Chapter 8

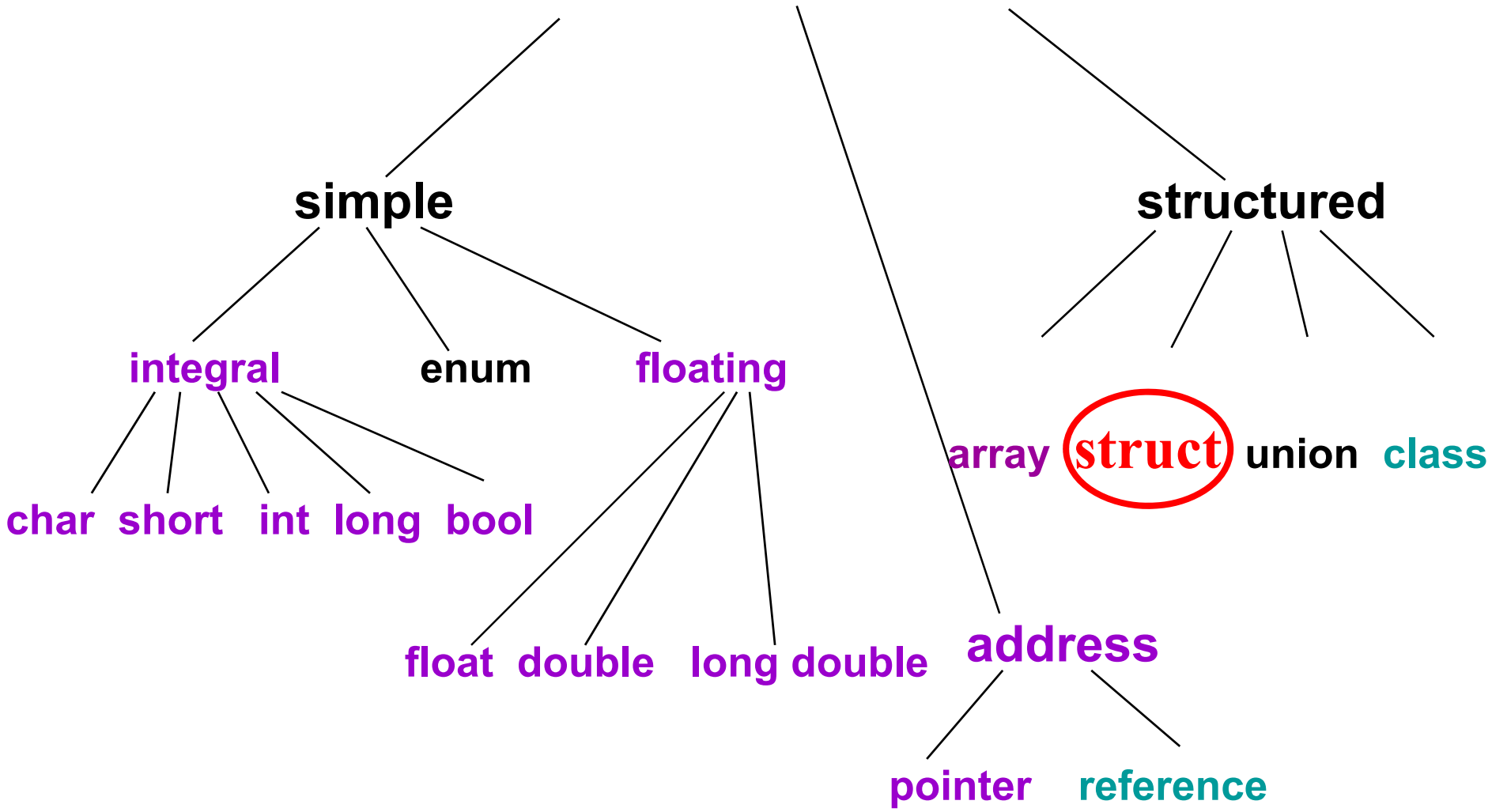
Structures & Large Program Design

Home Work:

A Project "Student Information Management"

(1)Using struct STUDENT to declare Students[100] in main().
(2)Write a function Input() to input 5 students' data to Students[100].
(3)Write a function Print() to print all students' information in Students[100].
(4)Write a function WriteToFile() to write all students information to file "Students.txt".
(5)Write a function ReadFromFile() to read from "Students.txt" to Students[100].
(6)Write a function Append() to input 1 new student's data to the end of all current students.
(7)Write a function RetrieveByID() to search a student with specific nID and print all information about this student.
(8)Write a function RetrieveByGender() to search all 'M'(Male) student(s) and print all information about them.
(9)Write a function DeleteAStudent() to delete a student with specific nID, then move all the following student(s) forward and print all existed students.
(10)Write a function SortData() to sort all students by their nID in ascendent(from small to large).
(11)Write a function InsertAStudent() to Insert a new student according to nID in a sorted Students(It meas, frist to move all the array elements of Students with larger nID backward to leave a blank Students element. Then copy the new student's information to the blank Students element).

