Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University
2010
Chapter 2. Data Types
Introduction

Goal: Hierarchy of data types. Building new data types from atomic data types.
Goal: Hierarchy of data types. Building new data types from atomic data types.

Mechanisms for creating new types:

- Pointers: Memory address of a value (may be an address itself).
- Arrays: Collection of data values of the same type. Accessed by indices.
- Records: Collection of data values (may be of different types). Identified by names.
Outline

1. Enumeration Types
2. Data and Memory
3. Pointers
4. Arrays
5. Pointers and Arrays
6. Records
Another atomic type defined by listing the elements in its domain.

Example. Definition

```c
enum directionT {North, East, South, West}
```

North, East, ...: Enumeration constants
Another atomic type defined by listing the elements in its domain.

Example. Definition

```c
enum directionT {North, East, South, West}
```

North, East, ...: Enumeration constants

variable declaration

directionT dir;
Assigning integers to enumeration constants:

**Automatic**

North = 0, East = 1, ...
Assigning integers to enumeration constants:

Automatic
North = 0, East = 1, ...

manual

enum coinT {
    Penny = 1,
    Nickel = 5,
    Dime = 10,
    Quarter = 25
};
semi-automatic

```c
enum monthT {
    January = 1, February, March, April, May, June,
    July, August, September, October, November, December
};
```
You can perform integer operations on values of an enumeration type

Example

directionT RightFrom(directionT dir) {
    return directionT((dir + 1) % 4);
}
semi-automatic

enum monthT { 
    January = 1, February, March, April, May, June, 
    July, August, September, October, November, December 
};

You can perform integer operations on values of an enumeration type

Example

directionT RightFrom(directionT dir) { 
    return directionT((dir + 1) % 4);
}

A general type class: scalar types (enumeration types, 
characters, and various representations of integers).
Implicit conversion from a value of a scalar type into an integer.
Outline

1. Enumeration Types
2. Data and Memory
3. Pointers
4. Arrays
5. Pointers and Arrays
6. Records
Memory units:
bit (smallest)
byte (typically 8 bits, size of char)
word (size of int, 2 bytes or 4 bytes or others)
Memory units:
bit (smallest)
byte (typically 8 bits, size of `char`)
word (size of `int`, 2 bytes or 4 bytes or others)

Memory addresses: Byte addressable, starting from 0
Memory units:
bit (smallest)
byte (typically 8 bits, size of char)
word (size of int, 2 bytes or 4 bytes or others)

Memory addresses: Byte addressable, starting from 0

sizeof operator usage:
sizeof(int) sizeof x

returns the number of bytes.
Example. Memory allocation

```c
char ch;
ch = 'A';
```
Example (cont.)

```c
int i;
i = 123;
```

![Diagram showing memory address and variable i pointing to 123]
Pointer: An address in memory, typically four bytes, for memory of size up to 4GB.
Pointers

Pointer: An address in memory, typically four bytes, for memory of size up to 4GB.

lvalue: An expression that refers to an internal memory location (can appear on the left side of an assignment).

lvalues: simple variables, \( x = 1.0 \)

not lvalues: constants, arithmetic expressions \((x + 1)\)
Pointers

Pointer variables

```c
int *p;
pointer-to-int, base type is int

char *cptr;
pointer-to-char, base type is char
```
Pointer variables

```c
int *p;
pointer-to-int, base type is int

char *cptr;
pointer-to-char, base type is char
```

Operator & (address-of)

```
&x
memory address in which x (lvalue) is stored.
```
Example. \* and &

```c
int x, y; (lvalues)
int *p1, *p2; (pointer-to-int)
```

```
1000  x
1004  y
1008  p1
1012  p2
```
Example. * and &

```c
int x, y; (lvalues)
int *p1, *p2; (pointer-to-int)
```

```
1000       x
1004       y
1008       p1
1012       p2
```

&x is 1000, &y is 1004
Example. `*` and `&`

```c
x = -42; y = 163;
p1 = &x; p2 = &y;
```
Example. ∗ and &

Dereferencing

\[ ∗p1 = 17 \]
Example. * and &

Pointer assignment and value assignment
\[ p1 = p2; \text{ and } *p1 = *p2; \]
null pointer **NULL**

A special value that does not point to any valid data.
null pointer **NULL**

A special value that does not point to any valid data.

