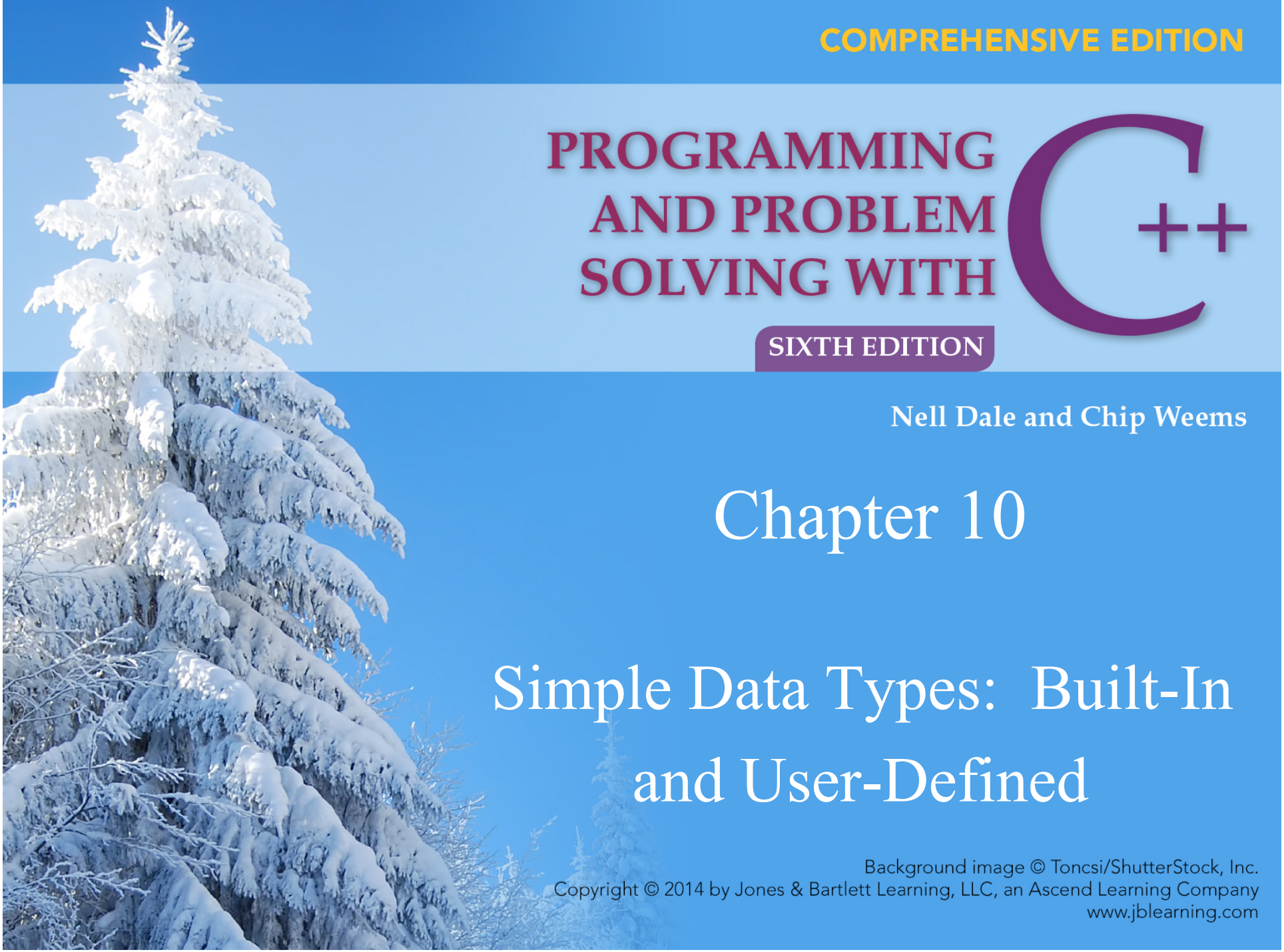


COMPREHENSIVE EDITION

PROGRAMMING AND PROBLEM SOLVING WITH C++

SIXTH EDITION

Nell Dale and Chip Weems

Chapter 10

Simple Data Types: Built-In and User-Defined

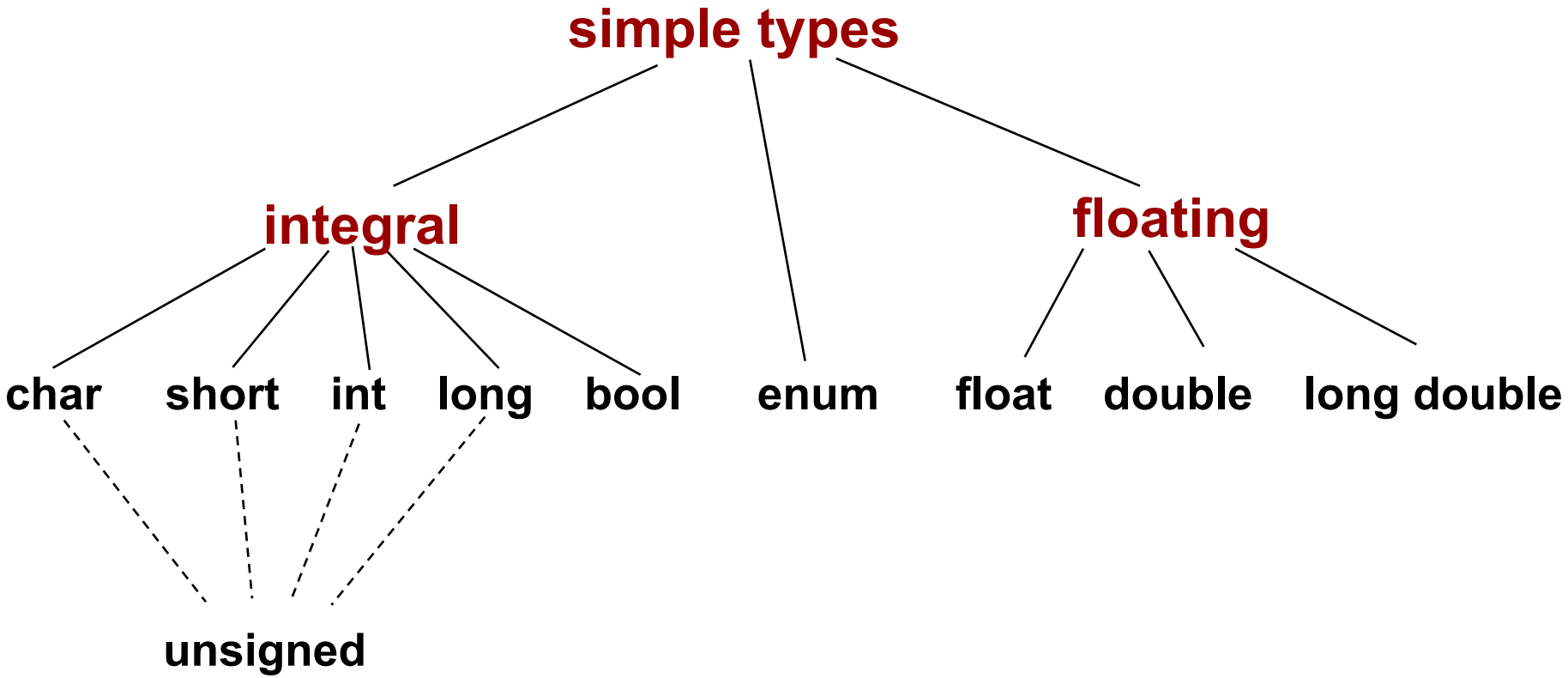
Chapter 10 Topics

- External and Internal Representations of Data
- Integral and Floating Point Data Types
- Using Combined Assignment Operators
- Using an Enumeration Type

Chapter 10 Topics

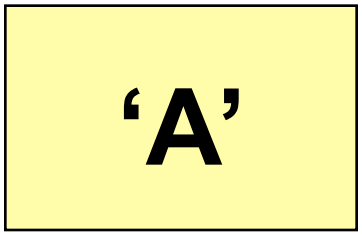
- **Creating and Including User-Written Header Files**
- **Meaning of a Structured Data Type**
- **Declaring and Using a `struct` Data Type**
- **C++ `union` Data Type**

C++ Simple Data Types



By definition,

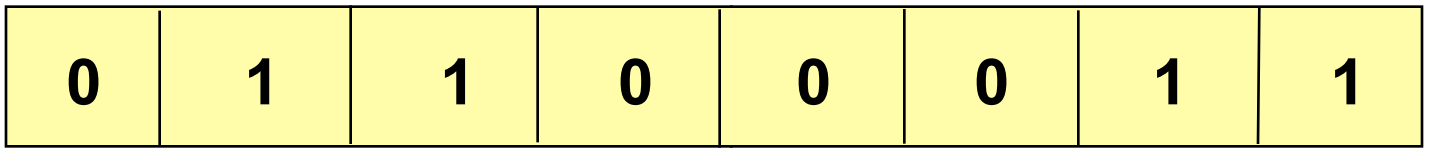
The size of a C++ char value is always 1 byte



exactly one byte of memory space

Sizes of other data type values in C++ are machine-dependent

Using one byte (= 8 bits)

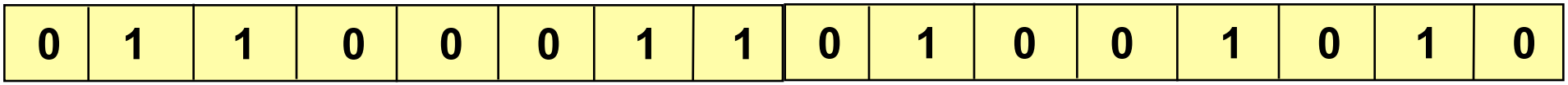


How many different numbers can be represented using 0's and 1's?

Each bit can hold either a 0 or a 1. So there are just two choices for each bit, and there are 8 bits.

$$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$$

Using two bytes (= 16 bits)



$$2^{16} = 65{,}536$$

So 65, 636 different numbers can be represented

If we wish to have only one number representing the integer zero, and half of the remaining numbers positive, and half negative, we can obtain the 65,536 numbers in the range -32,768 . . . . 0 32,767

Some Integral Types

Type	Size in Bytes	Minimum Value	Maximum Value
char	1	-128	127
short	2	-32,768	32,767
int	2	-32,768	32,767
long	4	-2,147,483,648	2,147,483,647