C++ Data Types



Different Data Types->struct

One Person may have:

```
int        nAge;
double     fWeight;
char       cGender;
char       szName[10];
```

```
struct PERSON
{
  int        nAge;
  double     fWeight;
  char       cGender;
  char       szName[10];
};
```

PERSON						
nAge	**fWeight**	**cGender**	**szName**			

Different Data Types--struct

PERSON						
nAge	**fWeight**	**cGender**	**szName**			

```
struct  PERSON
{
  int          nAge;
  double       fWeight;
  char         cGender;
  char         szName[10];
};
```

struct-A grouping of possibly **different** data types

Once a structure is defined, we treat it much the same as we treat ***int*** and ***float***

Array- A set of **same** data type

Initialization of struct

```
struct  PERSON
{
    int         nAge;
    double      fWeight;
    char        cGender;
    char        szName[10];
};
PERSON    Li={19,50.6,'M',"Li"};
```

Li						
nAge	**fWeight**	**cGender**	**szName**			
19	50.6	'M'	L	i	\0	

Accessing Members

```
struct  PERSON
{
    int         nAge;
    double      fWeight;
    char        cGender;
    char        szName[10];
};
PERSON          Wang;
struct  PERSON  zhang;
    Wang.nAge=18;
    Wang.fWeight=60.2;
    strcpy(Wang.szName,"Wang");
    Zhang=Wang;
```

Wang						
nAge	**fWeight**	**cGender**	**szName**			
19	50.6	'M'	L	i	\0	

```
#include "string.h"
struct  PERSON
{
int         nAge;
double   fWeight;
char       cGender;
char       szName[10];
};
int _tmain(int argc, _TCHAR* argv[])
{
PERSON                  Wang,Li={19,50.6,'M',"Li"};
struct   PERSON         zhang;
  Wang.nAge=18;
  Wang.fWeight=60.2;
  strcpy(Wang.szName,"Wang");
  printf("Name=%s,Age=%d\n",Wang.szName,Wang.nAge);
  return 0;
}
```

Try!

Different struct Declarations

```
struct PERSON
{
int       nAge;
double fWeight;
char     cGender;
char     szName[10];
};
PERSON Wang;
struct PERSON Li;
```

```
struct PERSON
{
int       nAge;
double fWeight;
char     cGender;
char     szName[10];
}Wang,Li;
```

```
struct   ?
{
int       nAge;
double fWeight;
char     cGender;
char     szName[10];
}Wang,Li;
```

But cannot used for further declarations

Try!

- To define a *struct* STUDENT with 2 members:

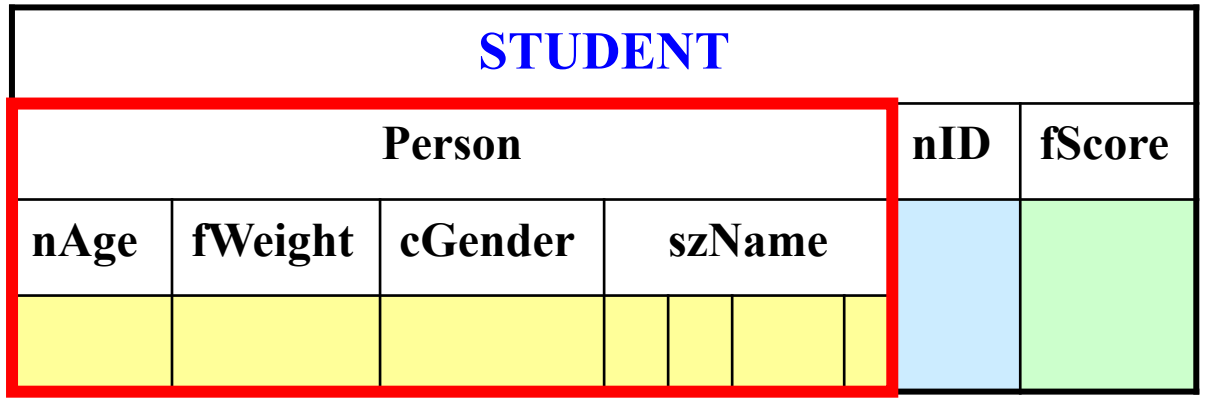
(1)int nID
(2)float fScore

- Declare 2 variables named Hou and Jin of *struct* STUDENT

Hou	
nID	fScore

Nested struct

```
struct  PERSON
{
    int          nAge;
    double       fWeight;
    char         cGender;
    char         szName[10];
};
struct STUDENT
{
    struct  PERSON   Person;
    int              nID;
    float            fScore;
}Hou,Jin;
```