Do not dereference a null pointer (do not use `*NULL`)
null pointer **NULL**

A special value that does not point to any valid data.

Do not dereference a null pointer
**(do not use \*NULL)**

Do not use pointer variables whose values have not yet been initialized.
Arrays

An array is characterized by

- element type;
- array size (number of elements).
Arrays

An array is characterized by

- element type;
- array size (number of elements).

Declaration

type name[size]
An array is characterized by
- element type;
- array size (number of elements).

Declaration

```
type name[size]
```

**style**

Define a constant for array size.
Example

const int N_JUDGES = 5;
double scores[N_JUDGES];

Element selection

scores[0] = 9.2;

array name and index
Passing arrays as parameters

Example.

```c
double Mean(double array[], int n) {
    double total = 0;

    for (int i = 0; i < n; i++) {
        total += array[i];
    }

    return total / n;
}
```
Passing arrays as parameters

Example.

double Mean(double array[], int n) {
    double total = 0;

    for (int i = 0; i < n; i++) {
        total += array[i];
    }

    return total / n;
}

- use empty brackets (a pointer to the array, elements can be modified);
- pass the effective size as a parameter.
Example: gymjudge.cpp, p. 61

/*
 * File: gymjudge.cpp
 * ------------------
 * This program averages a set of gymnastic scores.
 */

#include <iostream>
#include "genlib.h"
#include "simpio.h"
/* constants */
const int MAX_JUDGES = 100;
const double MIN_SCORE = 0.0;
const double MAX_SCORE = 10.0;

/* Private function prototypes */
void ReadAllScores(double scores[], int nJudges);
double GetScores(int judge);
double Mean(double array[], int n);
Example: gymjudge.cpp

```cpp
int main() {
    double scores[MAX_JUDGES];

    cout << "Enter number of judges: " << endl;
    int nJudges = GetInteger();

    if (nJudges > MAX_JUDGES) Error("Too many judges");

    ReadAllScores(scores, nJudges);

    cout << "The average score is " << Mean(scores, nJudges) << endl;

    return 0;
}
```
Example: gymjudge.cpp

```cpp
int main() {
    double scores[MAX_JUDGES];

    cout << "Enter number of judges: " << endl;
    int nJudges = GetInteger();

    if (nJudges > MAX_JUDGES) Error("Too many judges");

    ReadAllScores(scores, nJudges);

    cout << "The average score is " << Mean(scores, nJudges) << endl;

    return 0;
}
```

Remarks

- Basic structure: Declaration and initialization - input - compute - output;
- Robustness: Handle all possible inputs.
/*
 * Function: ReadAllScores
 ...
 */

void ReadAllScores(double scores[], int nJudges) {
    for (int i = 0; i < nJudges; i++) {
        scores[i] = GetScore(i + 1);
    }
}

Use empty brackets when passing an array as a parameter (pointer). Elements are modified.
Example: `gymjudge.cpp`

```cpp
/*
  * Function: GetScore
  ...
  */

double GetScore(int judge) {
  while (true) {
    cout << "Score for judge #" << judge << ": " << endl;
    double score = GetReal();
    if (score >= MIN_SCORE && score <= MAX_SCORE) return score;
    cout << "That score is out of range. Try again." << endl;
  }
}
```

- Robustness, bullet-proof your program;
- Loop-and-half structure.
Multidimensional arrays

Array of arrays.
Multidimensional arrays

Array of arrays.

Two-dimensional arrays for matrices (rectangle structure).

Example:

double mat[3][2]

An array of three arrays, each of which is an array of two floating-point numbers, representing a three-by-two matrix.
Multidimensional arrays (cont.)

