# Data Type

## COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

# Outline

# Type

**Type** A collection of values.

- $\mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\}$
- $\mathbb{B} = \{\mathit{FALSE}, \mathit{TRUE}\}$
- $\mathbb{R}$ = set of real numbers
- 2D points = $\mathbb{R}^2$
- Suit = {SPADE, HEART, DIAMOND, CLUB}
- Card = {A, 2...10, J, Q, K}$\times$Suit

# Data Type

**Data Type** A *type* together with *operations*. Almost any noun can give rise to a data type, e.g., integer, string, complex number, person, vector, stack, queue, deque, graph.

Example data types

- Boolean supports AND, OR, XOR, NOT.
- Integer supports $+$, -, $\times$, /, ceil, floor, abs, comparison, etc.
- Stack supports push, pop, top, size, empty, etc.
- String supports concat, sub string, append, etc.

# Data Structure

**Data Structure** The actual implementation of a data type. It details the algorithms underlying the operations.

- Linear structures.
  - Array, e.g., matrix, vector, sparse array.
  - List, e.g., doubly linked list, skip list, dynamic list.
- Tree.
  - Binary tree, e.g., AVL tree, RB tree, order statistic tree.
  - B-tree, e.g., B-tree, B+-tree.
  - Heap, binary heap, 2-3 heap.
  - Multiway tree, e.g., K-ary tree, ternary tree, disjoint set.
- Hash, e.g., hash table, hash tree.
- Graphs, e.g., adjacency list, adjacency matrix, directed graph.
- etc.

# Outline

# Primitive Data Type

**Primitive Data Type** Data types that are not defined in terms of other types.

- Numeric Types
  - Integer
  - Floating point
  - Complex
  - Decimal
- Boolean Type
- Character Type

# C++ Primitive – Integer

`int` basic integer type. Guaranteed to have at least 16 bit. On 32/64 bit systems, however, it is guaranteed to be at least 32 bit.

Signedness

- `signed` (default) target type will have signed representation.
- `unsigned` target type will have unsigned representation.

Size

- `short` at least 16 bit.
- `long` at least 32 bit.
- `long long` at least 64 bit. (since `C++11`).

# C++ Primitive – Floating Number

`float` single precision, IEEE-754 32 bit type.

`double` double precision, IEEE-754 64 bit type.

`long double` extended precision, usually 80 bit x87 type.

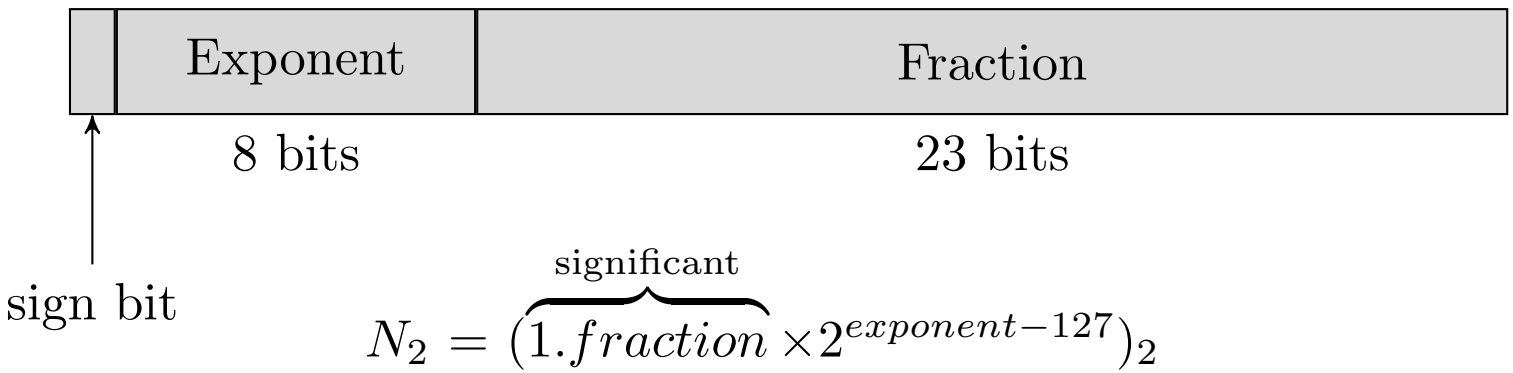



Figure: IEEE 754 32 bit float type

# Range and Precision

**Precision** The minimum difference between two successive significant representations; thus it is a function of the number of significant digits.