NOTE: Values given for one machine; actual sizes are machine-dependent

Data Type `bool`

- **Domain contains only 2 values, true and false**
- **Allowable operation are the logical (!, &&, ||) and relational operations**

Operator `sizeof`

`sizeof` A C++ unary operator that yields the size on your machine, in bytes, of its single operand. The operand can be a variable name, or it can be the name of a data type enclosed in parentheses.

```
int  age;
cout  <<  "Size in bytes of variable age is "
      <<  sizeof  age  <<  end;
cout  <<  "Size in bytes of type float is "
      <<  sizeof (float) <<  endl;
```

The only guarantees made by C++ are . . .

1 = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)

1 <= sizeof (bool) <= sizeof (long)

sizeof (float) <= sizeof (double) <= sizeof (long double)

. . . and the following three other C ++ guarantees

char is at least 8 bits

short is at least 16 bits

long is at least 32 bits

Exponential (Scientific) Notation

2.7E4 means 2.7×10^{4} =

2.7000 =

27000.0

2.7E-4 means 2.7×10^{-4} =

0002.7 =

0.00027

Floating Point Types

Type	Size in Bytes	Minimum Positive Value	Maximum Positive Value
float	4	3.4E-38	3.4E+38
double	8	1.7E-308	1.7E+308
long double	10	3.4E-4932	1.1E+4932

NOTE: Values given for one machine; actual sizes are machine-dependent

More about Floating Point Types

- Floating point constants in C++ like 94.6 without a suffix are of type double by default
- To obtain another floating point type constant a suffix must be used
  - The suffix F or f denotes float type, as in 94.6F
  - The suffix L or l denotes long double, as in 94.6L

