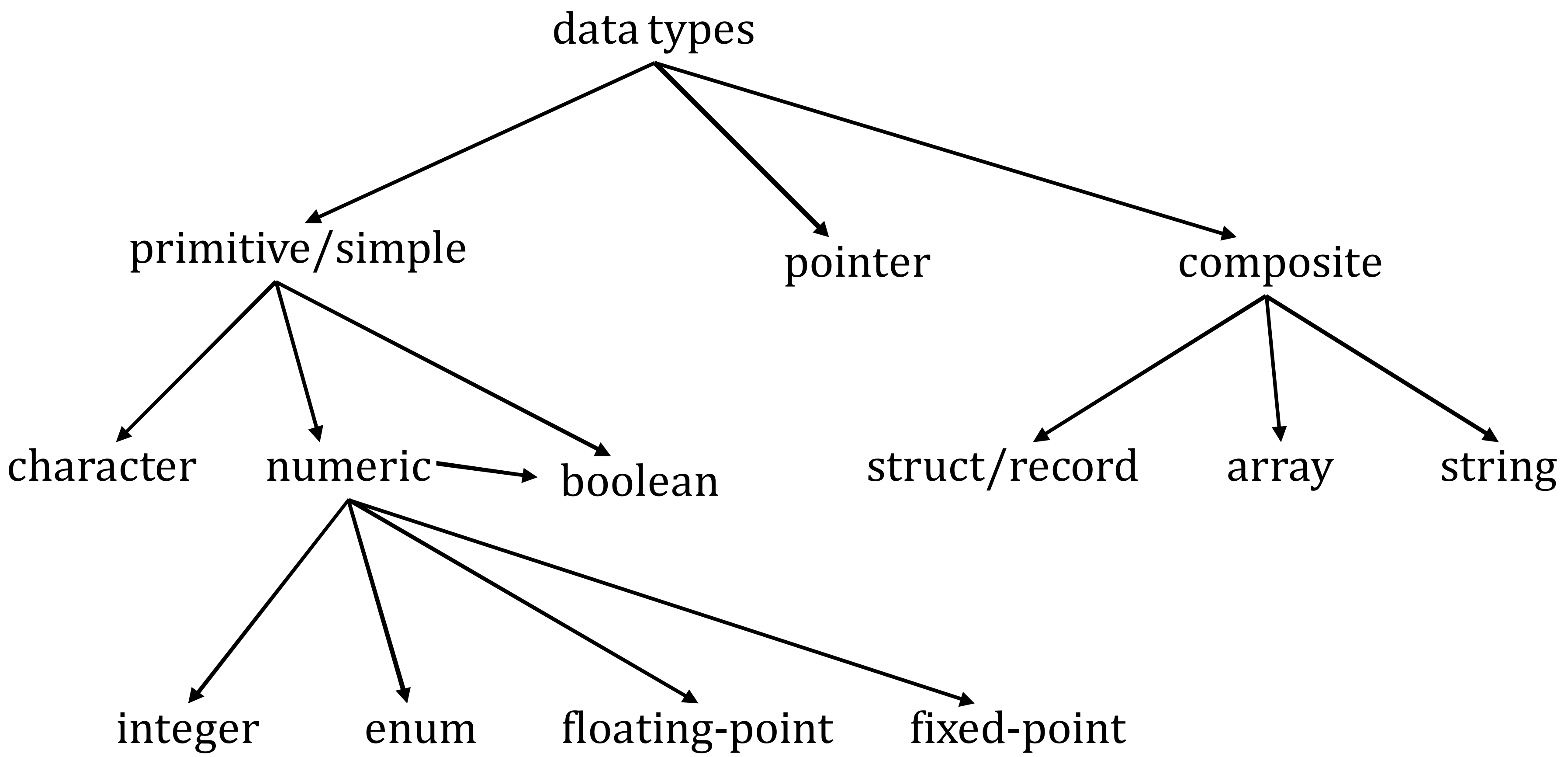
- makes memory allocation and data access efficient.

Ex:
```
struct {
        int i;
        char* c;
}             // 8 bytes
```

→ Sizes are statically known.

→ Useful for the compiler to optimize memory allocation (e.g., use stack in stead of heap)

Hierarchies of types in most languages



data types

primitive/simple

pointer

composite

character

numeric

boolean

struct/record

array

string

integer

enum

floating-point

fixed-point

C/C++ → char, int/long, float/double, struct/class, array, string(?)

Java → … boolean

no clear distinction between char and int, no special op for char/string

Selection of data types

- "What kinds of types should be included in a language" is very important for programming language design.
- Primitive/simple types are supported in almost all existing programming languages
- Composite types being supported differ from language to language based on what is the purpose of the language.
- Several issues related to the selection of types
  - fixed-point vs. floating-point real numbers
  - array bounds
  - structure of composite types
  - pointer types
  - subtypes

Cobol for string type ?
The class type for OO languages

Fixed-point vs. Floating-point

- fixed-point → decimal point for decimal numbers
  - Precision and scale are fixed.