**Range** The combination of the range of fractions and the range of exponents.

**Gap** The difference between two successive numbers.

# C++ Special Floating Numbers

- Zero
  - sign 0 for positive zero and 1 for negative zero
  - exponent all 0
  - fraction all 0
- Infinity
  - sign 0 for positive infinity and 1 for negative infinity
  - exponent all 1
  - fraction all 0
- NaN, Not a Number
  - sign *either* 0 or 1
  - exponent all 1
  - fraction anything *except* all 0

# C++ Non-Numbers Example

```cpp
#include <limits>                    // for numeric_limits
#include <cmath>                     // for isinf, isnan
#include <iostream>                  // for cout, endl
using namespace std;

int main() {
  cout << isinf(numeric_limits<float>::infinity()) << endl
       << isnan(numeric_limits<float>::signaling_NaN()) << endl;
}
```

# Boolean

PL without an explicit Boolean data type may represent with other data types. E.g., in `C90`, `0` is false and not zero is true; in `Common Lisp`, empty list is false.

Table: Common Boolean algebraic operations

| Operation | Keywords | Operator |
|---|---|---|
| Conjunction | `AND` | `& *` |
| Disjunction | `OR` | `\| +` |
| Equivalence | `EQV` | `= ==` |
| Exclusive OR | `XOR` | `^ !=` |
| Negation | `NOT` | `~ !` |

# C++ Primitive – bool

C++ support native `bool` type, with two values `false` and `true`.

Bit-wise operations AND `&`, OR `|`, XOR `^` and NOT `~`.

Logic operation AND `&&`, OR `||` and NOT `!`.

```
int main() {
  bool a = true;
  bool b = !a;
  int a = 3 & 4;
  int b = 3 && 4;
}
```

# Character Type

Character data are stored in computers as *numeric coding*.

- ASCII is the most commonly used 8-bit coding.
- UTF-8 (U-niversal Coded Character Set T-ransformation F-ormat 8-bit) is a character encoding capable of encoding all possible characters, or code points, in Unicode.
  - The encoding is variable-length, and
  - uses 8-bit code units.
  - is designed for backward compatibility with ASCII,