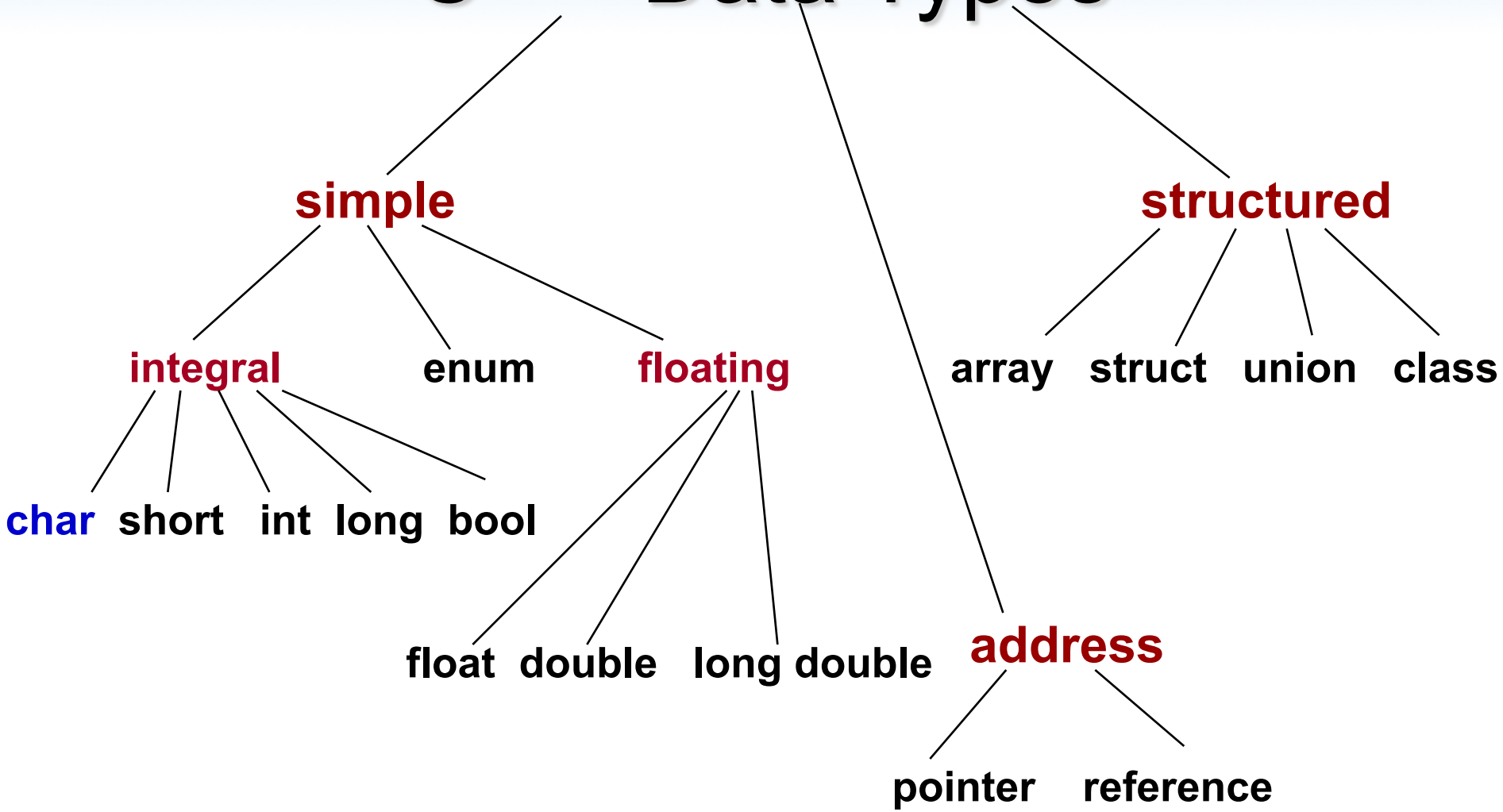
Header Files
climits and cfloat

- Contain constants whose values are the maximum and minimum for your machine
- Such constants are FLT_MAX, FLT_MIN, LONG_MAX, LONG_MIN

Header Files climits and cfloat

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


cout  << "Maximum long is "  << LONG_MAX
      << endl;
cout  << "Minimum long is "  << LONG_MIN
      << endl;
```

C++ Data Types



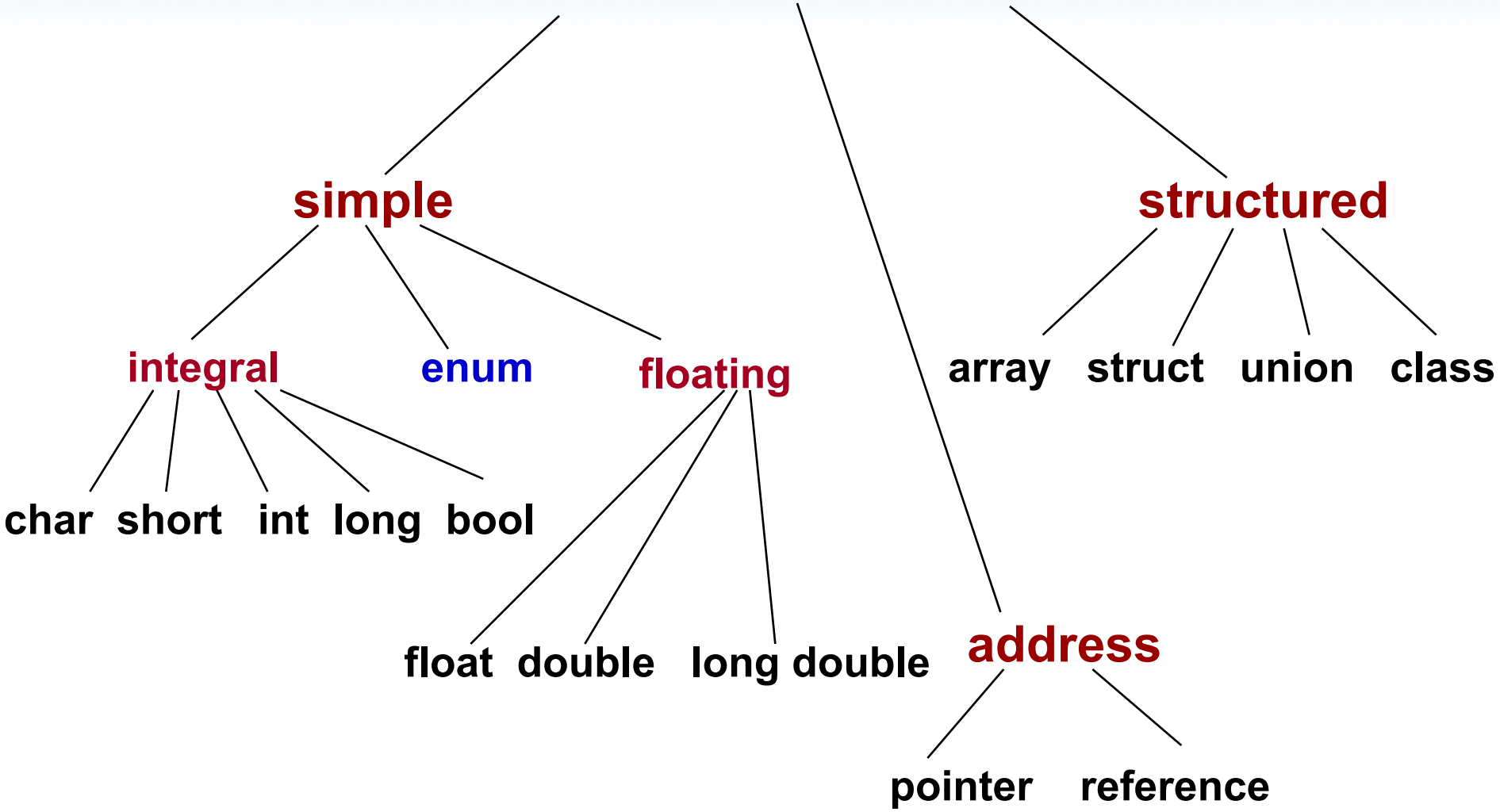
ASCII and EBCDIC

- **ASCII** (pronounced ask-key) and **EBCDIC** are two character sets commonly used to represent characters internally as one-byte integers
- ASCII is used on most personal computers; EBCDIC is used mainly on IBM mainframes
- The character 'A' is internally stored as integer 65 in ASCII and 193 in EBCDIC
- In both sets, uppercase and lowercase letters are in alphabetical order, allowing character comparisons such as 'A' < 'B', 'a' < 'b'...
- ASCII is a subset of **Unicode**, a character set that uses two bytes to represent each character and has a wider international following than **ASCII**

ASCII (Printable) Character Set

Left Digit(s) \ Right Digit	0	1	2	3	4	5	6	7	8	9
3			”	!	“	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	I	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~			

C++ Data Types



`typedef` statement

- typedef creates an additional name for an already existing data type

- Before bool type became part of ISO-ANSI C ++, a Boolean type was simulated this way on the following slide

typedef statement