    → a fixed radix point for all real numbers of the same type
  - ex: salary amount of graduate assistants

    → 6 digits for precision and 2 digits for scale

    → 1234.56, 2000.00
- floating-point
  - radix points are floating
  - ex: 21.32, 9213.1, 4.203e+9
- COBOL, PL/1 and Ada support the fixed-point real type, but most of other languages (Fortran, C, ...) don't.
  - Ada: `type salary is delta 0.01 range 0.0..3000.0`
  - C++: `float salary;`

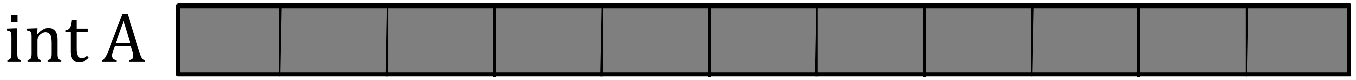
Fixed-point vs. Floating-point

- Problem with fixed-point
  - possible loss of information after some operations at run-time
    - ex: *double the salary of EE students!*
- Problem with floating-point
  - Large numbers may be machine-dependent.
    - ex: *port a C-program to 32-bit and 64-bit machines!*
  - Less secure
    - ex: *double the salary illegally*

Composite types

- array/string types
  - a homogeneous aggregation of data elements

    → student ids, exam scores, profit earned every day, ….
  - An individual element is identified by its relative position in the array.

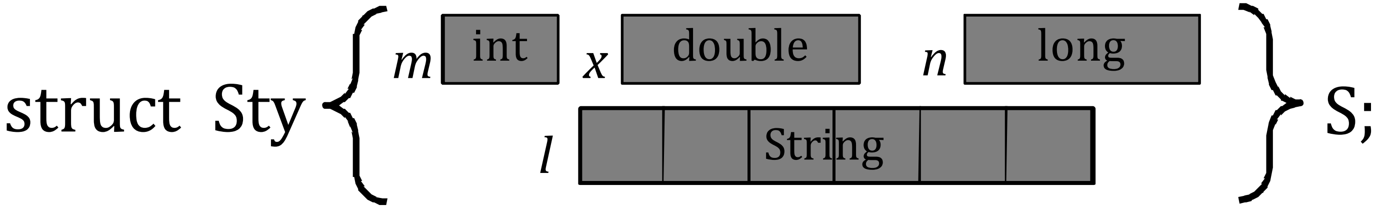


```
int A[11];
```

```
A[0] A[1] A[2]  …
```

- struct/record/union types
  - used to model collections of data that are of heterogeneous forms

    → personal record of each student (name,addr,sex,id,…}, table entries,…
  - Each data element is identified by its name.

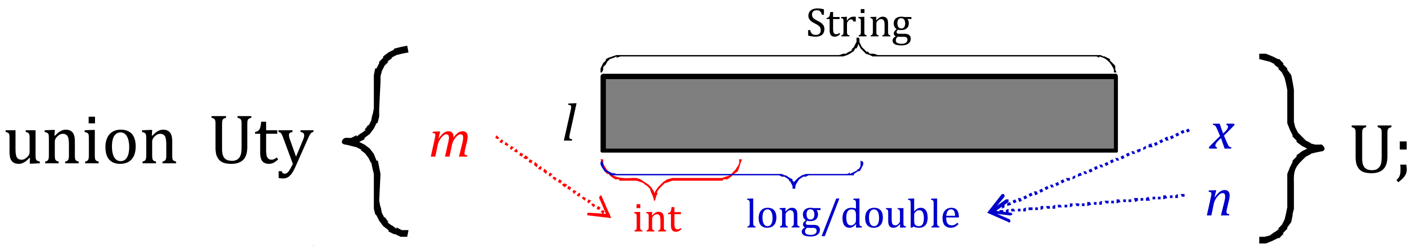


```
struct Sty {
    int m;
    double x;
    long n;
    String l;
} S;
```

```
S.m …  S.x …  S.n  …  S.l …  S.l[3]
```

Composite types

- union types
  - Similar to struct types in that both store heterogeneous data.
  - A union saves memory by sharing locations among its elements.
  - It may store different type data elements in the shared locations at different times during program execution.



- Ex: a group of members in a company
  - Everybody has a name and a SS#.
  - An employee has the department that he/she belongs to.
  - An employer (stock holder?) has the percentage of share in the company.

```
struct Member {
  String name;
  long SSN;
  union { String dept;
          float share; } f;
};
…
Member M[100];
… M[i].name … M[i].f.dept …
```

Determination of array bounds

- static arrays (C/C++, Fortran, Pascal)

  array bounds determined at compile time and static storage allocation.

  ex: `int a[10], b[5];` → *efficient*

- dynamic (C/C++, Fortran90):

  array bounds determined at run time and dynamic storage allocation

  ex:

```
int *a, *b;
a = b = new int[10]
      …
delete [] a;    //  a is freed only when explicitly demanded
 … b[3] …          // error: dangling pointer
b = new int[20];
```

  → *expensive, but flexible (useful for scientific/engineering apps)*

- stack-dynamic (C/C++, Ada)

  array bounds determined at run time but static storage allocation

  ex:

```
void foo (int n) {
   int* q = (int*) alloca(n);  // stack dynamic
   int* p = new int[n];        // dynamic
       …
}                              //q is freed when foo returns
```

Array bounds as a data type

- Many languages (e.g., C/C++ and Fortran) don't include array bounds as a type.
  - needs complex memory managements and error-prone

ex:

```
float *x = new int[5]; // OK
x[i] = …;                   // what if i > 5?
x = new int [10];                 // OK
```

  - but more flexible → *consider the function* **`foo`** *in the Pascal code*
- Some languages (e.g., Java and Pascal) where security is of high priority include array bounds as a data type.