Internal structure (row orientation)

\[
\begin{array}{c|c}
& \text{mat}[0][0] \\
& \text{mat}[0][1] \\
& \text{mat}[1][0] \\
& \text{mat}[1][1] \\
& \text{mat}[2][0] \\
& \text{mat}[2][1] \\
\end{array}
\]
Initializing arrays

double mat[3][2] = {
    { 1.0, 2.0 },
    { 2.0, 1.0 },
    { 3.0, 2.0 }
};

matrix:

\[
\begin{bmatrix}
1.0 & 2.0 \\
2.0 & 1.0 \\
3.0 & 2.0 \\
\end{bmatrix}
\]
In C++, it is more efficient to access elements in rows than in columns.

```c++
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        ... mat[i][j] ...
    }
}
```

is more efficient than

```c++
for (int j = 0; j < n; j++) {
    for (int i = 0; i < m; i++) {
        ... mat[i][j] ...
    }
}
```
Enumeration Types

Data and Memory

Pointers

Arrays

Pointers and Arrays

Records
int intList[5];

- intList is identical to &intList[0]
- &intList[i] is the same as intList + i*sizeof(int)
- the prototype
  int SumIntArray(int array[], int n)
  works the same way as
  int SumIntArray(int *array, int n)
- int intList[5] allocates five consecutive words, whereas int *p allocates one word for an address
- a pointer allows you to create a new array as the program runs (dynamic allocation, later)
Outline

1. Enumeration Types
2. Data and Memory
3. Pointers
4. Arrays
5. Pointers and Arrays
6. Records
Records

A coherent collection of components of possibly different types. Each of these components is called a field or member of the record.
A coherent collection of components of possibly different types. Each of these components is called a field or member of the record.

Defining a new structured type

1. Define a structure, including fields, names and types of the fields. This structure defines a model, but does not reserve any storage;

```c
struct employeeRecordT {
    string name;
    string title;
    string ssn;
    double salary;
    int withholding;
};
```

2. Declare variables of the new type.

```c
employeeRecordT empRec;
```
Records

Field selection

empRec.title (recordName.fieldName)

an lvalue
Field selection
empRec.title (recordName.fieldName)
an lvalue

Initializing records

empRec.name = "Ebenezer Scrooge";
empRec.title = ...

or

employeeRecordT empRec = {
    "Ebenezer Scrooge", ...
};
Often variables that hold structured data are declared to be pointers to records.

```c
employeeRecordT *empPtr;
```
Often variables that hold structured data are declared to be pointers to records.

```
employeeRecordT *empPtr;
```

**Field selection**

```
empPtr->salary means (*empPtr) . salary
```
Pointers to records

Often variables that hold structured data are declared to be pointers to records.

```c
employeeRecordT *empPtr;
```

Field selection

```c
empPtr->salary means (*empPtr).salary
```

**What does *empPtr.s salary mean?**
Allocation

- Static allocation: Global variables that persist throughout the entire program.
- Automatic allocation: Local variables inside a function, allocated on the system stack and freed when the function returns.
- Dynamic allocation: Variables created while the program is running, allocated on the heap, the pool of memory available to a program.
Example

employeeRecordT *empList = new employeeRecordT[1000];
Allocation

Example

employeeRecordT *empList = new employeeRecordT[1000];

Allocates an array of 1000 employee records in the heap and returns the pointer to the first record.
Deallocation

Coping with memory limitations.
Free pieces of memory when you are finished using them.

```cpp
double *dptr = new double;
int *arr = new int[45];
...
delete dptr;
delete[] arr;
```
Deallocation

Coping with memory limitations.
Free pieces of memory when you are finished using them.

double *dpotr = new double;
int *arr = new int[45];
    ...
delete dpotr;
delete[] arr;

Don’t worry about it for this course.
Examples

Declared arrays and dynamic arrays

double dblArray[10];

Memory is allocated automatically as part of declaration process. The elements are allocated as part of the frame for the function (on the stack). The size must be a constant.
**Examples**

Declared arrays and dynamic arrays

declare double `dblArray[10]`;

Memory is allocated automatically as part of declaration process. The elements are allocated as part of the frame for the function (on the stack). The size must be a constant.

declare `double *` `dblList`;

declare `dblList = new double[10]`;

Memory is not allocated until `new` is invoked. The elements are allocated on the heap. The size can be a variable.
Examples (cont.)

Dynamic array of \( n \) pointers to \texttt{employeeRecordT}

\begin{verbatim}
employeeRecordT **list;
list = new employeeRecordT*[n];
list[0] = new employeeRecordT;
\end{verbatim}