Structure member can be another structure

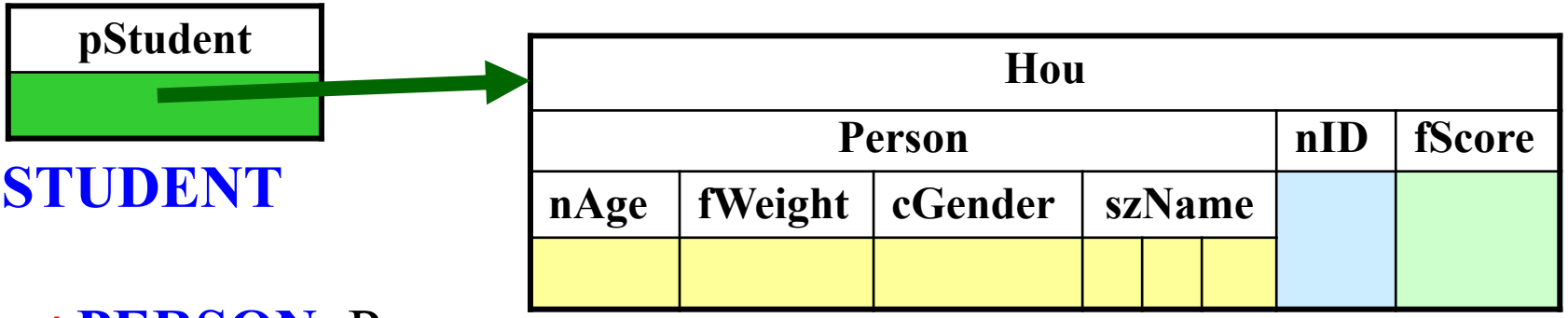
Try!

```
#include "string.h"
struct  PERSON
{
  int        nAge;
  double  fWeight;
  char      cGender;
  char      szName[10];
};

struct STUDENT
{
  struct PERSON Person;
  int                    nID;
  float                 fScore;
}Hou,Jin;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
  Hou.nID=1001;
  Hou.Person.nAge=20;
  strcpy(Hou.Person.szName,"Hou");
  Jin=Hou;
  return 0;
}
```

Pointers to Structures




```
struct STUDENT
{
    struct PERSON  Person;
    int            nID;
    float          fScore;
} Hou;
```

- To declare pointer to structure

  STUDENT *pStudent;
- Pointer assignment

  pStudent=**&**Hou;
- Access Members Using Pointers

  pStudent->nID=1002;

  (*pStudent).fScore=60.2;

Try!

Array of struct

```
struct  PERSON
{
    int         nAge;
    double      fWeight;
    char        cGender;
    char        szName[10];
};
```

	Persons						
	nAge	**fWeight**	**cGender**	**szName**			
0	19	50.6	'M'	L	i		
1	20	60.1	'F'	Y	u		
2	21	70.2	'F'	L	i	u	
3	22	80.3	'M'	W	u		
4							

```
PERSON    Persons[5]={{19,50.6,'M',"Li"},
                      {20,60.1,'F',"Yu"},
                      {21,70.2,'F',“Liu"},
                      {22,80.3,'M',"Wu"}};
    Persons[0].fWeight=47.2;
```