  ➔ What does this imply to programming?

Array bounds as a data type

- Including array bounds as a data type implies …
  1. All arrays are static.
  2. Illegal array assignments & parameter passing can be detected at compile time, or critical faults will be prevented at run time via exception control.

ex:

```
int z [] = new int [5];            // OK: creation w/o initialization
int y [] = { 3, 6, 9, 12, 15 };  // OK: creation w/ initialization
int x [];
x = new int [5];                   // OK
x = new int [10];                  // error: x’s size must be static!
z[5] = …;         // error: out of bound ← detectable at compile time
… = y[i] + …      // if i ≥ 5 or i < 0, an exception will be raised at run time
```

Exception in thread “main” ava.lang.ArrayIndexOutOfBoundsException: line#

Equivalence in struct types

- Is this assignment legal?

```
struct man { char* name; int age; }
struct woman { char* name; int age; }
man Tom;
woman Jane;
   …
Jane = Tom;
```

  - If yes → *structure equivalence*
  - If no → *name equivalence/compatibility*
  - For C++ : *error*!   "**`man`** cannot be converted to **`woman`**"

  ➔ To avoid this error, assignment operator '='for class woman/man must be explicitly defined for type conversion from man to woman

- Pros and cons of name equivalence
  - easy type checking by string comparison → fast compilation
  - more secure and less error-prone compilation → `Jane = Tom;` (*unsafe!*)
  - But!... less flexible programming
    - Cannot be compatible with anonymous type →
    - Type must be globally defined. → Why?

```
struct {
    char* name;
    int age;
} Tom;
```

The pointer type

- In some (old) languages, the pointer has no data type.
  - This is more flexible and may save memory, but is error-prone!

```
declare p pointer;
declare i integer;
declare c, d char;                      PL/1
. . .
p = address of(c);
. . .
p = address of(i);
d = dereference(p);      //  Error won't be detected until run time
```

  - Memory cost becomes less critical, and S/W quality is more important!
- Thus, in most existing languages, the pointer type is a part of data types.

```
int* p;
int i;
char c, d;                C/C++
. . .
p = &i;
d = *p;          // Error can be detected by the compiler
```

User defined types

- Ordinal types
  - The range of possible values can be easily associated with the set of positive integers. → i.e., the values are countable.
  - primitive ordinal types in C/C++/Java: `char`, `int`, `long`, `boolean`(Java)
  - Language constructs for user defined ordinal types
    - `enum`(C/C++/C#), `subrange`(Pascal,Ada)
    - The user defined ordinary types are called ***subtypes***, since they are created by restricting existing primitive types.
- Abstract data types
  - More generalized forms of user defined types
  - Languages provides constructs for encapsulating the *representation* for a certain data type and the subroutines for the *operations* for that type.
  - `class` (Smalltalk/C++/Java), `package`(Ada)

Why do we need subtypes?

- Primitive types provided by languages are not enough. Why?

```
int day, month;
month = 9;    // It's OK…but need more…
day = -11;    // Non-sense! Semantic error! may not be caught even at run time
```

➔ *How can we capture this semantic error with data types?*

- Users need to restrict the primitive types. ➔ *subtypes*
  - enumerated types (C++, Pascal)
    - *C++*

```
enum day_type {first = 1, second, …, thirty_first};
enum month_type {Jan, Feb, . . . , Dec};        // Jan = 0
day_type day;
month_type month = Sep;     // That's better. More readable.
day = 0;                    // Error detected at compile time!
```

    - *Pascal*

```
type month_type = (Jan, Feb, . . . , Dec);
```

  - subrange type (Pascal) - in some case, more <u>compact and flexible</u>

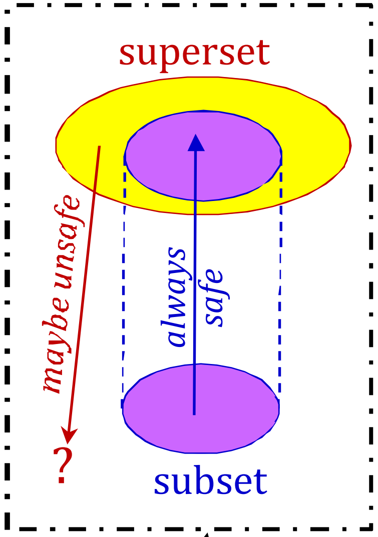
enum type for 1..99999?

```
subtype day_type is integer range 1..31;
day := -11;                   // Error still can be detected.
day := day + 20;        // Also it can be used in integer operation.
                 // This may be error. But error can be easily detected at run-time
```

Subtype example with C++ **enum**

- The enum type improves software quality in terms of readability and maintainability. (Recall Note 1)