```
typedef  int  Boolean;
const  Boolean  true = 1;
const  Boolean  false  = 0;

     :
Boolean dataOK;

     :
dataOK  =  true;
```

Combined Assignment Operators

```
int   age;
cin >>  age;
```

A statement to add 3 to age

```
age   =    age + 3;
```

OR

```
age   +=   3;
```

A statement to subtract 10 from `weight`

```
int   weight;
cin >>  weight;
```

```
weight  =  weight - 10;
```

OR

```
weight  -=  10;
```

A statement to divide money by 5.0

```
float   money;
cin >>  money;
```

```
money   =   money / 5.0;
```

OR

```
money   /= 5.0;
```

A statement to double `profits`

```
float   profits;
cin >>  profits;
```

```
profits  = profits * 2.0;
```

OR

```
profits  *=  2.0;
```

A statement to raise `cost` 15%

```
float   cost;
cin >>  cost;
     cost = cost + cost * 0.15;
```

OR

```
cost = 1.15 * cost;
```

OR

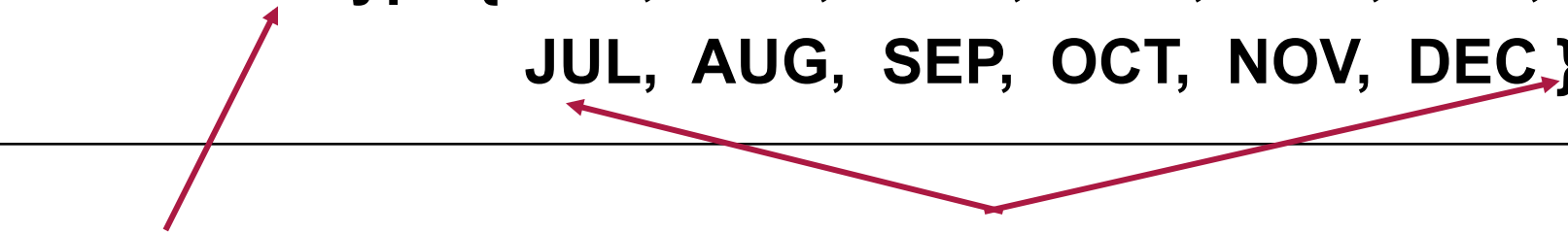
```
cost *= 1.15;
```

Enumeration Types

- C++ allows creation of a new simple type by listing (enumerating) all the ordered values in the domain of the type

EXAMPLE

```
enum MonthType { JAN,  FEB,  MAR,  APR,  MAY, JUN,
                 JUL,  AUG,  SEP,  OCT,  NOV,  DEC };
```



name of new type

list of all possible values of this new type

enum Type Declaration

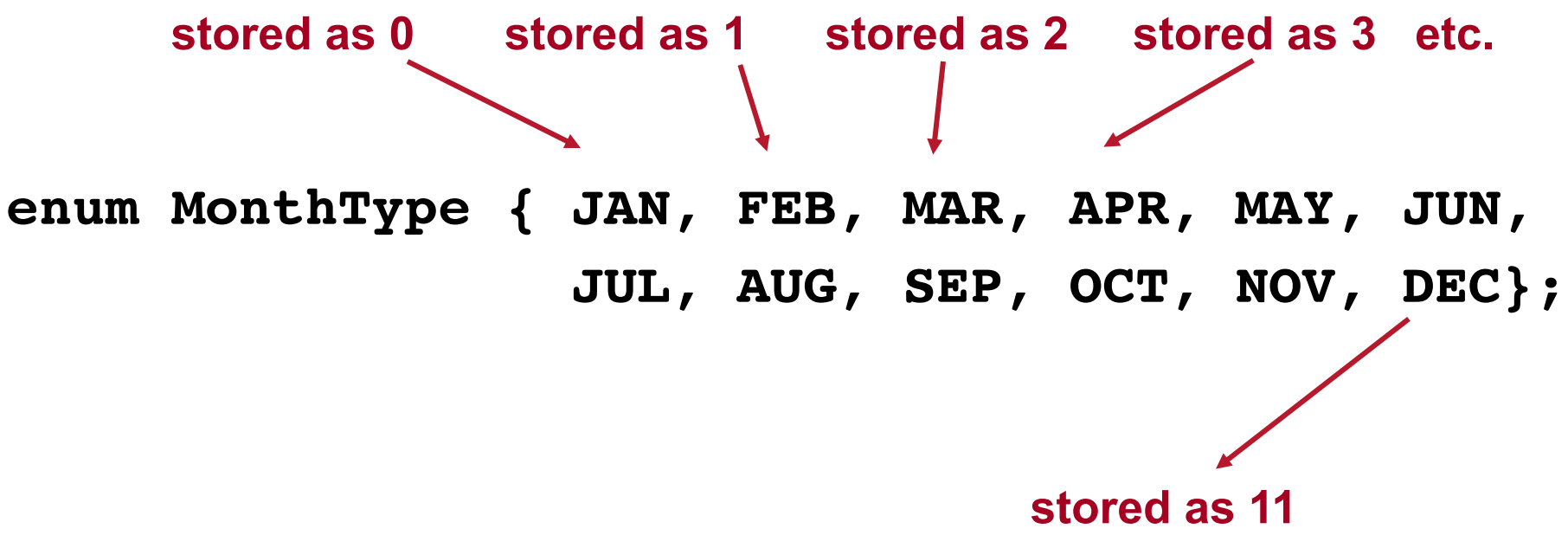
```
enum MonthType { JAN, FEB, MAR, APR, MAY, JUN,
                 JUL, AUG, SEP, OCT, NOV, DEC};
```

- The enum declaration creates a new programmer-defined type and lists all the possible values of that type--any valid C++ identifiers can be used as values
- The listed values are ordered as listed; that is, JAN < FEB < MAR < APR , and so on
- You must still declare variables of this type

Declaring enum Type Variables