# UTF-8 Takes the World

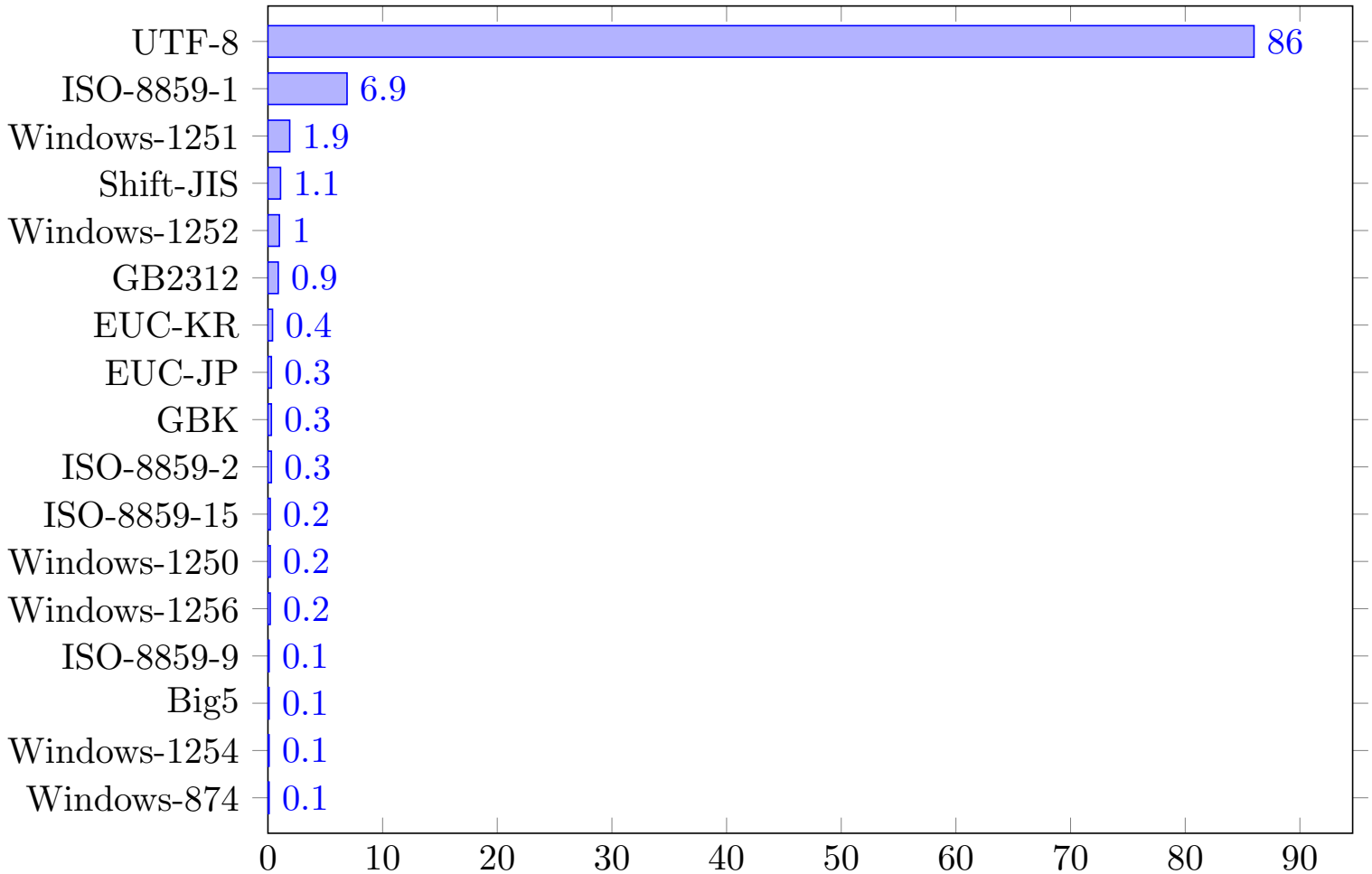



Figure: Character Coding Usage 2015 (by W$^3$Techs)

# C++ Character Type

`char` Type for character representation which can be most efficiently processed on the target system. Since C++14, The character types are large enough to represent 256 different values (in order to be suitable for storing UTF-8 encoded data)

`wchar_t` Type for wide character representation.

---

```cpp
#include <iostream>
using namespace std;

int main() {
  cout << u8"\u0391" << endl;
}
```

---

# Outline

# Array

**Array** A *homogeneous* aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element. References to individual array elements are specified using *subscript* expressions. Operations may include assignment, concatenation, slices, comparison, indexing and etc.

```cpp
#include <iostream>
using namespace std;

int main() {
  int a[3] = {1, 2, 3};
  cout << a[0] << a[1] << a[2] << endl;
}
```

# Array Design Issue

What types are legal for subscripts? E.g., in C++, any type that can be implicitly cask to `int` may be used as indices. Negative indices is invalid in C++. In Python however, it means "count starts from the end".

```cpp
#include <iostream>
using namespace std;

int main() {
  int a[3] = {1, 2, 3};
  int i0 = 2;           cout << a[i0] << endl;
  bool i1 = false;      cout << a[i1] << endl;
  char i2 = '\2';       cout << a[i2] << endl;
  float i3 = 1.43;      cout << a[i3] << endl; // WRONG
}
```

# Array Design Issue Cont'd

Are indexing ranges checked? E.g., in C++, no. The follow is legal albeit undefined. In Python, however, `IndexError: list index out of range` will occur.

```
#include <iostream>
using namespace std;

int main() {
  int a[3] = {1, 2, 3};
  cout << a[5] << endl;
}
```

# Array Design Issue Cont'd

Are jagged or rectangular multi-dimensional array supported?

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  int** a = new int* [2];
  a[0] = new int [4];
  a[1] = new int [2];

  vector<vector<int> > b = {{1, 2, 3}, {4, 5}};

  delete[] a[0];
  delete[] a[1];
  delete[] a;
}
```

# Array Design Issue Cont'd

Array slices supported? In C++, by default, no slicing syntax is supported. Python supports basic slicing.

```
a = range(5)
a[1:3]                          # 1, 2
a[-1]                           # 4
a[3:]                           # 3, 4
a[1:-1]                         # 1, 2, 3
```

# Array Design Issue Cont'd

Consider the following issues.

1. When are subscript ranges bound?
2. When does array allocation take place?
3. Where is the storage allocated?

Based on different design, there are five categories of array.

# Static Array

- Subscript ranges are statically bound, and
- storage allocation is static, i.e., done before runtime.

```
int main() {
  static int arr[3] = {1, 2, 3};
}
```

Advantage is efficient, no dynamic allocation.

# Fixed Stack-Dynamic Array

- Subscript ranges are statically bound, but
- the allocation is done at declaration elaboration time during execution.

```
void f1() { int arr[3] = {1, 2, 3}; }
void f2() { int arr[3] = {1, 3, 2}; }

int main() {
  f1();
  f2();
}
```

- Advantage is space efficiency.
- Disadvantage is allocation and deallocation overhead.

# Stack-Dynamic Array

- Subscript ranges are dynamically bound, and
- Storage allocation are also dynamically bound at elaboration time.

Since C++14, it is implemented as *runtime-sized array*.