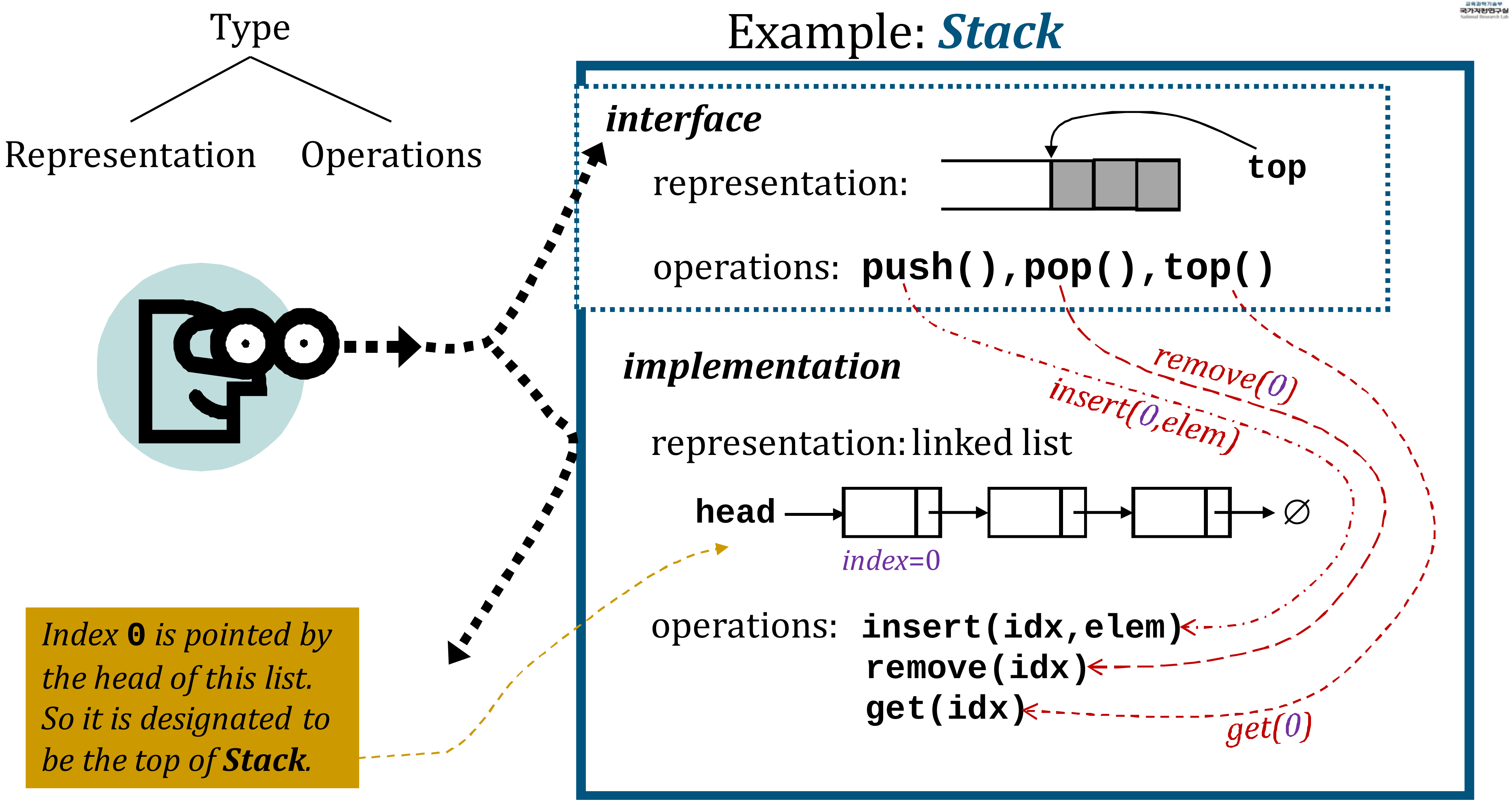
```
enum Month {Jan = 1, Feb, . . . , Dec}; // subtype of int
Month m;
int n;
…
while (m >= Jun && m <= Aug)     // during summer
 …
for (n = Jan; n <= Dec; n += 2)   // for every odd month
 …
cout << “enter 1 for January, 2 for February, …”;
cin >> n;         // enum type variable cannot take integer directly
switch (n) {                 // for each different month
   case Jan: …
   case Feb: …
```



- Compatibility between primitive types and subtypes

```
Month m = Mar;
int n = m;       // OK!  Conversion from Month to int
n = Jun + 1;     // n = 7, indicating July
m = n + 1;       // must be compiler error: Conversion from int to its subtype
m = m + 1;       // error… why?     These two errors are due to ensuring type safety
m = (Month) (n + 1);    // OK!  If users insist type conversion, it must be done so.
```

Abstract data types



Type

Representation Operations

Example: ***Stack***

interface

representation:

operations: `push(),pop(),top()`

implementation

representation: linked list

operations: `insert(idx,elem)`
`remove(idx)`
`get(idx)`

Index **`0`** *is pointed by the head of this list. So it is designated to be the top of* ***Stack****.*

Implementing the ADT Stack w/ `class`

A *class* defines an ADT (blue-print?).

- (private?) data representation
- (public?) methods for operations

An *object* is an instance of that ADT.

