G52CPP
C++ Programming
Lecture 6

Dr Jason Atkin

http://www.cs.nott.ac.uk/~jaa/cpp/
g52cpp.html

Last lecture

- The Stack
- Lifetime of local variables
- Global variables
- Static local variables

Example

```
int iGlobal = 1;

int* funcstatic()
{
   static int iStatic = 10;
   iStatic++;
   return &iStatic;
}
int* funclocal()
{
   int iLocal = iGlobal;
   iLocal++;
   return &iLocal;
}

int overwrite()
{
   int iOverwrite1 = 20;
   int iOverwrite2 = 30;
   iOverwrite1 = iOverwrite2;
   return iOverwrite1;
}
```

```
int main(int argc, char* argv[])
{
   int* piStatic = funcstatic();
   int* piLocal = funclocal();
   funcstatic();
   funclocal();

   printf( "%d %d %d\n", iGlobal,
        *piStatic, *piLocal );

   overwrite();

   printf( "%d %d %d\n", iGlobal,
        *piStatic, *piLocal );

   return 0;
}
```

visibility.cpp

This lecture

- structs
- unions

- sizeof(struct), sizeof(union)

- -> operator

- Bit fields, enums and typedef

structs

Without any methods

(for the moment)

structs

- We will start with C-type structs
  - C++ structs and classes (introduced later) can be considered to be extensions of C structs, e.g. allowing member functions, inheritance etc
  - Structs and classes are virtually the same thing in C++
- These group related data together
- Examples:
  - Group three integers together to specify a time:

```
struct Time
{
        int hour;
        int minute;
        int second;
};
```

struct.cpp

Note the ; at the end!

  - Shorter version, for day, month, year:

```
struct Date { int d, m, y };
```

Creating a **struct** on the stack

- Create objects of type struct using the name
  - Need to say ‘**struct <name>**’ in C
- Example:

struct.cpp

```
struct Date { int d, m, y; };

int main( int argc, char* argv[] )
{
  struct Date dob = { 1, 4, 1990 };
  printf( “DOB: %02d/%02d/%04d\n”,
          dob.d, dob.m, dob.y );
  dob.d = 2;
  return 0;
}
```

Creates a **struct** on the stack
Note: no ‘**new**’ operator is used!!!

Accessing members of a **`struct`**

- Use the . operator to access members
  - Exactly as for Java classes
- Example:

struct.cpp

```
struct Date { int d, m, y; };

int main( int argc, char* argv[] )
{
  struct Date dob = { 1, 4, 1990 };
  printf( “DOB: %02d/%02d/%04d\n”,
          dob.d, dob.m, dob.y );
  dob.d = 2;
  return 0;
}
```

Initialisation
Like an array

Access values

`struct`s act like any other type

- Once defined, you can use **`struct`**s as any other type
- You can take the address of a variable of type **`struct`** and store it in a **`struct`** pointer, e.g.

```
struct Date* pDob = &dob;
```

  - Note: C++ does not need this ‘struct’ keyword
- You can embed a **`struct`** as a member of another **`struct`**
- You can create an array of **`struct`**s
- You can ask for the **`sizeof()`** a **`struct`**

Creating an initialised struct Date

```
struct Date { char d, m; short y; };
```

```
Date singleDate = { 1, 2, 2000 };
```

```
printf(
"Initialised singleDate is:%02d/%02d/%04d\n",
          singleDate.d, singleDate.m,
          singleDate.y );
```

Creating an initialised `struct Date`

```
struct Date { char d, m; short y; };
```

1) Define the type ‘struct Date’

```
Date singleDate = { 1, 2, 2000 };
```

2) Create and initialise a variable of type ‘struct Date’

```
printf(
"Initialised singleDate is:%02d/%02d/%04d\n",
          singleDate.d, singleDate.m,
          singleDate.y );
```

```
Initialised singleDate is : 01/02/2000
```

Array of structs (on the stack)

```
Date arrayOfDatesOnStack[5];

for ( i=0 ; i < 5 ; i++ )
   printf(
      "arrayOfDatesOnStack[%d] is : %02d/%02d/%04d\n",
      i,
      arrayOfDatesOnStack[i].d,
      arrayOfDatesOnStack[i].m,
      arrayOfDatesOnStack[i].y );
```