```
int main() {
  int p = 3;
  int a[p];
}
```

Advantage is flexibility.

# Fixed Heap-Dynamic Array

- Subscript ranges are fixed, and
- Storage binding is also fixed.

Differences from fixed stack-dynamic array are that

1. both the subscript ranges and storage bindings are done when the user program requests them during execution (flexible albeit slower), and
2. storage is allocated from the heap, rather than the stack.

```
int main() {
  int* a = new int [3];
  delete[] a;
}
```

# Heap-Dynamic Array

- Subscript ranges are dynamic,
- storage allocation is also dynamic, and
- both may change any number of times.

```
int main() {
  int size = 3;
  int* heap_dynamic_array = new int[size];
  delete[] heap_dynamic_array;
}
```

# Initialization

It is a language-dependent feature. The following is C++ snippet. The last two show a new feature called *uniform initialization*.

```
int main() {
  int a[3] = {1, 2, 3};
  int b[3]{1, 2, 3};              // C++11
  int* c = new int[3]{1, 2, 3}  // C++11
}
```

# Heterogeneous Array

**Heterogeneous Array** The elements need not be of the same type.

- C++ natively does not support, however, `Boost.Any` provides excellent work around.
- `Python`, `Javascript` and etc, supports this kind of array.

```
a = [12, 3.5, -1, 'two']
# a = [PyIntObject, PyFloatObject, PyIntObject, PyStringObject]
# a = [PyObject, PyObject, PyObject, PyObject]
```

# 1D Array in Memory

1D Array occupies contiguous block of memory with equal amount of space for each element. $k^{th}$ element is access based on offset from the the first.

$$addr_k = add_0 + k \times \text{element\_size}$$

```
#include <iostream>
using namespace std;
int main() {
  int a[3] = {1, 2, 3}
  cout << *a                    // a[0]
       << *(a + 1)              // a[1]
       << *(a + 2) << endl;     // a[2]
}
```

# Multi-Dimensional Array

Two way to layout the memory

- Single block of contiguous memory.
  - Arrays must be rectangular
  - Address of array is starting memory location
- Array of arrays
  - Supports jagged array
  - Array variable is a pointer

# 2D Array – Single Block

In case of *true multi-dimensional* array, there are two ways to layout 2D array in memory, *row major order* and *column major order*. E.g., given a 2D matrix

$$\begin{matrix} 3 & 4 & 7 \\ 6 & 2 & 5 \\ 1 & 3 & 8 \end{matrix}$$

- Row major order: 3 4 7, 6 2 5, 1 3 8
- Column major order: 3 6 1, 4 2 3, 7 5 8

# C++ – Row Major Order

C++ implements the row major ordering.

```cpp
#include <iostream>
using namespace std;

int main() {
  int c[3][2]{0, 1, 2, 3, 4, 5};
  for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 2; ++j)
      cout << c[i][j] << ' ';
  // output: 0 1 2 3 4 5
}
```

# Compile-Time Descriptor

| Array |
| --- |
| Element type |
| Index type |
| Index lower bound |
| Index upper bound |
| Address |

| Multidimensional Array |
| --- |
| Element type |
| Index type |
| Number of dimension |
| Index range 0 |
| $\cdots$ |
| Index range $n-1$ |
| Address |

# Outline

# Associative Array

**Associative array** An *unordered* collection of data elements that are indexed by an equal number of values called *keys*. User-defined keys need to be stored with data.

C++ does not provide native support for this data type. However, STL provides convenient containers, `unordered_map` and `map`.

```cpp
#include <unordered_map>
using namespace std;

int main() {
  unordered_map<int, int> um;
  um[0] = 3;
  um[33] = 4;
}
```

# Outline

# Record Type

**Record** An aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure.

Common operations on record type include declaring, referencing fields within record.

# Record Definition

Difference with array

- record elements, or *fields*, are referenced by identifies, and
- record may include *unions*.

In C++, `struct` is almost the same as `class`.