Array of Nested struct

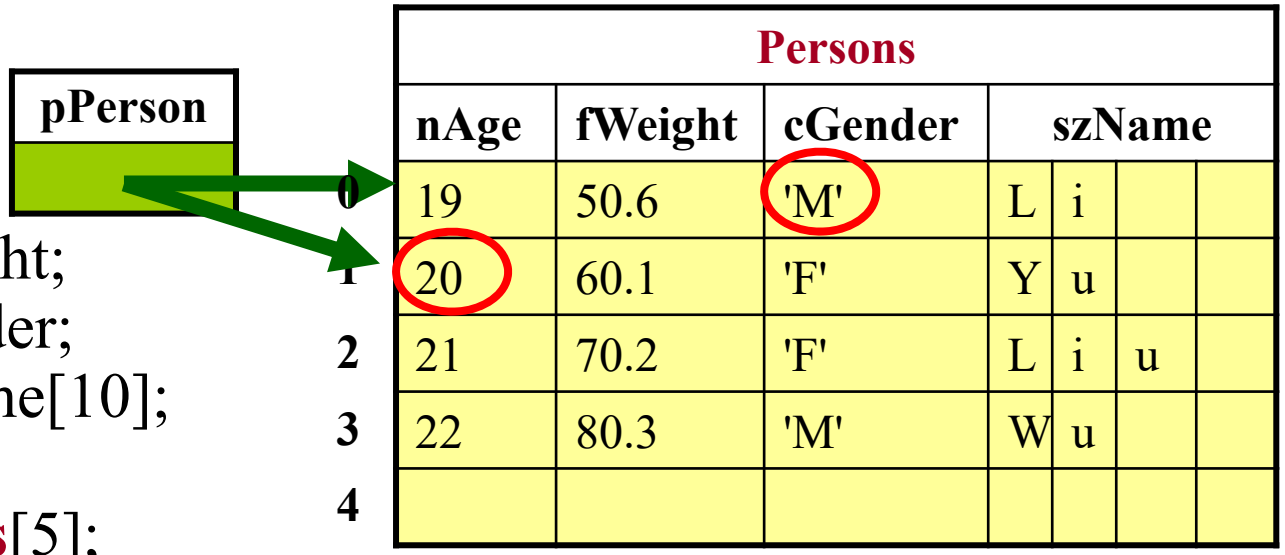
```
struct  PERSON
{
  int        nAge;
  double  fWeight;
  char      cGender;
  char      szName[10];
};
struct  STUDENT
{
  struct  PERSON  Person;
  int                  nID;
  float                fScore;
};
struct  STUDENT  Students[3];
   Students[0].nID=1001;
   Students[1].Person.fWeight=50.0;
```

	Students							
	Person						nID	fScore
	nAge	fWeight	cGender	szName				
0							101	
1		50.0						
2								

Try!

Pointer to Array of struct

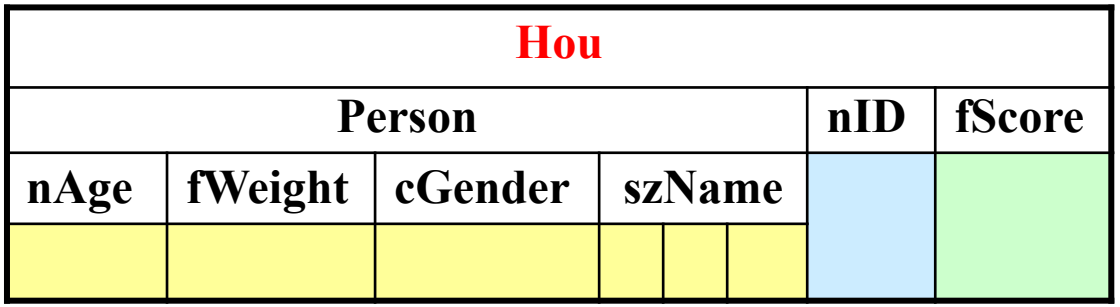
```
struct  PERSON
{
    int          nAge;
    double       fWeight;
    char         cGender;
    char         szName[10];
};
PERSON     Persons[5];
PERSON    *pPerson;
  pPerson=Persons;
  pPerson->cGender='F';
  pPerson++;
  (*pPerson).nAge=18;
  printf("Age=%d\n",pPerson->nAge);
```



Try!

struct and Functions(1)

```
struct STUDENT
{
  struct PERSON  Person;
  int            nID;
  float          fScore;
}Hou;
int _tmain(int argc, _TCHAR* argv[])
{
void  PrintAStudent(struct STUDENT AStudent);
  PrintAStudent(Hou);
  return 0;
}
void PrintAStudent(struct STUDENT AStudent)
{
  printf("ID=%d,",AStudent.nID);
  printf("Name=%d,",AStudent.Person.szName);
  printf("Age=%d,",AStudent.Person.nAge);
  printf("Score=%.1f\n",AStudent.fScore);
}
```



- Complete copies of all members of a structure can also be passed to a function by including the name of the structure as an argument to the called function.

Students							
Person						nID	fScore
nAge	fWeight	cGender	szName				
	50.0					101	