Array of 5 elements

Array of structs (on the stack)

```
Date arrayOfDatesOnStack[5];

for ( i=0 ; i < 5 ; i++ )
   printf(
      "arrayOfDatesOnStack[%d] is : %02d/%02d/%04d\n",
      i,
      arrayOfDatesOnStack[i].d,
      arrayOfDatesOnStack[i].m,
      arrayOfDatesOnStack[i].y );
```

```
arrayOfDatesOnStack[0] is : 00/00/0000
arrayOfDatesOnStack[1] is : 02/00/0000
arrayOfDatesOnStack[2] is : -104/-51/0034
arrayOfDatesOnStack[3] is : -41/53/24833
arrayOfDatesOnStack[4] is : -71/-74/24854
```

Values are uninitialised!!!

Array of dates (on the stack)

```
/* Uses array initialiser and struct initialiser */
Date initArrOfDatesOnStack[] = {
      {1,1,2001}, {2,2,2002}, {3,3,2003},
      {4,4,2004}, {5,5,2005} };

for ( i=0 ; i < 5 ; i++ )
      printf(
"initArrayOfDatesOnStack[%d] is : %02d/%02d/%04d\n",
            i, initArrayOfDatesOnStack[i].d,
            initArrayOfDatesOnStack[i].m,
            initArrayOfDatesOnStack[i].y );
```

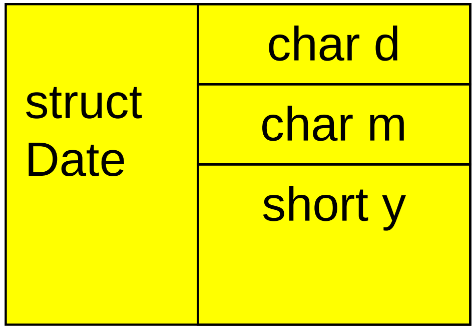
```
initalisedArrayOfDatesOnStack[0] is : 01/01/2001
initalisedArrayOfDatesOnStack[1] is : 02/02/2002
initialisedArrayOfDatesOnStack[2] is : 03/03/2003
initialisedArrayOfDatesOnStack[3] is : 04/04/2004
initialisedArrayOfDatesOnStack[4] is : 05/05/2005
```

Position of data

- Like arrays, the positions of the members inside a `struct` ***are*** known
- Elements will be placed sequentially in memory, in the order they are defined in the structure (sometimes this matters)
- So you **CAN** use the ordering to determine where parts will be in memory
- More on sizeof(structs), and positions in a struct later

Arrays of `structs`

```
struct Date
{
   char d, m;
   short y;
}
```



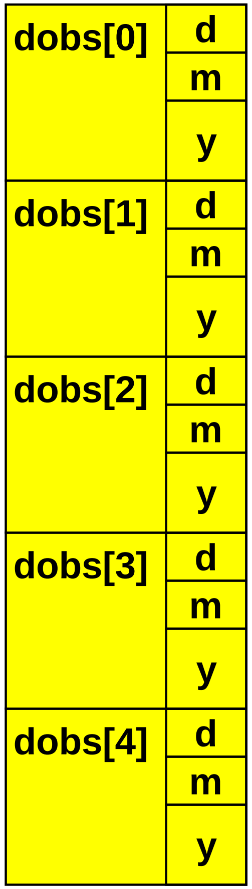
struct Date

char d

char m

short y

```
Date dobs[5];
```



dobs[0] d m y

dobs[1] d m y

dobs[2] d m y

dobs[3] d m y

dobs[4] d m y

Notes:

Syntax is the same as for arrays of basic types, e.g. int

Elements are one after another in memory (like other arrays)

Passing **`structs`** into functions

```
struct Date dob = {1, 4, 1990};
```

- Either pass the struct
  - A **(bit-wise) copy** of the **`struct`** is put on the stack
    - You can change this, using C++ copy constructor – see later
  - Any changes made inside the function affect the **copy**

```
void foo(struct Date dob) { dob.m = 3; }
foo( dob );
```

Use . to access struct members

- Or a pointer to the **`struct`**
  - A **copy of the pointer** is put on the stack
  - You can use the pointer to access the original copy

```
void bar(struct Date* pdob){(*pdob).m =3;}
bar( &dob );
```