```
enum  MonthType { JAN,  FEB,  MAR,  APR,  MAY,
JUN,
                  JUL,  AUG,  SEP,  OCT,  NOV,
DEC };

MonthType  thisMonth; // Declares 2 variables
MonthType  lastMonth; // of type MonthType

lastMonth  =  OCT;    // Assigns values
thisMonth  =  NOV;    // to these variables

lastMonth = thisMonth;
thisMonth = DEC;
```

Storage of `enum` Type Variables



Use Type Cast to Increment `enum` Type Variables

```
enum MonthType { JAN, FEB, MAR, APR, MAY, JUN,
                 JUL, AUG, SEP, OCT, NOV, DEC};
MonthType  thisMonth;
MonthType  lastMonth;

lastMonth  =  OCT;
thisMonth  =  NOV;
lastMonth = thisMonth;
```

Use Type Cast to Increment `enum` Type Variable, cont...

```
thisMonth = thisMonth++; // COMPILE ERROR !

thisMonth = MonthType(thisMonth + 1);
// Uses type cast
```

More about enum Type

Enumeration type can be used in a Switch statement for the switch expression and the case labels

Stream I/O (using the insertion << and extraction >> operators) is not defined for enumeration types; functions can be written for this purpose

More about `enum` Type

Comparison of enum type values is defined using the 6 relational operators (< , <= , > , >= , == , !=)

An enum type can be the return type of a value-returning function in C++

```
MonthType  thisMonth;

switch (thisMonth) // Using enum type switch
  expression
{
    case    JAN  :
    case    FEB  :
    case    MAR  :   cout << "Winter quarter";
                     break;
   case     APR  :
   case     MAY  :
   case     JUN  :   cout << "Spring quarter";
                    break;
```

```
case    JUL  :
case    AUG  :
case    SEP  :   cout << "Summer quarter";
                  break;
case    OCT  :
case    NOV  :
case    DEC  :   cout << "Fall quarter";
}
```

Using enum type Control Variable with for Loop

```
enum  MonthType { JAN,  FEB,  MAR,  APR,  MAY, JUN,
  JUL,  AUG,  SEP,  OCT,  NOV,  DEC };

void  WriteOutName (/* in */ MonthType); //
  Prototype
      .
      .
      .
```

Using enum type Control Variable with for Loop

```
MonthType  month;

for  (month = JAN; month <= DEC;
      month = MonthType (month + 1))
// Requires use of type cast to increment
{
     WriteOutName (month);
    // Function call to perform output
      :
}
```

```
void   WriteOutName ( /*  in  */  MonthType
  month)
// Prints out month name
// Precondition:  month is assigned
// Postcondition: month name has been
// written out
```

```
{       switch (month)
        {
                case JAN : cout << " January ";   break;
                case FEB : cout << " February:    break;
                case MAR : cout << " March ";   break;
                case APR : cout << " April ";   break;
                case MAY : cout << " May ";     break;
                case JUN : cout << " June ";    break;
                case JUL : cout << " July ";    break;
                case AUG : cout << " August ";        break;
                case SEP : cout << " September "; break;
                case OCT : cout << " October ";     break;
                case NOV : cout << " November "; break;
                case DEC : cout << " December "; break;
        }
}
```

Function with enum Type Return Value

```
enum SchoolType {PRE_SCHOOL, ELEM_SCHOOL,
  MIDDLE_SCHOOL,  HIGH_SCHOOL,  COLLEGE };
```

```
SchoolType   GetSchoolData (void)

// Obtains information from keyboard to
// determine level
// Postcondition: Return value ==
//                          personal school level
{
    SchoolType  schoolLevel;
    int age;
    int lastGrade;
    // Prompt for information
    cout  << "Enter age :  ";
    cin  >> age;
```

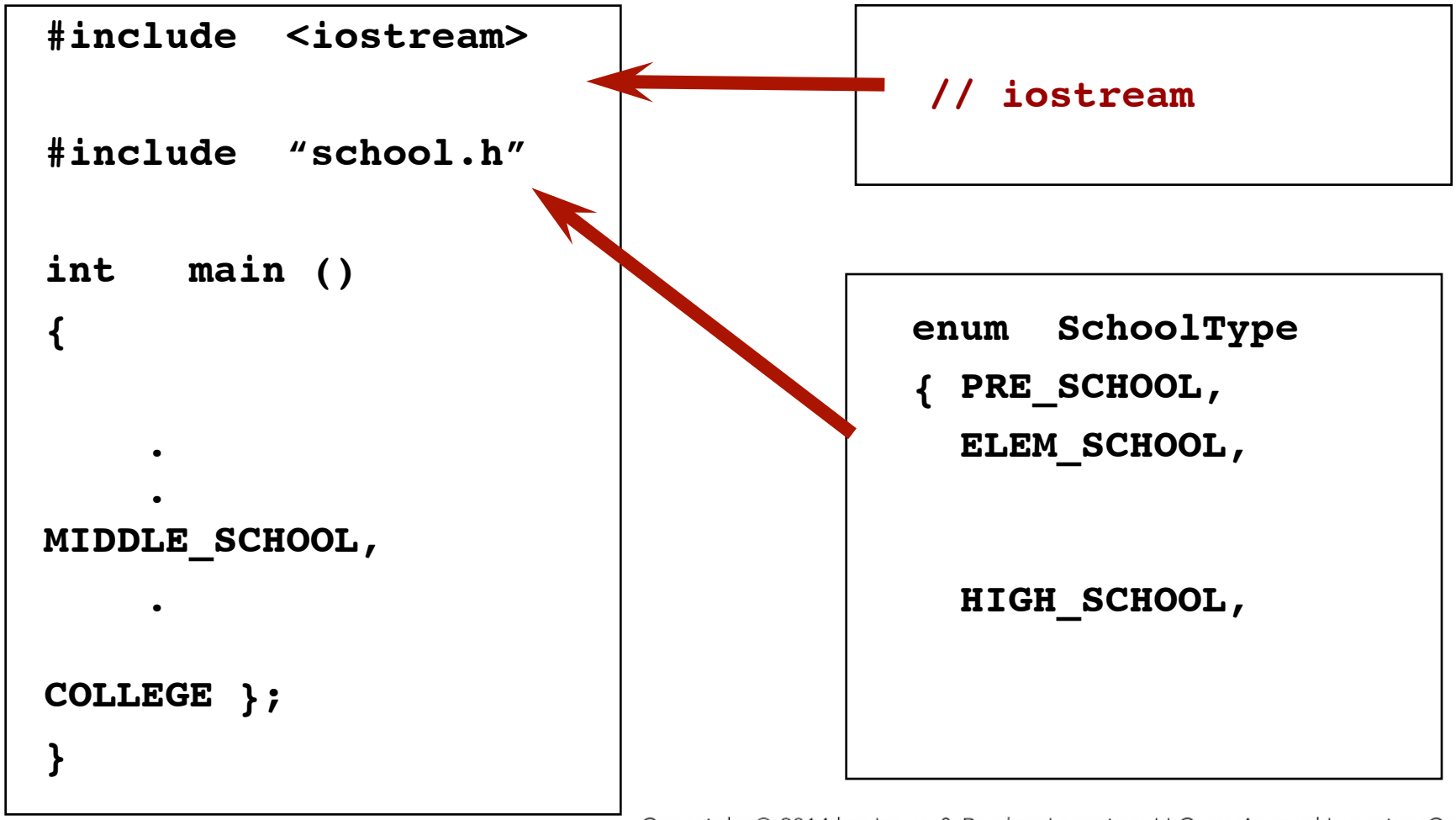
```
if  (age  <   6)
     schoolLevel = PRE_SCHOOL;