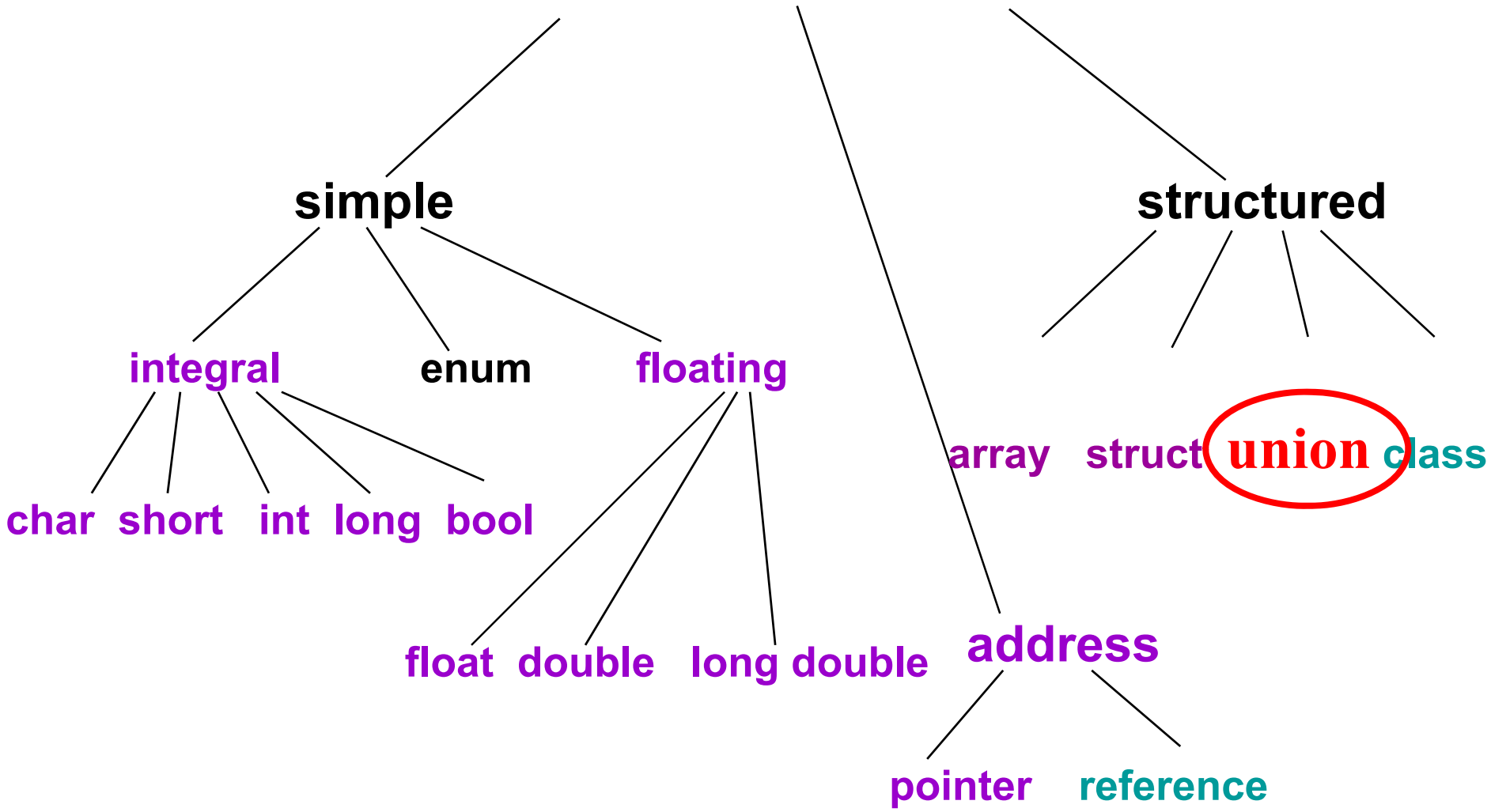
```
int _tmain(int argc, _TCHAR* argv[])
{
STUDENT  Students[3];
    InitializeStudents(&Students[0]);
   …
}
void InitializeStudents(STUDENT Data[3])
{
    Data[0].nID=1001;
    Data[0].fScore=80.0;
    Data[0].Person.nAge=20;
    Data[0].Person.fWeight=50.0;
    Data[0].Person.cGender='M';
    strcpy(Data[0].Person.szName,"Zhang");
…
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
STUDENT  Students[3];
    pMinStudent=FindMinScoreStudent(3,Students);
   …
}
struct STUDENT* FindMinScoreStudent(int nNum,STUDENT *pStudent)
{
 float            fMinScore=10000.0;
 STUDENT   *pMinStudent;
  for(int i=0;i<nNum;i++)
  {
    if(fMinScore>pStudent->fScore)
    {
      fMinScore=(*pStudent).fScore;
      pMinStudent=pStudent;
    }
    pStudent++;
  }
  return pMinStudent;
}
```

pStudent

Students							
Person						nID	fScore
nAge	fWeight	cGender	szName				

C++ Data Types



union (共用体)

- A ***union*** is a data type that reserves the same area in memory for two or more variables, each of which can be a different data type.
- Each of these types, but **only one** at a time, can actually be assigned to the union variable.

```
union WEIGHT
{
  double  dTons;
  int     nOunce;
  float   fPounds;
}
```

struct

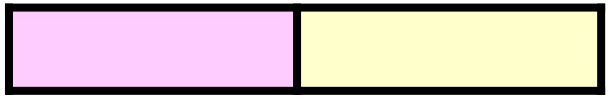
dTons	nOunce	fPounds
double	int	float
(8 bytes)	(4bytes)	(4bytes)

union

dTons (8 bytes)