```
struct point {
  int x;
  int y;
};

int main() {
  point p{1,2};
}
```

# Reference to Record Fields

Fully qualified reference

$$\overbrace{recordName0.recordName1 \ldots recordNameN}^{\text{enclosing records}} . \overbrace{fieldName}^{\text{desired field}}$$

Elliptical references

$$\overbrace{recordName0\cancel{.recordName1 \ldots recordNameN}}^{\text{enclosing records}} . \overbrace{fieldName}^{\text{desired field}}$$

- Elliptical reference is a "convenient?" syntactic sugar
- Compiler need elaborate data structures and procedures to correctly identify the referenced field.
- They are also somewhat detrimental to readability.

# C++ Record Reference

C++ uses the *dot notation*.

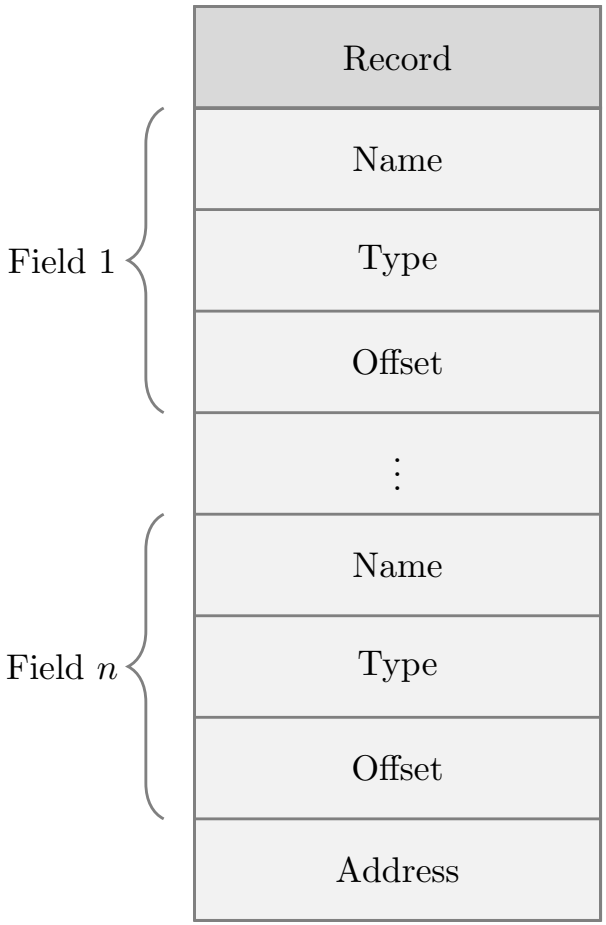
```
struct point {
  int x, y;
};
struct rect {
  point bottom_left, upper_right;
};

int main() {
  rect r;
  r.bottom_left.x = 3;
  r.upper_right.y = 10;
}
```

# R Record Reference

R uses *dollar notation*, similar to dot notation.