 else
 {
   cout <<  "Enter last grade completed in "
        <<  " school: ";
   cin  >> lastGrade;
```

```
    if  (lastGrade  <  5)
                schoolLevel  =  ELEM_SCHOOL;
            else if  (lastGrade  <  8)
                schoolLevel  =  MIDDLE_SCHOOL;
                else if  (lastGrade  < 12)
                    schoolLevel  =  HIGH_SCHOOL;
                else
                    schoolLevel  =  COLLEGE;
        }
        // Return enum type value
        return  schoolLevel;
    }
```

Multifile C++ Programs

- C++ programs often consist of several different files with extensions such as .h and .cpp
- Related typedef statements, const values, enum type declarations, and similar items are often placed in user-written header files
- By using the `#include` preprocessor directive, the contents of these header files are inserted into any program file that uses them

Inserting Header Files



Structured Data Type

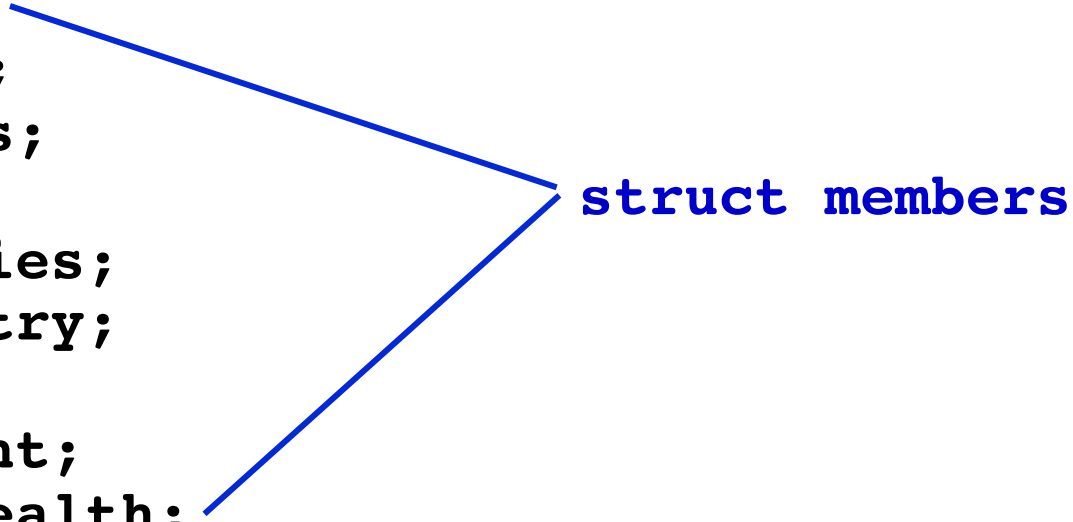
A structured data type is a type in which each value is a collection of component items

- **The entire collection has a single name**
- **Each component can be accessed individually**
- **Used to bundle together related data of various types for convenient access under the same identifier**

For example . . .

struct AnimalType

```
enum  HealthType  { Poor, Fair, Good, Excellent };

struct  AnimalType // Declares a  struct data type
{                  //  does not allocate memory
    long      id;
    string    name;
    string    genus;

    string    species;
    string    country;
    int       age;
    float     weight;
    HealthType  health;
};
```



struct AnimalType

```
// Declare  variables of AnimalType

AnimalType thisAnimal;
AnimalType anotherAnimal
```

`struct` type Declaration

SYNTAX

```
struct  TypeName          // Does not  allocate memory
{
    MemberList
};
```

`struct` type Declaration

The struct declaration names a type and names the members of the struct

It does not allocate memory for any variables of that type!

You still need to declare your struct variables

More about `struct` type declarations

Scope of a struct

- If the struct type declaration precedes all functions, it will be visible throughout the rest of the file
- If it is placed within a function, only that function can use it

More about `struct` type declarations

- It is common to place struct type declarations in a (.h) header file and #include that file
- It is possible for members of *different* struct types to have the same identifiers;
- Also a non-struct variable may have the same identifier as a structure member

Accessing `struct` Members

Dot (period) is the member selection operator

After the struct type declaration, the various members can be used in your program only when they are preceded by a struct variable name and a dot

EXAMPLES

```
thisAnimal.weight
anotherAnimal.country
```

Operations on struct Members

The type of the member determines the allowable operations

```
thisAnimal.age  =  18;
thisAnimal.id   =  2037581;
cin  >>  thisAnimal.weight;
getline (cin, thisAnimal.species);
thisAnimal.name = “giant panda”;
thisAnimal.genus[0] =
  toupper(thisAnimal.genus[0]);
thisAnimal.age++;
```

Aggregate Operation

An aggregation operation is an operation on a data structure as a whole, as opposed to an operation on an individual component of the data structure

Aggregate `struct` Operations

- **Operations valid on struct type variables are**
  - **Assignment to another struct variable of the same type**
  - **Pass as an argument (by value or by reference)**
  - **Return as value of a function**
- **I/O, arithmetic, and comparisons of entire struct variables are NOT ALLOWED!**

Aggregate `struct` Operations