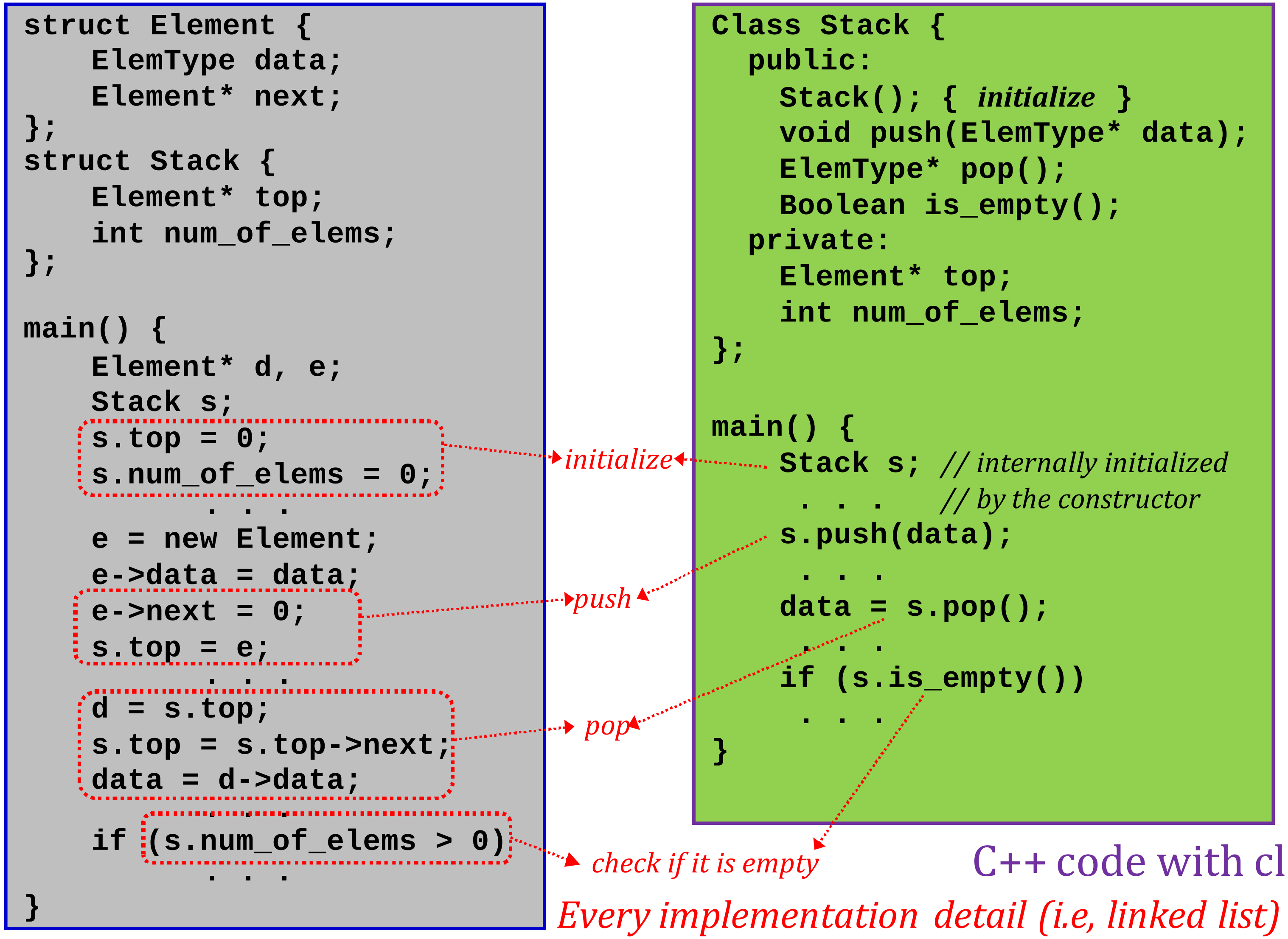
- It is created by declaring a **variable** of the ADT or by dynamic allocation
- All instances share all their methods, but each one has its own set of data.

```
template <class T> class Stack {
  public:
    Stack();
    void push(T& elem);
    T* pop();
    T& top();
    int is_empty();
  private:
    LinearList<T>* L;
};
```

```
template <class T>
T* LinkedStack<T>::pop() {
  if (is_empty())
    exit(1);      // error
  return L->remove(0);
}
template <class T>
void LinkedStack<T>::push(T& data) {
  L->add(0,data);
}
```

```
main() {
  T* x, y, z;
  Stack<int>* st = new Stack<int>();
   …
  st->push(*x);    // insert x to the array
  st->push(*y);    // insert y to the linked list
  z = st->pop();
   …
```

Why do we need user-defined ADTs?



```
struct Element {
    ElemType data;
    Element* next;
};
struct Stack {
    Element* top;
    int num_of_elems;
};

main() {
    Element* d, e;
    Stack s;
    s.top = 0;
    s.num_of_elems = 0;
           . . .
    e = new Element;
    e->data = data;
    e->next = 0;
    s.top = e;
           . . .
    d = s.top;
    s.top = s.top->next;
    data = d->data;
           . . .
    if (s.num_of_elems > 0)
           . . .
}
```

ordinary C code

```
Class Stack {
  public:
    Stack(); { initialize }
    void push(ElemType* data);
    ElemType* pop();
    Boolean is_empty();
  private:
    Element* top;
    int num_of_elems;
};

main() {
    Stack s;  // internally initialized
      . . .   // by the constructor
    s.push(data);
      . . .
    data = s.pop();
      . . .
    if (s.is_empty())
      . . .
}
```

C++ code with class Stack

initialize

push

pop

check if it is empty

Every implementation detail (i.e, linked list) is exposed w/o ADT Stack.

The ADT Stack hides low-level details of its implementation.

So in the code, it acts like ordinary types such as **`int`** or **`char`**.

Thus the Stack object can be treated in the same way as the objects of other types.

Consequently, the resulting code has better readability and maintainability.