```
point <- function(x, y)
    structure(list(x = x, y = y),
                class = "point")
rect <- function(x0 = 0, y0 = 0,
                 x1 = 1, y1 = 1)
    structure(list(bottom_left = point(x0, y0),
                   upper_right = point(x1, y1),
                   class = "rect"))
a <- rect()
a$bottom_left$x
```

# Compile-Time Descriptor




- The fields of records are stored in adjacent memory locations.
- Fields are handled with *offset*.
- Runtime descriptor is unnecessary.

# Outline

# Pointer

**Pointer** It has a range of values that consists of memory addresses and a special value, `nil`, which is an invalid address used to indicate that a pointer cannot currently be used to reference a memory cell.

- Pointer is not structured type, like array and record.
- Pointer is not *value type*, like scalar variables. Rather, it is *reference type* which references other variable than store values themselves.

# Indirect Addressing

Pointer provides the powerful indirect addressing used in Assembly.

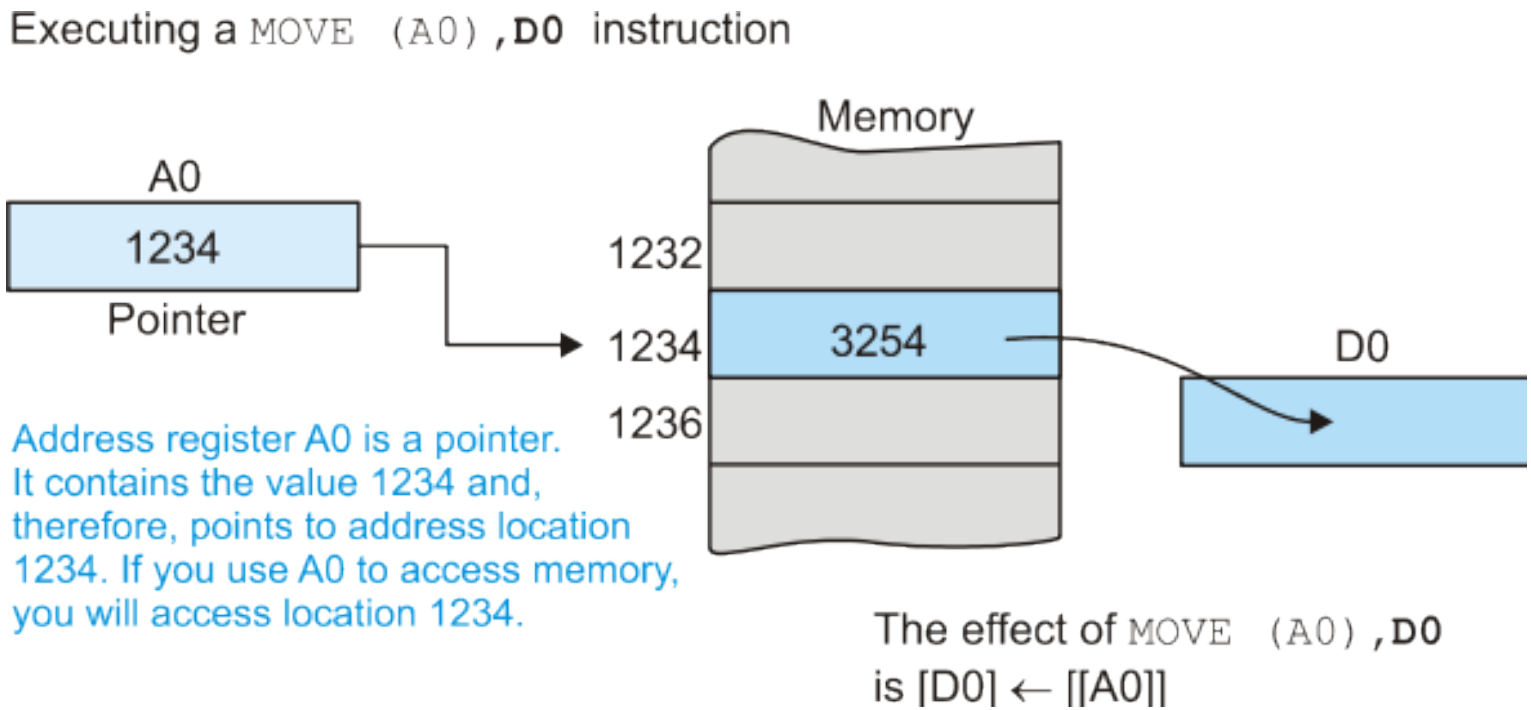



Figure: Indirect Addressing in Assembly (by Alan Clements)

# Heap Access

Pointer also provides a way to manage dynamic storage on *heap*.

Heap Dynamic Variables Variables that are dynamically allocated from the heap. They are often *anonymous variables* (no identifiers associated with them) and thus can be referenced only by pointer or reference type variables.

# Pointer Operations

Two fundamental pointer operations are needed

**Assignment** set a pointer variable's value to some useful address.

**Dereference** Takes a reference through one level of indirection, i.e., get the value referenced.

# Pointer Assignment

1. A pointer may be initialized through the allocation mechanism, whether by operator or built-in subprogram

```
int main() {
  int* a = new int{3};            delete a;
  char* b = (char*) malloc(3);    free(b);
}
```

2. In the case of indirect addressing, an explicit operator or built-in function is needed to get the address of variables not on heap.