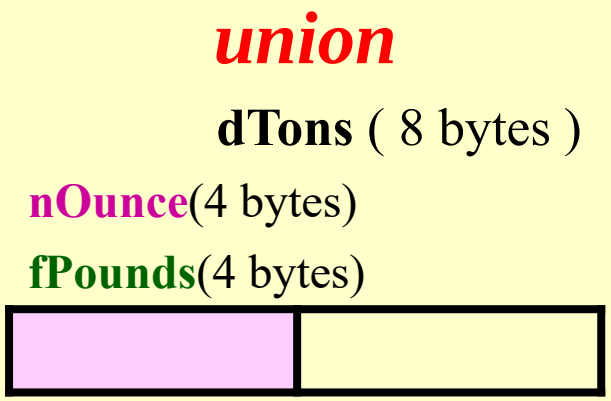
nOunce(4 bytes)

fPounds(4 bytes)



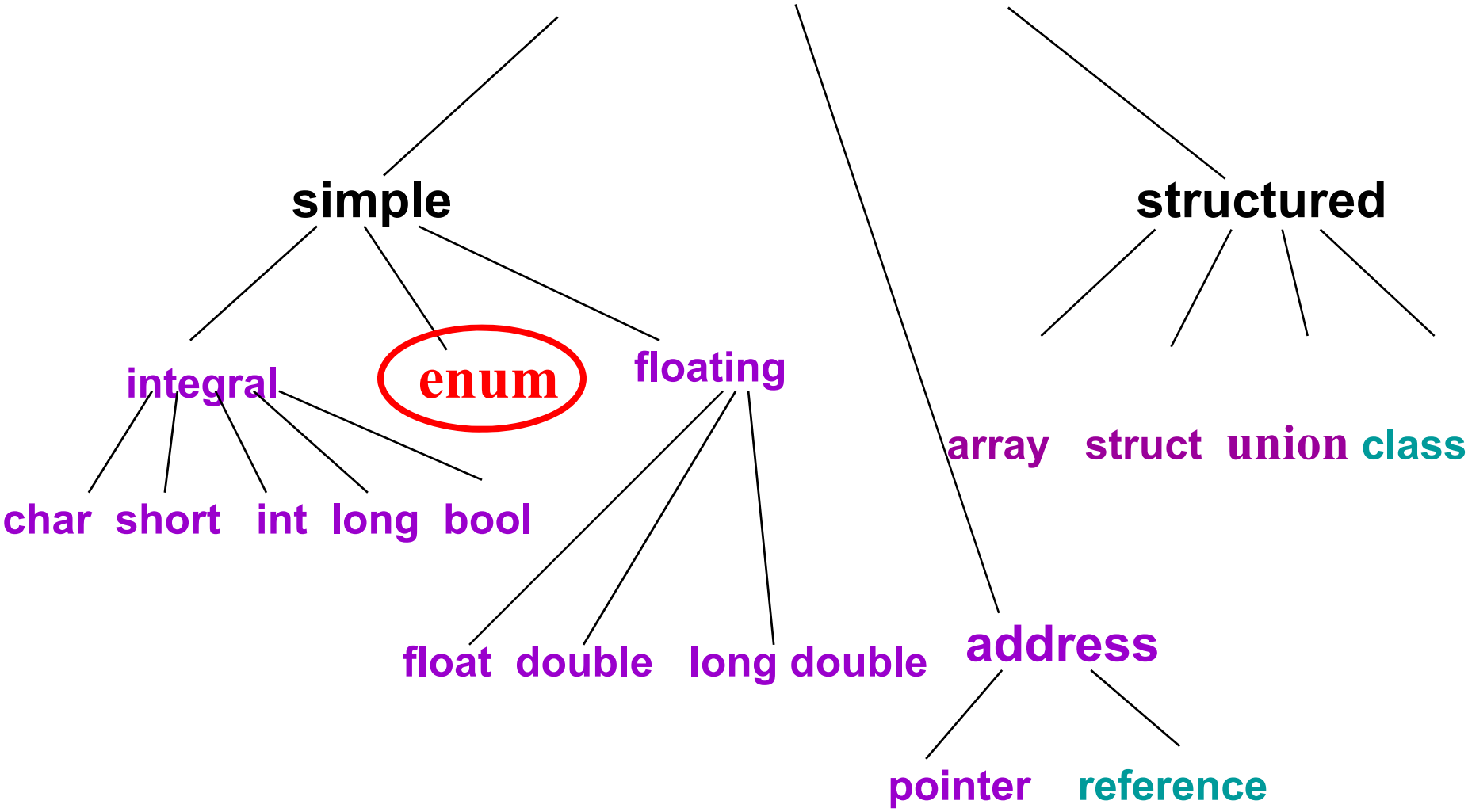
```
union  WEIGHT
{
  double  dTons;
  int       nOunce;
  float     fPounds;
};
WEIGHT  Weight;
  Weight.nOunce=123456789;
  printf("Weight.dTons=%f\n",Weight.dTons);
  printf("Weight.nOunce=%d\n",Weight.nOunce);

  Weight.dTons=4.8123456789;
  printf("Weight.dTons=%f\n",Weight.dTons);
  printf("Weight.nOunce=%d\n",Weight.nOunce);
  printf("Size of union Weight=%d\n", sizeof(Weight));
```