For a pointer you could use **`(*pdob).m`**

X->Y means (*X).Y

```
struct time { int hour, minute, second; };

struct time t;
t.hour = 12;
t.minute = 34;
t.second = 14;

struct time* pt = &t;
pt->hour = 11;        /* = (*pt).hour     */
pt->minute = 13;      /* = (*pt).minute   */
pt->second = 5;       /* = (*pt).second   */

printf( "The time is %02d:%02d:%02d\n",
     t.hour, t.minute, t.second );
```

The return statement

- Functions can return only ONE value
- **The returned value is copied!**
- The value may be:
  - a basic type (e.g. `int`)
  - a pointer (or C++ reference, see later)
    - The address is copied (same for references)
  - a struct, union or object (C++ only)
    - The struct, union, object etc is copied
- May create a temporary variable in calling function, to store the returned value

Stack reminder

These **`struct`**s were created on the stack (i.e. as local variables)

Remember:

Data on the stack vanishes when the stack frame that contains it is removed from the stack

- i.e. when the function/block in which it is defined ends
- Do not return a pointer to one of these!

unions

Treating something as
“one thing OR another”

Very rarely used compared with structs
usually for low-level (e.g. o/s) code

Unions

- `union`s are very similar to `struct`s **except** that the data members are in the same place

- In `struct`s data members are one after another in memory (possibly with gaps)
- In `union`s data members all have the same address

- i.e. data is of one type OR another, not both

Unions

- Elements of unions are in the SAME place
- Elements of unions may be different sizes
  - **A union is as big as the biggest thing in it** (plus any packing)
- Unions are a way of providing different ways of looking at the same memory

```
union charorlong
{
  unsigned long ul;
  char ar[8];
};
```

Size 4?

Size 8

Addr:	ul	ar
1000	ul	[0]
1001		[1]
1002		[2]
1003		[3]
1004	■	[4]
1005	■	[5]
1006	■	[6]
1007	■	[7]

Bitfields and typedef

Bit fields

- Within structs you can specify fields with size less than a byte

```
struct position
{
   unsigned char x : 3; /* 3 bits */
   unsigned char y : 3; /* 3 bits */
   unsigned char z : 2; /* 2 bits */
};
```

- Which order the bits appear in the bytes is undefined (i.e. it could be high bits first, but could be low bits first, so ***bit order is implementation dependent***)
- No faster at runtime that using a **`char/int`** and the bitwise operators (**&**, **|**, etc)

typedef

- **Declare** a new type name using **`typedef`**
- Usage:

```
typedef old_type new_name
```

- E.g.

```
typedef struct DATE
  { int d, m, y; } Date;
```

  - Code can then use type `Date` instead of `struct DATE`
- In C++ (**not C**) you can omit the keywords `struct, enum, union` anyway
  - Similar to an automatic typedef

Sizes and packing

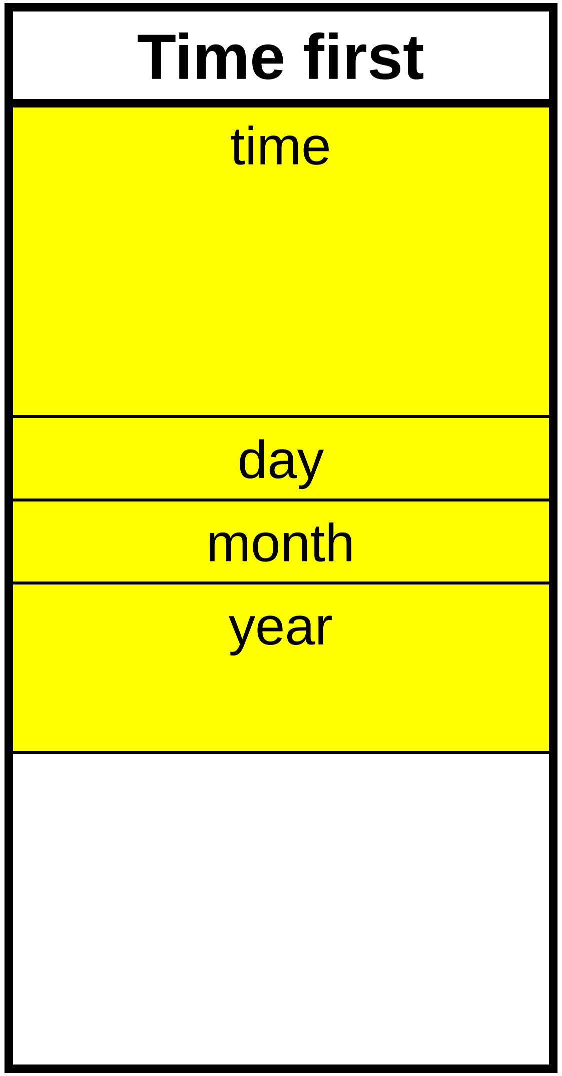
structs

```
struct DateTime
{
  int time;
  char day;
  char month;
  short year;
};

int main( int argc, char* argv[] )
{
  DateTime dt = { 80000, 01, 04, 1990 };

  printf( "DOB: %5d %02d/%02d/%04d\n",
           dt.time, dt.day, dt.month, dt.year );

  return 0;
}
```