```
int main() {
  int a = 3;                      // on stack, not heap
  int* b = &a;                    // & to get address of a
}
```

# Pointer Dereference

When a pointer appears in an expression, it may be interpreted as

- a normal scalar value, or
- a value to which is points to, i.e., indirect addressing resulted by dereferencing the pointer..

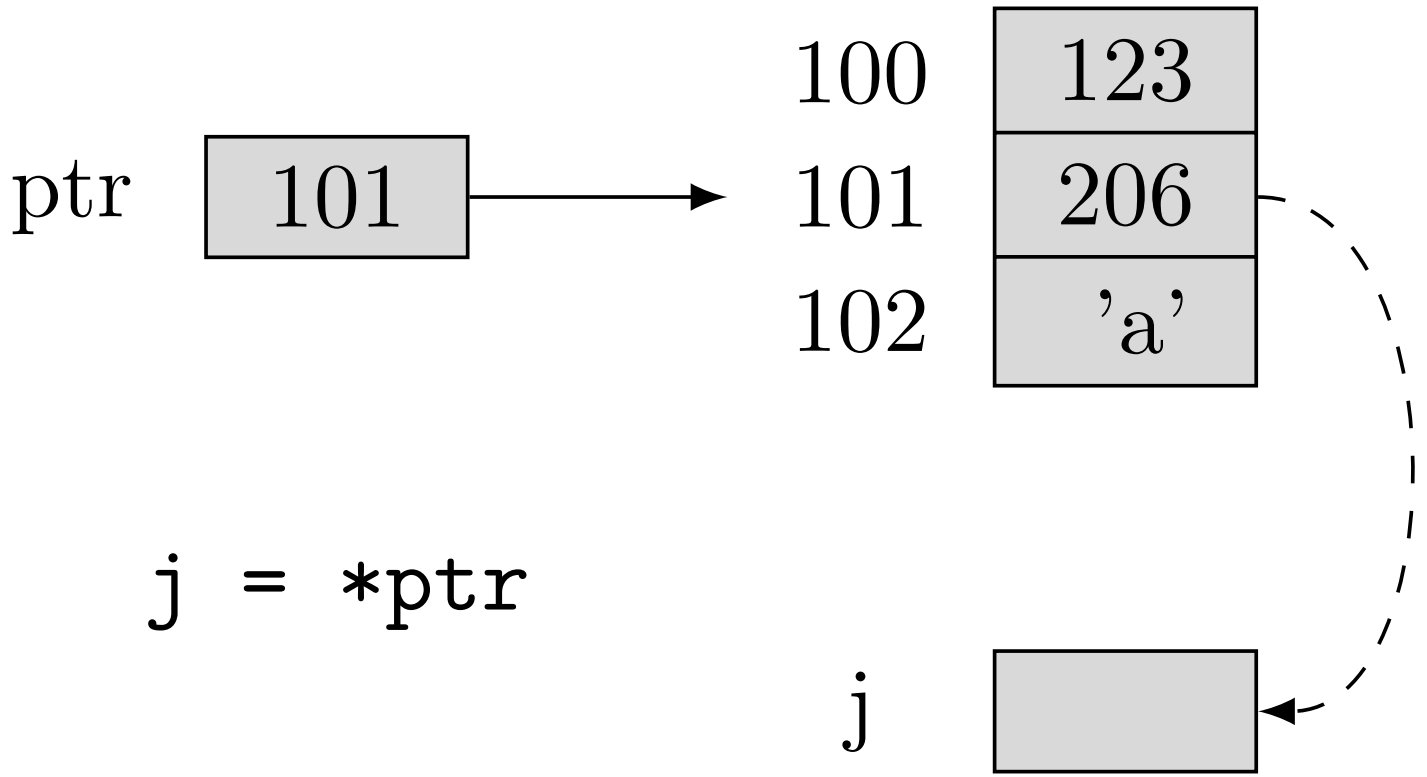



Figure: Pointer Dereferencing

# Pointer to Record

The syntax may vary. The following is the syntax for C++.

```
struct point {
  int x, y;
};

int main() {
  point* p = new point;
  (*p).x = 3;
  p->y = 4;
  delete p;
}
```

# Pointer Headache

**Dangling Pointer** Contains the address of a heap-dynamic variable that has been deallocated.

```
int main() {
  int* a = new int{3};
  int* b = a;
  delete a;                    // now b is dangling
  delete b;                    // ERROR! Double free!
}
```

# Pointer Headache Cont'd

**Memory Leakage** An allocated heap-dynamic variable that is no longer accessible to the user program.

They are are called *garbage*

- they are not useful for their original purpose, and
- they also cannot be reallocated for some new use in the program.

```
int main() {
  int* a = new int{3};
  a = new int{4};                    // the old memory is lost
}
```

# Pointer Painkiller – Tombstone

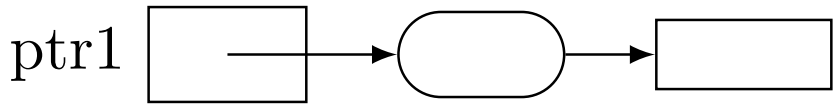



Figure: new(ptr1)

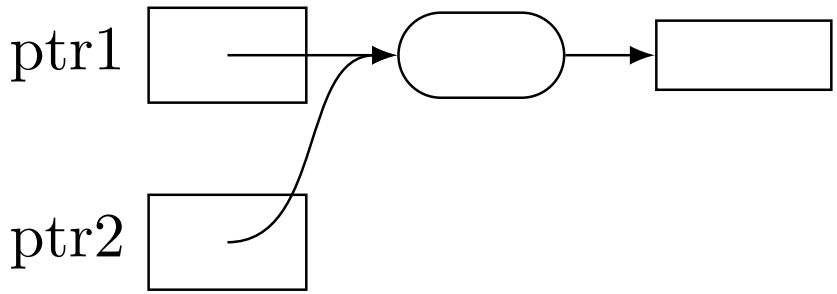



Figure: ptr2 = ptr2

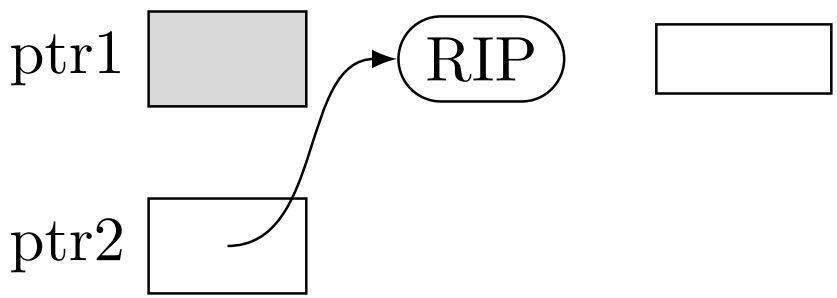



Figure: delete(ptr1)

# Tombstone Issues

- For heap objects, it is easy to invalidate a tombstone when the program calls the deallocation operation.
- For stack objects, the program must be able to find all tombstones associated with objects in the current stack frame when returning from a subroutine.

Tombstone is expensive, both in time and in space. Time overhead includes

- creation of tombstones,
- checking for validity on every access, and
- double indirection.

Tombstone is easy for *storage compaction* because of double indirection.

# C++ Pointer Wrapper

- For small programs, it is OK and to stay with native pointer.
- For large programs, use `shared_ptr`, `unique_ptr` and etc. declared in `<memory>` from C++ STL.