Topics

Definition of a type

Kinds of types

Type binding and checking

Type conversion

Polymorphism

Type checking

- Recall → **data type** = set of **data items** + set of **operators**
- A *data item* is **compatible** with an operator if it can be passed to the operator as the operands.

```
int i, j;
i * 3                // legal: integer value(= item of int type) is compatible with *
i * “eesnu”          // illegal: string (of char[] type) must not be compatible with *
Σ(1.3, 3.01, 2.0)  // legal
Σ(3.2, j, i)         // illegal in principle (but in reality it can be avoided → ex: coercion)
```

- **Type error** occurs if an operator is applied to *incompatible* items.
- A program is **type safe** if it results in no *type error* while being executed.
- **Type checking** is the activity of ensuring that a program is *type safe.*

Static vs. dynamic type binding

- In static type binding, a variable …
  - is bound to a certain type by a declaration statement, and
  - should have only one type during its life time.

```
float x;            // x is of a real type
char* x;            // This is an error
```

➔ most existing languages such as Fortran, PL/1, C/C++ and ML

- In dynamic type binding, a variable …
  - is bound to a type when it is assigned a value during program execution, and
  - can be bound to as many types as possible.

```
> (define x 4.5)            // x is of a real type
> (define x '(a b c))       // now, x is of a list type
```

➔ Scheme, LISP, APL, SNOBOL, ***virtual functions*** in C++

Static type checking

- Type checking performed during compile time
  - Pascal, Fortran, C/C++, Ada, ML, …
  - The type of an expression is determined by static program analysis.
- To support static type checking in a language, a variable (or memory location) and a procedure must hold only one type of values, and this type must be statically bound or inferred.
  - *Pascal*

```
var x : real;
    w : array of [1..10] of real;
function foo(n : integer): real
       . . .
begin
   w[9] := foo(5) / w[1];              // OK
   w[10] := foo(x) + 3.4;              // error
```

  - *C++*

```
#include <stream.h>
main() {
   int i = bar();          // error: undefined function bar
```

Dynamic type checking

- type checking performed during program execution
- required by languages that
  - perform dynamic type binding, or

```
class Employee {
    virtual void print() = 0;      // pure virtual
};
class Researcher: public Employee {
    virtual void print() { . . . }
};
class Secretary: public Employee {
    virtual void print() { . . . }
};
   . . .
Employee* p;
if (…) p = new Secretary else p = new Researcher;
p->print();      // which print() is called is determined at run time
```

  - check the value of a program variable at run time.

```
enum Month {Jan = 1, Feb, . . . , Dec};
Month m;
int n;
…
m = (Month)(n * 2);   // no compiler error, but is 1 ≤ n ≤ 6?
```

Strongly vs. weakly typed languages

- strongly typed (Java, Ada, ML, Pascal) if (almost) all programs are guaranteed to be type safe by static/dynamic type checking
  - J2SE Ver.5

```
enum Month = {Jan, Feb, …, Dec};
Month m = Feb;
int x;
boolean b;
x = m;      // Java discourages this conversion, although C++ allows it
while (x < 100) …        // OK
if (x++) …               // error
if (b) …                 // OK
```

- weakly typed or untyped (Fortran, C/C++, Scheme, LISP)
  - *C++*

```
float foo(char cc, float x) { cout << cc << x; }
main() {
   float x = foo(100.7,'c');   // it runs! ➔  output:  d 99
   if (x++) …             // OK: boolean 'true' if y ≠ 0
```

char(100.7) → char(100) → 'd'
float('c') → 99.0

  - *Fortran*

```
real r
character c
equivalence (r,c)
print*, r;
```

// maybe wrong, but maybe still OK

Overloading

- Often it is more convenient to use the same symbol/operator to denote several values or operations of different types.
  - *Pascal*

```
subtype day_type = 1..31
```

→ The numbers 1 ~ 31 are *overloaded* because the numeric symbols of type `day` are also of type `integer` in Pascal.

  - *C++*

```
int ::operator+(int, int) {  . . . }
float ::operator+(float, float) {  . . . }
```

→ This built-in symbol + is overload because it is used for the addition for integer and real types.

- In C++, users can overload operators with the class construct.

```
class complex {
      . . .
      complex operator+(complex, complex);
}
```

Overloading

- Type checking tries to resolve ambiguities in an overloaded symbol from the context of its appearance.

```
float foo(int x) { … }
void foo(char* y, HerType z) { … }
…
main () {
   enum Month = {Jan, Feb, …, Dec};
   int a, b;
   Month m;
    …
    … 4.3 + 1.3 …   // float addition
    … a + b …       // integer addition
    … foo(a) …      // foo(int x)
    … 3.46 + 2 …    // float addition???
    … m + 1 …       // integer addition???
    foo(a,b,m) …    // foo(int,int,Month)???
```

```
float foo(int x) {
   // code for this foo
}
```

```
void foo(char* y, HerType z) {
   // code for this foo
}
```

```
float ::operator+(…) {
   // code for float add
}
```

```
int ::operator+(…) {
   // code for integer add
}
```

- If the ambiguity cannot be resolved, type error occurs.

Type conversion

- In order to allow **`3.46 + 2`** instead of **`3.46 + 2.0`**, one solution is to create extra two overloaded functions

```
float ::operator+(float, int) {  . . . }
float ::operator+(int, float) {  . . . }
```

→ But, this solution is tedious and may cause exponential explosion of the overloaded functions for each possible combination of types such as **`short, int, long, float, double, unsigned,`** …

- A better solution: type conversion

→ *convert the types of operands*.