struct content positions

```
struct DateTime {
int time; char day; char month; short year;
};

printf( "Address of dt       = %p, size %d\n",
          &dt, sizeof(dt) );
printf( "Address of dt.time  = %p, size %d\n",
          &(dt.time), sizeof(dt.time) );
printf( "Address of dt.day   = %p, size %d\n",
          &(dt.day), sizeof(dt.day) );
printf( "Address of dt.month = %p, size %d\n",
          &(dt.month), sizeof(dt.month) );
printf( "Address of dt.year  = %p, size %d\n",
          &(dt.year), sizeof(dt.year) );
```

Positions in memory



Time first

time

day

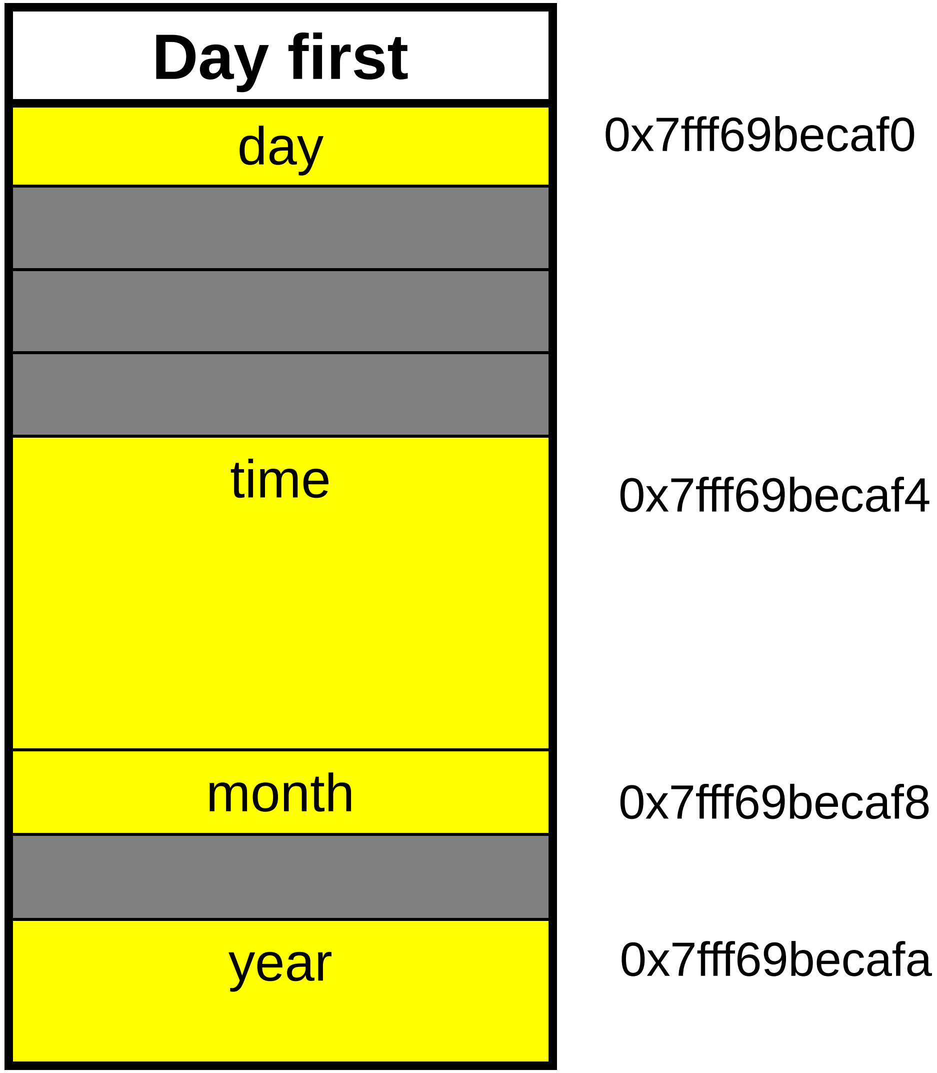
month

year

	Address	Size
dt	0x7fffaab18180	8
dt.time	0x7fffaab18180	4
dt.day	0x7fffaab18184	1
dt.month	0x7fffaab18185	1
dt.year	0x7fffaab18186	2