```cpp
#include <memory>
using namespace std;

int main() {
  shared_ptr<int> a{new int{3}};
  shared_ptr<int> b = a;
  b = nullptr;
}
```

# Reference

**Reference type** Similar to a pointer, with one fundamental difference: a pointer refers to an address in memory, while a reference refers to an object or a value in memory.

Reference type in C++ has the following features

- It is an *implicitly dereferenced constant* pointer.
- Must be defined when declared since it is constant.
- May not be set to reference other variables since it is implicitly dereferenced.

# Outline

# Type Checking

**Type Checking** The activity of ensuring that the operands of an operator are of *compatible types*.

**Compatible Type** A compatible type is one that either is legal for the operator or is allowed under language rules to be implicitly converted (coercion) by compiler-generated code (or interpreter) to a legal type.

**Type Error** The application of an operator to an operand of an inappropriate type.

---

```
int main() {
  int a = 1.2;
  float b = 1.2 + a;
  int* c = a;                    // WRONG!!
}
```

---

# Strong Typing

`C` is called both *strong typed* and *weak typed* by different authors.

- An example for a statically, but weakly typed language is C. (link)
- LISP engines . . . are themselves generally written in strongly typed languages like C. . . (link).
- In a weakly typed language such as C. . . (link)

# Static and Dynamic Typing

**Static Typed** Variable Type is known at compile time.

**Dynamic Typed** Variable Type is known at runtime.

## Static Typing

```
#include <string>
#include <iostream>

void silly(int a) {
  if (a > 0) std::cout << "Hi";
  else std::cout << 5.3 + "3";
}
int main() { }
```

## Dynamic Typing

```
def silly(a):
    if a > 0:
        print 'Hi'
    else:
        print 5.3 + '3'

silly(1)     # PASSES
silly(-1)    # WRONG
```

# Type Equivalence

**Name Equivalence** Two variables have equivalent types if they are defined either in the same declaration or in declarations that use the same type name, i.e., they can be represented by the same syntax tree with the same labels.

**Structural Equivalence** Two variables have equivalent types if their types have identical structures, i.e., replace the named types by their definitions and recursively check the substituted syntax tree.

# Structure Equivalence Issue

- What parts constitute a structural difference?
  - Storage: record fields, array size
  - Naming of storage: field names, array indices
  - Field order

```
struct point1 { int x, y; };
struct point2 { int a, b; };
int a[5];
int b[5];
```

- Intentional vs incidental structural similarities.

# Name Equivalence Issue

Argument: They're different because the programmer said so; if they're the same, then the programmer won't define two types!! So what about *alias*?

```
struct point1 { int x, y; };
typedef point1 point2;
```

# C++ Type Equivalence

Both name and structure equivalence exist.

- Name equivalence is used for `struct`, `enum` and `union`.
- Structure equivalence is used for other *non-scalar* types.

```
void foo(int a[]) { }

int main() {
  int a[5];
  int b[10];
  int* c = new int [3];
  foo(a); foo(b); foo(c);
  delete[] c;
}
```

# Outline

# Summary

Data type influences PL's usefulness and convenience.

- Most PL include primitive data types, i.e., numeric, character, and Boolean.
- Array (or equivalent) is essential in most PL's.
- Most PL provides user-defined types, e.g., record, for data abstraction. Discussed further in OOP.