- Two alternatives for type conversion
  - explicit: **type cast**
  - implicit: **coercion**

Type cast

- Explicit type conversion

```
enum Degree = {LOW = 0, MID = 10, HIGH = 100};
float x = 3.46 + (float) 2;
int *ptr = (int *) 0xffffff;
Degree d = MID;
x = x + (float) *ptr;
*ptr = 5 * (int) d;
```

- Drawback of type cast
  - Heedless explicit conversion may invoke *information loss*. (e.g. truncation)

```
enum Degree = {LOW = 0, MID = 10, HIGH = 100};
float x = 3.49;
int n = (float) x;          // no error, but 0.49 is truncated
Degree d = (Degree) n;  // no error, but semantically incorrect (no valid Degree value)
```

  - A solution? → implicit type conversion (*coercion*)!
    - Languages provide coercion to coerce the type of some of the arguments in a direction that has preferably no information loss.

Coercion

- is the rule in most languages (PL/1, COBOL, Algol68, C, Java).
- provides a predefined set of implicit coercion precedences.
  - Generally, a type is widened when it is coerced.
  - *C/Java*

```
character → int
  pointer → int
      int → float
    float → double
          …
```

➔ Convertion table (**D**: double, **F**: float, **I**: integer, **L**: long)

Calculation	Types	Result type	Stored in variable of type	divide operation
x = 5.0/3	D/I	D	D	`double ::opeator/`
y = 5/3.0F	I/F	F	F or D	`float ::opeator/`
z = 5/3	I/I	I	I L F or D	`integer ::opeator/`
v = 5/3.0	I/D	D	D	`double ::opeator/`

→ A constant like 3 is double by default.

Coercion

- Coercion may still suffer information loss.

  Ex: 32 bit integer → 32 bit float with 23 bit mantissa
  - Format for a 32-bit floating-point number (IEEE standard 754-1985)
  - Value = $\pm 1.f \times 2^{e-127}$ (exponent → a *biased* sign number)
  - 32 bits = 1 (sign) + 8 (exponent) + 23 (fraction)
  - 110000011111000000000000000000000 → 1.875×2^4

  → So this is safe to coerce an **`int`**-*constant* to double. (ex: 5/3.0)
- The generous coercion rules make typing less strict, thereby
  - possibly causing run time overhead (e.g., COBOL),
  - making code generation more complicated, and
  - in particular, making security that typing provides lost by masking serious programming errors.

  → *For the last reason, some strongly typed languages* (*Ada, Pascal*) *has almost no coercion rules.* (*but, possibly due to compatibility with C++, Java allows the same coercion rules as C++*)

Monomorphic/polymorphic objects

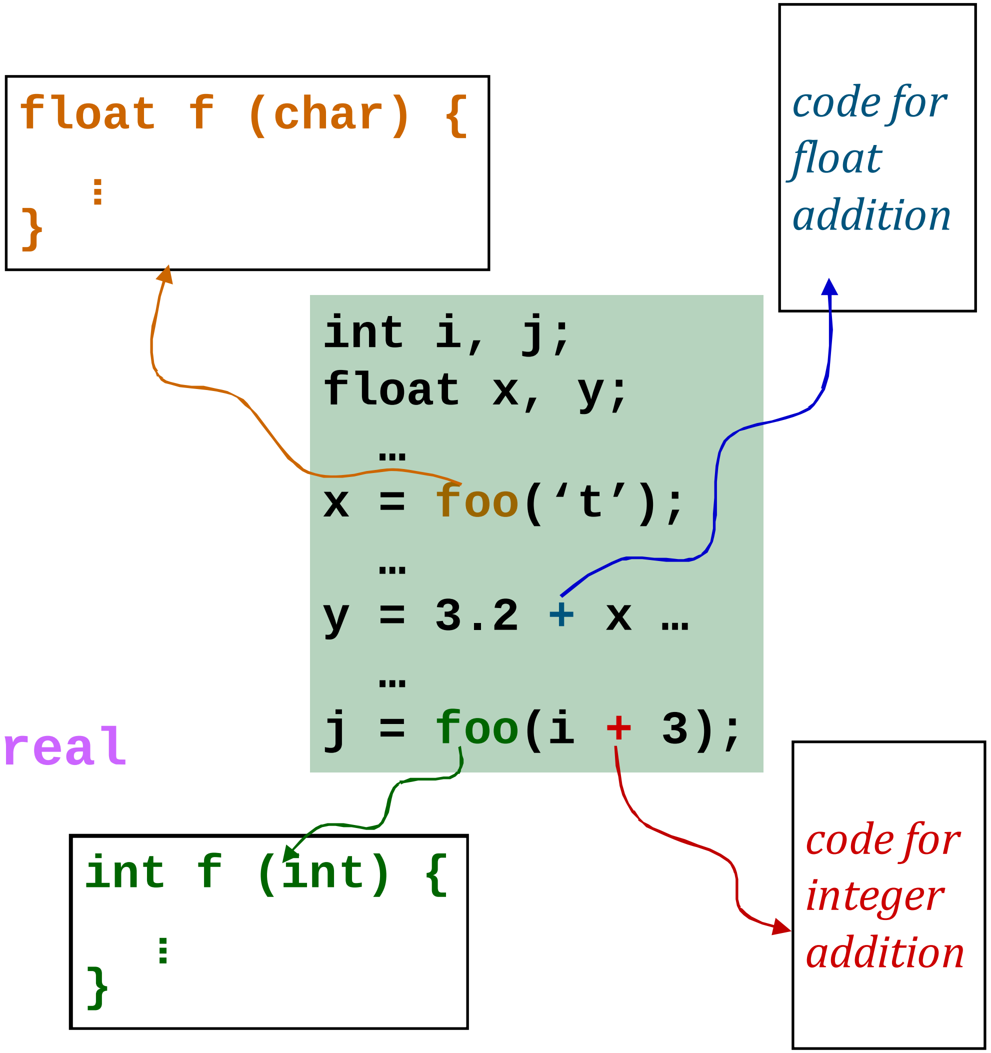
- A *monomorphic* object (function, variable, constant, …) has a single type.
  - constants of simple types (character/integer/real): ‘a’, 1, 2.34, …
  - variables of simple types: **`int i;`** (*C*), **`x :real;`** (*Pascal*)
  - various user-defined functions: **`int foo(char* c);`**
- A *polymorphic (generic)* object has more than one types.
  - the constant **`nil`** in Pascal and **`0`**(*integer,virtual function, pointer*) in C
  - basic operators for **`int`** and **`float`**: **`+, –, :=, ==, <, >, /, *`**(*multiply, dereference*), …
  - functions with the same name but with different argument lists: **`foo(int i)`**, **`foo(char c, int x)`**
  - subtype objects
    - subrange types
    - a base class pointer referencing its derived class objects in OOP
    - virtual functions in OOP