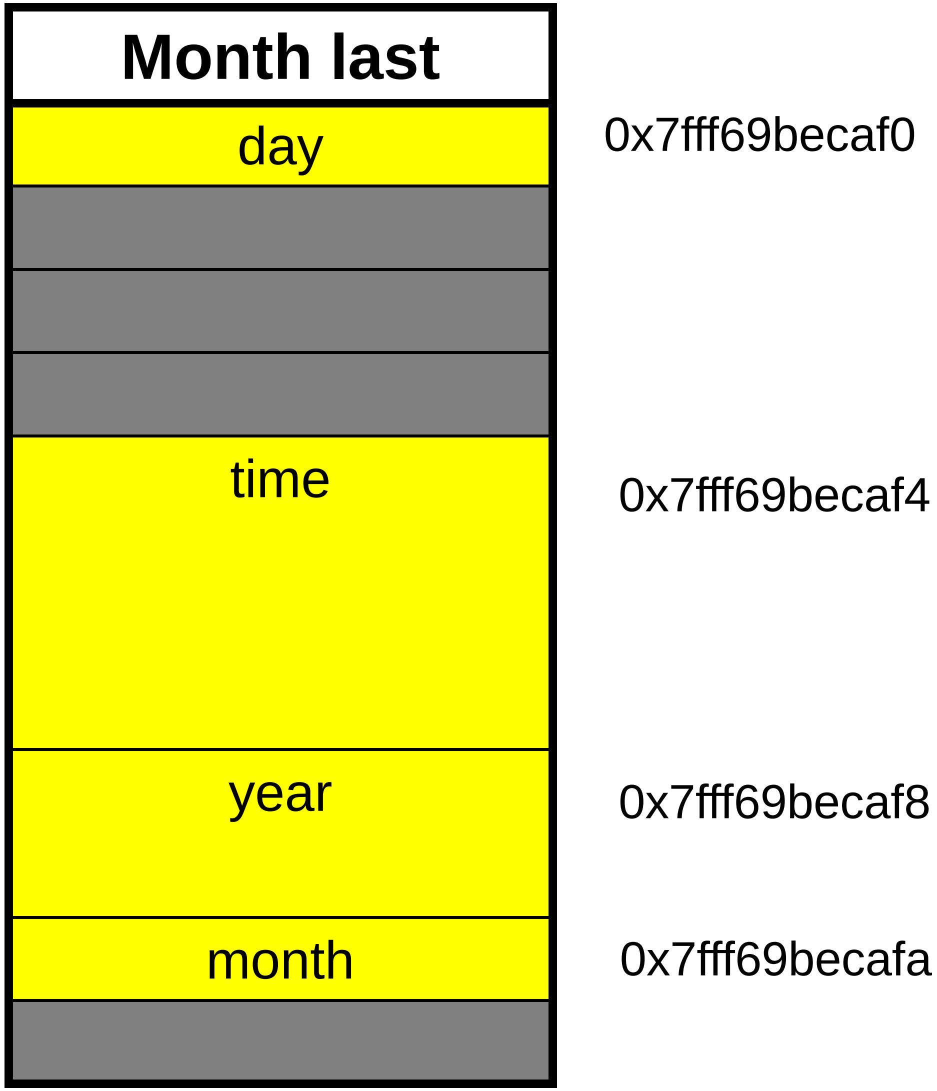
Gaps when day is first

Size of structure: 12



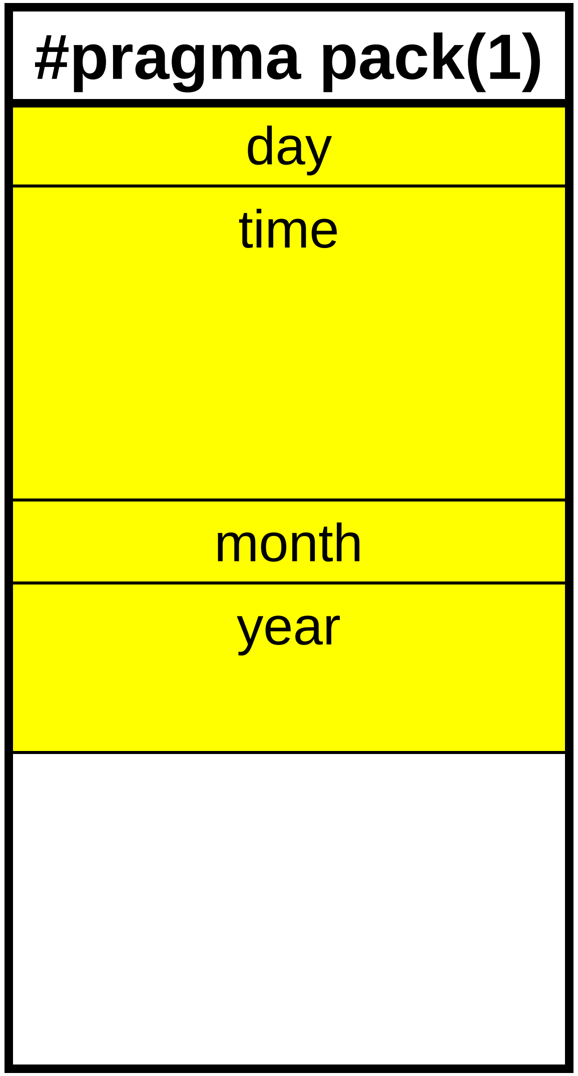
May have gaps at the end…

Size of structure: 12



Tell it to pack on 1 byte boundaries

	Address	Size
dt	0x7fff7e004280	8
dt.day	0x7fff7e004280	4
dt.time	0x7fff7e004281	1
dt.month	0x7fff7e004285	1
dt.year	0x7fff7e004286	2



#pragma pack(1)

Positions in memory

Time first	Day first	#pragma pack(1)
time	day	day
	▨	time
	▨	
	▨	
day	time	
month		month
year		year
	month	
	▨	
	year	

#pragma

- **`struct`**s may get empty space in them
- To align members for maximum speed
- You can usually tell compiler to pack structs
  - e.g. with gcc can use the command:

```
#pragma pack(1)
```

- **`#pragma`** means a compiler/operating system specific pre-processor directive

#pragma pack(1)

```
#include <cstdio>

struct A { int i; char c; };
union B { int i; char c; };

#pragma pack(1)
struct C { int i; char c; };
union D { int i; char c; };

int main( int argc, char** argv )
{
   printf( "sizeof(char): %d\n", sizeof(char) );
   printf( "sizeof(int): %d\n", sizeof(int) );
   printf( "sizeof(struct A): %d\n", sizeof(struct A) );
   printf( "sizeof(union B): %d\n", sizeof(union B) );
   printf( "sizeof(struct C): %d\n", sizeof(struct C) );