- After the last assignment statement, the previous int value 123456789 is gone, replaced by the double value 4.8123456789.

C++ Data Types



enum (enumeration)

- To define a new simple type by listing the literal values that make up the domain of the type.
- These literal values must be identifiers, not numbers.
- The enumerators are ordered

```
enum MONTH{JAN,FEB,MAR,APR,MAY,JUN,
                    JUL,AUG,SEP,OCT,NOV,DEC};
 MONTH ThisMonth;          // declares 2 variables
 MONTH LastMonth;          // of type MonthType

 LastMonth=OCT;                    // assigns values
 ThisMonth=NOV;
 LastMonth=ThisMonth;
 printf("JAN=%d\n",JAN);
 printf("FEB=%d\n",FEB);
 printf("MAR=%d\n",MAR);
 printf("APR=%d\n",APR);
```

typedef Declaration

- The ***typedef*** declaration statement permits constructing alternate names for an existing C data type name.

```
typedef     float                REAL;
REAL                             y;
typedef     unsigned   char      BYTE;
BYTE                             Data[1000];
struct _iobuf {
      char      *ptr;
      int       _cnt;
      char      *base;
      int       _flag;
      int       _file;
      int       _charbuf;
      int       _bufsiz;
      char      *tmpfname;
      };
typedef   struct   _iobuf        FILE;
FILE                             *fpFile;
```

Bitwise Manipulations

- C is capable of performing very low level operations of manipulating individual bits in memory
- Hexadecimal notations are convenient for representing bit patterns
- Hexadecimal notation uses preceding **0x** or **0X**, e.g. **0x**DFFF represents DFFF
- Octal notation uses preceding **0**. e.g., **0**364 represents octal 364

Decimal	Hexadecimal	Bit pattern
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

bitwise AND and bitwise OR

- bitwise AND uses notation **&**

  1 & 1= 1 (1 & 0 = 0, 0 & 0 = 0)

- bitwise OR uses notation | :

  0 | 0 = 0 (1 | 0 = 1, 1 | 1 = 1)

- E.g bitwise **&** .