Polymorphic functions

1. **Ad-hoc** polymorphic functions that work on a finite number of types
   - *overloaded* functions
     - built-in → `+`, `*`, …
     - user-defined

       ```
       int foo(int i);
       float foo(char c);
       ```
   - functions with parameter *coercion*

     Ex: convert `real + int` to `real + real`
   - After the ambiguity is resolved, a *different piece of code* is used.

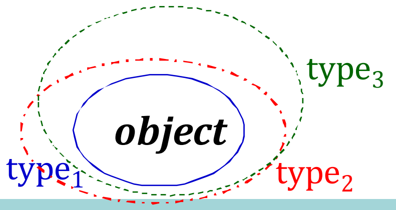


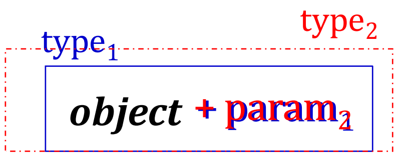
```
float f (char) {
   ⋮
}
```

```
int i, j;
float x, y;
  …
x = foo('t');
  …
y = 3.2 + x …
  …
j = foo(i + 3);
```

```
int f (int) {
   ⋮
}
```

Polymorphic functions





2. **Universal** polymorphic functions that work on an unlimited numbers of types
   - **inclusion** polymorphism: an object can be viewed as belonging to many different classes that need not be disjoint; that is, there may be inclusion of classes. → *subtypes, derived classes in C++/Java*
   - **parametric** polymorphism: a polymorphic function has an implicit or explicit type parameter which determines the type of the argument for each application of that function. → ***** (*dereference*), template in C++.

     Cf: Java does not support parametric polymorphism
   - Typically, the *same code* is used regardless of the types of the parameters, and the functions exploit a *common structure* among different types.
     - The template function `foo<>()` defines the common code to be used with the template parameter `T`.
     - The dereference operator ***** returns the value stored at the address where its input argument points.

```
template <class T>
foo(T& x) {
  …
};
```

Parametric polymorphism in C++

```
class SNU {
  public: void f() { … }
};
class EE {
  public: void f() { … }
};

template <class T>
void foo(T& x) {
  …
  x.f();
  …
};
```

Original Code

```
main() {
  SNU s;
  EE e;
  foo(s);
  foo(e);
}
```

The template function foo takes as its parameters not only variable x but also type T.

➔ *Here, T is called a type variable.*

1999, U. Eisenecker

Inclusion polymorphism in C++

```
class SNU {
  public: void f() { … }
};
class EE : public SNU {
  public: void f() { … }
};

void foo(SNU* x) {
  …
  x->f();
  …
};
```

```
main() {
  SNU s;
  EE e;
  foo(&s);
  foo(&e);
}
```

1999, U. Eisenecker

Overloading in C++

```
class SNU {
  public: void f() { … }
};
class EE {
  public: void f() { … }
};

void foo(SNU& x) {
  …
  x.f();
  …
};
void foo(EE& x) {
  …
  x.f();
  …
};
```

```
main() {
  SNU s;
  EE e;
  foo(s);
  foo(e);
}
```

1999, U. Eisenecker

Coercion in C++

```
class SNU {
  public: void f() { … }
};
class EE {
  public: void f() { … }
          operator SNU() { }
};

void foo(const SNU& x) {
  …
  x.f();
  …
};
```

// type conversion operator

Without this, we will have an error!

```
main() {
  SNU s;
  EE e;
  foo(s);
  foo(e);
}
```

e changes to **s**, so this code is not equivalent to the original code ←

Coercion in C++

```
class EE {
  public: void f() { … }
};
class SNU {
  public: SNU(EE&) { }               // type conversion constructor
          SNU() { }                  // ordinary constructor
          void f() { … }
          operator SNU() { }
};

void foo(const SNU& x) {
  …
  x.f();
  …
};
```

```
main() {
  SNU s;
  EE e;
  foo(s);
  foo(e);
}
```

`e` changes to **`s`**, so this code is not equivalent to the original code ←