```
// Assignment
anotherAnimal = thisAnimal;

// Value parameter
WriteOut(thisAnimal);

// Reference parameter
ChangeWeightAndAge(thisAnimal);

// Function return value
thisAnimal = GetAnimalData();
```

```
void  WriteOut( /*  in  */ AnimalType thisAnimal)
// Prints out values of all members of thisAnimal
// Precondition: all members of thisAnimal
//   are assigned
// Postcondition:all members have been written out
```

```
{
    cout << "ID # " << thisAnimal.id
         << thisAnimal.name  << endl;

        cout << thisAnimal.genus
             << thisAnimal.species
             << endl;
```

```
cout << thisAnimal.country << endl;

    cout << thisAnimal.age << " years " << endl;

    cout << thisAnimal.weight << " lbs. "
         << endl;

       cout << "General health : ";

    WriteWord (thisAnimal.health);
}
```

Passing a `struct` Type by Reference

```
void ChangeAge(/* inout */ AnimalType& thisAnimal)

// Adds 1 to age
// Precondition: thisAnimal.age is assigned
// Postcondition:thisAnimal.age ==
//   thisAnimal.age@entry + 1

{

    thisAnimal.age++;

}
```

```
AnimalType GetAnimalData ()

// Obtains all information about an animal from
// keyboard
// Postcondition:
//    Return value == AnimalType members entered at
//    kbd
{
```

```
{
        AnimalType  thisAnimal;
        char response;
        do
    {
    // Have user enter members until they are
    // correct
                    .

                    .

                    .
        }  while (response != 'Y');
        return  thisAnimal;
}
```

Hierarchical Structures

- The type of a struct member can be another struct type
- This is called nested or hierarchical structures
- Hierarchical structures are very useful when there is much detailed information in each record

For example . . .

struct MachineRec

- Information about each machine in a shop contains:
- an idNumber;
- a written description;

struct MachineRec

- **the purchase date;**
- **the cost;**
- **and a history (including failure rate, number of days down;**
- **and date of last service);**

```
struct  DateType
{
     int     month;          //  Assume 1 . . 12
     int     day;            //  Assume  1 . . 31
     int     year;           //  Assume 1900 . . 2050
};
struct  StatisticsType
{
     float  failRate;
     // DateType is a struct type
     DateType lastServiced;
     int downDays;
};
```

```
struct MachineRec
{
    int idNumber;
    string description;
    // StatisticsType is a struct
    StatisticsType  history;
    DateType      purchaseDate;
    float cost;
};
MachineRec  machine;
```

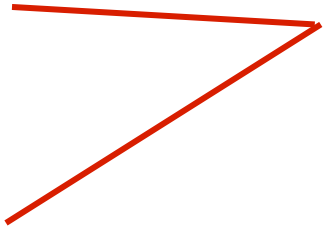
Unions in C++

DEFINITION

A union is a struct that holds only one of its members at a time during program execution.

EXAMPLE

```
union WeightType
{
     long wtInOunces;
     int  wtInPounds;
     float wtInTons;
 };
```



Only one at at time

Using Unions

```
// Declares a union type
union  WeightType
{
     long  wtInOunces;
     int   wtInPounds;
     float wtInTons;
 };
```

Using Unions

```
// Declares a union variable
WeightType   weight;
weight.wtInTons = 4.83;

//  Weight in tons is no longer
//  needed
//  Reuse the memory space

weight.wtInPounds = 35;
```

Pointer Variables in C++

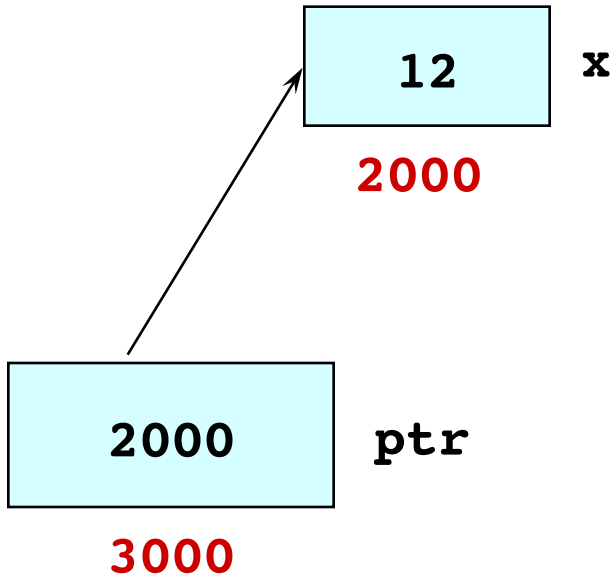
- A pointer variable is a variable whose value is the address of a location in memory

- To declare a pointer variable, you specify the type of value that the pointer will point to, for example:

```
int* ptr; // ptr will hold the address of an int

char*  q; // q will hold the address of a char
```

Using a Pointer Variable

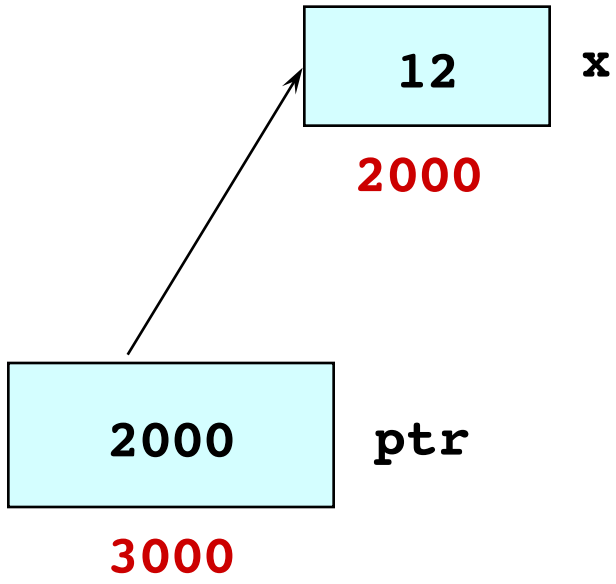
```
int   x;
x = 12;