	1	0	1	0
&	1	1	0	0
	1	0	0	0

bitwise Complement and Exclusive OR

- Bitwise **complement** operator ~ reverses all of the bits of its operand

  ~(1010) = 0101

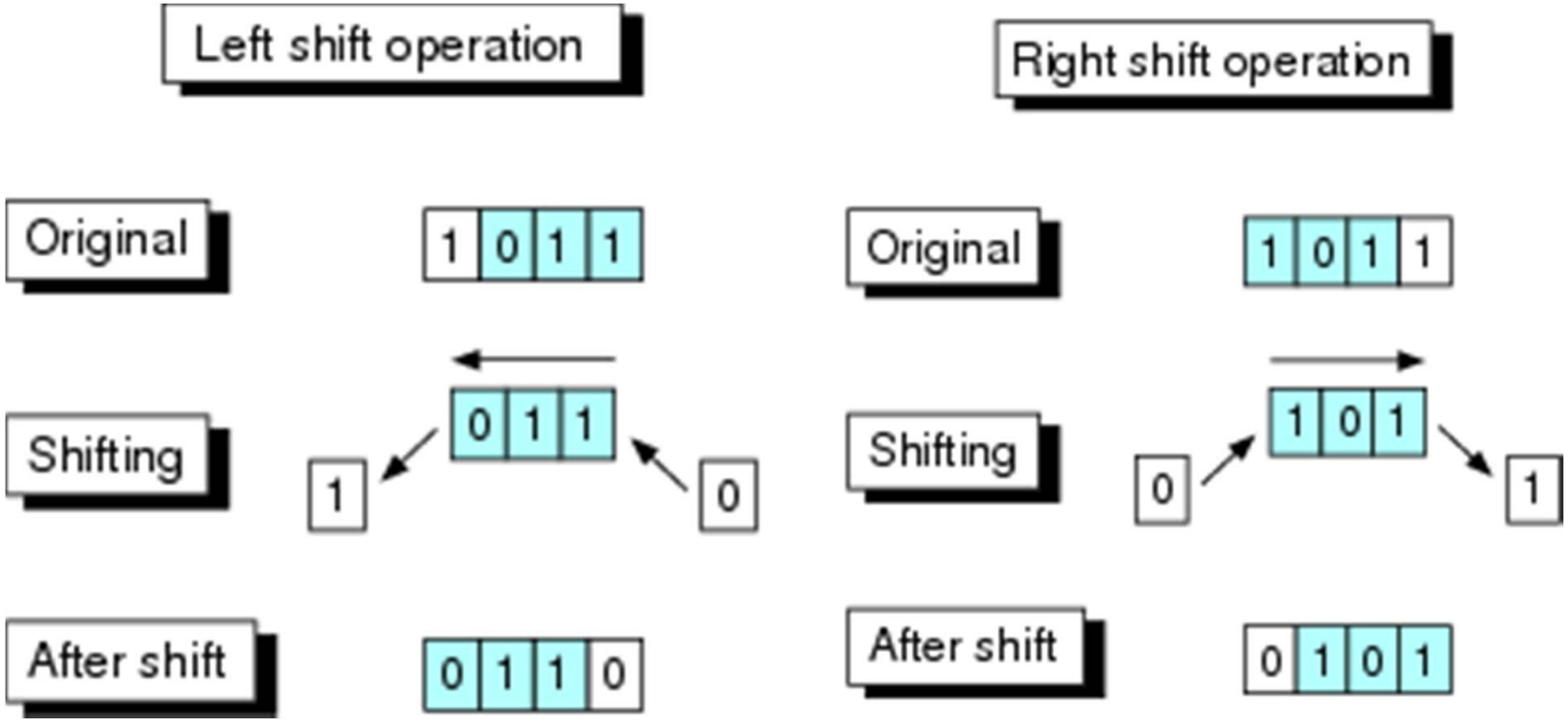
- Bitwise **Exclusive OR** (^)

  0 ^ 1 = 1 (0 ^ 0 = 0, 1 ^ 1 = 0)

	1	0	1	0
^	1	1	0	0
	0	1	1	0

bitwise shift

- bitwise shift operators (>> and <<) move all of the bits in a cell either to the right or left and add clear bits in the shift



```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
unsigned int a,b,a_BitwiseAnd_b,a_BitwiseOR_b;
unsigned int Complement,ExclusiveOR,LeftShif,RightShif;

  a=0x000A;
  b=0x000C;
  a_BitwiseAnd_b=a & b;
  a_BitwiseOR_b=a | b;
  Complement=~0x0000000f;
  ExclusiveOR=0x0000000A^0x0000000C;
  LeftShif=0x80000001<<1;
  RightShif=0x80000001>>2;
  return 0;
}
```

Part II : Advanced Programming Techniques

Header File

- To create a header file, uses extension ***.h***
- What should be placed in header file
  - Macros
  - function prototypes
  - structure definitions
- The created header file can be used by including the line

  #include "ourheaderFile.h"
- Enclosing the header file name in " "rather than <> tells the compiler that the header file probably is not within the library of the standard header files but in the working folder (the same directory with the source file)

Multiple Source Files & Storage Classes

- In developing large programs, it is not practical to keep all of source code in a single file
- Functions in different files that use other functions, are linked together using the *linker*

Storage Classes- extern

- Declaration "***extern*** *int* **x**;" does not reserve memory for **x**
- Using ***extern*** lets the compiler know that the global variable **x** is declared in another file
  - e.g. **x** is declared in **File1.cpp** & used (declare as ***extern***) in **File2.cpp**

File1.cpp

```
…
int     x;
…
 x=10;
```

File2.cpp

```
extern  int   x;
…
 x=20;   //???
```

Storage Classes- register, auto

- ***register***

  register frequently will cause single variables to be stored in the register portion of the CPU

- ***auto***

  auto specifier is the default specification for local variables

Storage Class- static

- ***static*** local variable
  - ***static*** local variable retains its value even after program control has returned to calling function
- ***static*** global variable
  - ***static*** global variable is a global variable accessible only to functions within a given file. Static global can therefore avoid name conflict

```
float GetAverage(float fData)
{
static float      fTotal=0.0;
static int        nNumber=0;
float             fAverage;
  fTotal+=fData;
  nNumber++;
  if(nNumber>0)
    fAverage=fTotal/nNumber;
  return fAverage;
}
```