   printf( "sizeof(union D): %d\n", sizeof(union D) );
   return 0;
}
```

Example:

char	:	1
int	:	4
struct A	:	?
union B	:	?
struct C	:	?
union D	:	?

#pragma pack(1)

```
#include <cstdio>

struct A { int i; char c; };
union B { int i; char c; };

#pragma pack(1)
struct C { int i; char c; };
union D { int i; char c; };

int main( int argc, char** argv )
{
   printf( "sizeof(char): %d\n", sizeof(char) );
   printf( "sizeof(int): %d\n", sizeof(int) );
   printf( "sizeof(struct A): %d\n", sizeof(A) );
   printf( "sizeof(union B): %d\n", sizeof(B) );
   printf( "sizeof(struct C): %d\n", sizeof(C) );
   printf( "sizeof(union D): %d\n", sizeof(D) );
   return 0;
}
```

Example:

```
    char : 1
     int : 4
struct A : 8
 union B : 4
struct C : 5
 union D : 4
```

Sizes of unions and structs

If there is no excess space for packing:

- sizeof(struct) is **total** of the size of the members (i.e. **sum** of member sizes)
  - Members are one after another in memory
  - Bitfield structs use minimum number of bytes necessary
- sizeof(union) is **size of the largest member** (i.e. **maximum** of member sizes)
  - All members are in the same place
  - Largest member determines size

Next lecture

- Dynamic memory allocation
- Linked lists in C/C++