int*   ptr;
ptr = &x;
```



NOTE: Because ptr holds the address of x, we say that ptr "points to" x

Unary operator * is the indirection (deference) operator

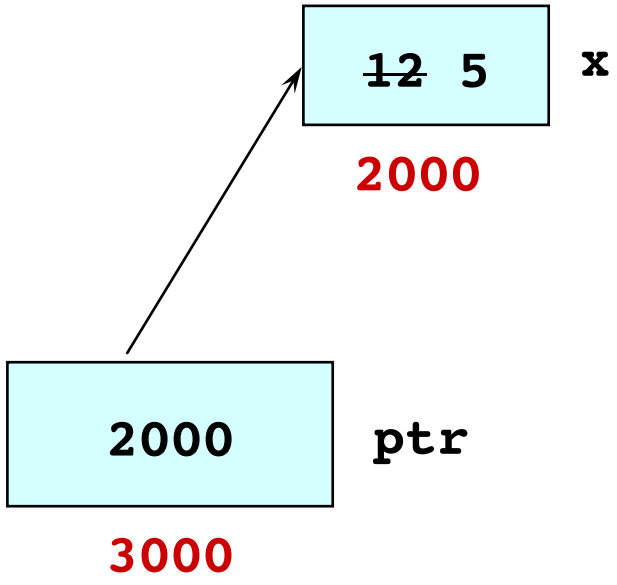
```
int   x;
x = 12;


int*   ptr;
ptr = &x;

cout   <<   *ptr;
```



NOTE: The value pointed to by ptr is denoted by *ptr

Using the Dereference Operator

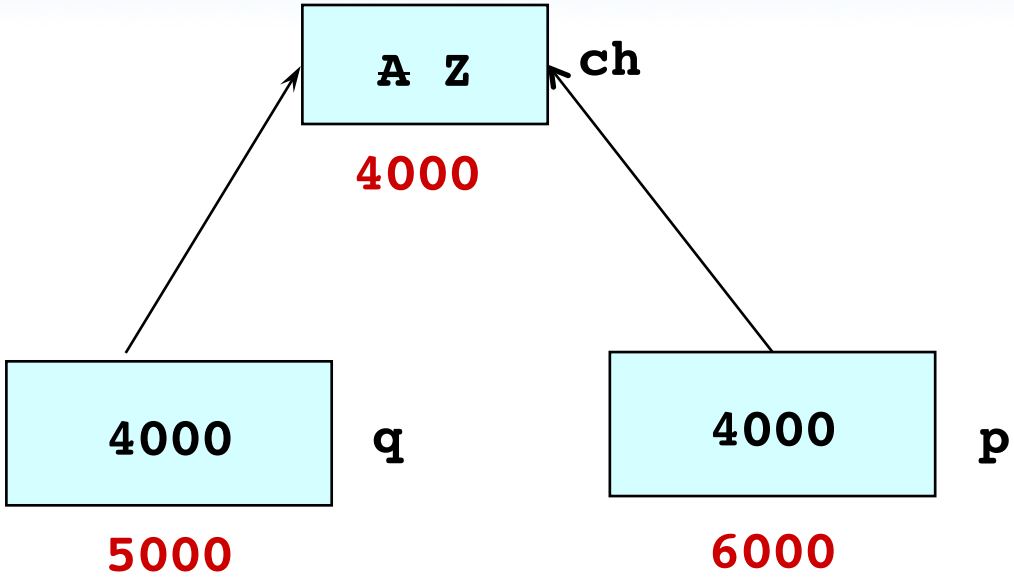
```
int   x;
x = 12;


int*   ptr;
ptr = &x;

*ptr = 5;
```




```
// Changes the value
//  at address ptr to 5
```

Another Example

```
char  ch;
ch =  'A';

char*  q;
q  = &ch;

*q = 'Z';
char*  p;
p = q;
```




```
// The rhs has value 4000
// Now p and q both point
// to ch.
```

Pointer Expressions

- **Arithmetic expressions** are made up of variables, constants, operators, and parenthesis.
- **Pointer expressions** are composed of pointer variables, pointer constants, pointer operators, and parenthesis.

Pointer Constants

- In C++, there is only one literal pointer:
  - The value 0 (the NULL pointer)

```
char* charPtr = 0;
```

- Programmers prefer to use the named constant **NULL** defined in `cstddef`:

```
#include <cstddef>
char* charPtr = NULL;
```

Pointers to Structs

- Pointers can point to any type of variable, including structs:

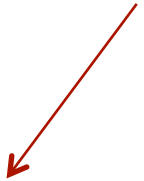
```
struct PatientRec
{
  int idNum;
  int height;
  int weight;
};
PatientRec patient;
PatientRec* patientPtr = &patient;
```

Pointers to Structs

- Pointers can point to any type of variable, including structs:

```
struct PatientRec
{
   int idNum;
   int height;
   int weight;
};
PatientRec patient;
PatientRec* patientPtr = &patient;
```

A pointer variable of Type "pointer to PatientRec"



Pointers, Structs, & Expressions

- How can I access a struct member variable using a pointer to a struct?

Pointers, Structs, & Expressions

- How can I access a struct member variable using a pointer to a struct?
- **Approach #1:**

```
(*patientPtr).weight = 160;
```

Pointers, Structs, & Expressions

- How can I access a struct member variable using a pointer to a struct?
- **Approach #1:**

```
(*patientPtr).weight = 160;
```

First, dereference. We need to use parenthesis because the '.' operator has higher precedence.

Pointers, Structs, & Expressions

- How can I access a struct member variable using a pointer to a struct?
- **Approach #1:**

```
(*patientPtr).weight = 160;
```

Then, we access the member variable.

Pointers, Structs, & Expressions

- How can I access a struct member variable using a pointer to a struct?
- **Approach #1:**

```
(*patientPtr).weight = 160;
```

- **Approach #2:**

```
patientPtr->weight = 160;
```

Pointers, Structs, & Expressions

- How can I access a struct member variable using a pointer to a struct?
- **Approach #1:**

```
(*patientPtr).weight = 160;
```

- **Approach #2:**

Because member access is so common we use the '->' operator as a shorthand for * and ().

```
patientPtr->weight = 160;
```

Approach #1 and #2 "do the same thing"!

Reference Types

- Like pointer variables, reference variables contain the addresses of other variables:

```
PatientRec& patientRef;
```

- This declares a variable that contains the address of a `PatientRec` variable.

Reference versus Pointers

- **Similarities**
  - Both contain addresses of data objects.
- **Differences**
  - Pointers require * for dereference and & to get the address of a data object.
  - References do this automatically

Reference/Pointer Comparison

Using a Pointer Variable

```
int gamma = 26;
int* intPtr = gamma;
// intPtr is a pointer
// variable that points
// to gamma.

*intRef = 35;
// gamma == 35

*intRef = *intRef + 3;
// gamma == 38
```

Using a Reference Variable

```
int gamma = 26;
int& intRef = gamma;
// intRef is a reference
// variable that points
// to gamma.

intRef = 35;
// gamma == 35

intRef = intRef + 3;
// gamma == 38
```

C++ Data Types