# Even More C++

## Outline

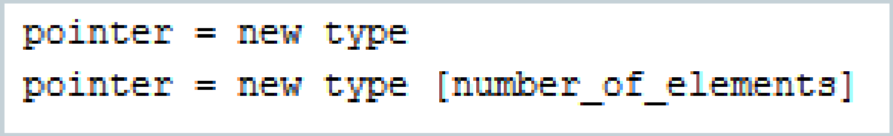
- Dynamic Memory
- Data Structures
- Other Data Types

## Dynamic Memory

- Fundamental data types take up a fixed size in memory
  - Memory can be allocated when the variables are declared
- There are times when memory size can only be determined at runtime
  - In these cases, programs need to dynamically allocate (and de-allocate) memory
    - This is done using the `new` and `delete` operators

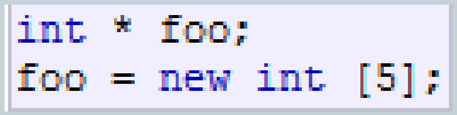
## new and new[]

- new is followed by a data type specifier and if there are multiple elements needed, brackets are used, to specify an array



```
pointer = new type
pointer = new type [number_of_elements]
```

- For example:



```
int * foo;
foo = new int [5];
```

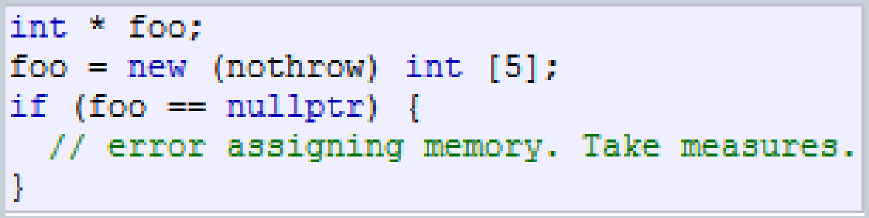
- In this example, a pointer to an integer is created, and then a block of memory is allocated to store 5 of them

## new and new[]

- So why not just create an array?
- Array size must be declared in one way or another at compile time
- Using dynamic memory assigns memory at runtime so you can use a variable memory size
- Memory is allocated at runtime from the heap
  - There is no guarantee that there is enough memory to handle a given request

# Checking for Allocation Success

- By default, C++ will throw an exception if something went wrong with memory allocation
  - In this case, the program will terminate if the exception is not handled (unlike Java which either requires you to “catch” an exception or duck it
- You can tell C++ not to throw an exception and then deal with it in your own code:

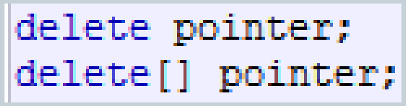


```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

- Using exceptions is more efficient – we will talk about those later

## delete and delete[]

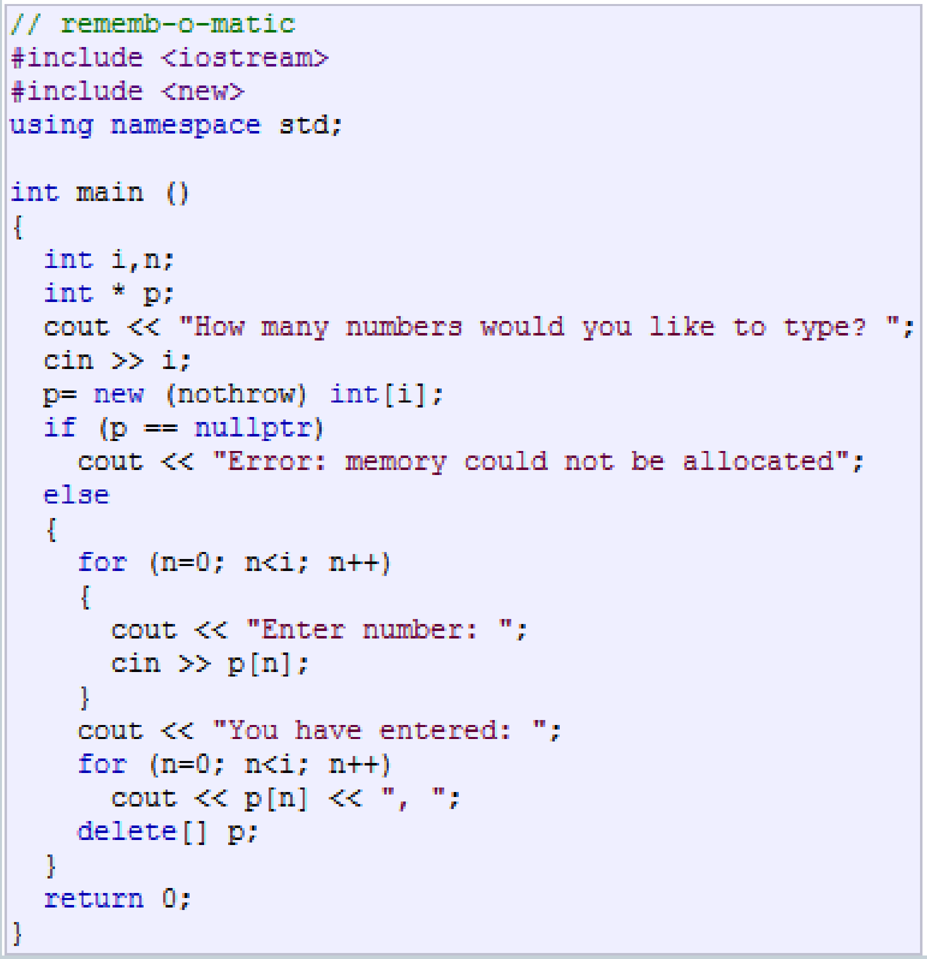
- C++ does not handle garbage collection for you
  - You need to determine when a particular data item is no longer needed and then remove it
  - Use delete and delete[] to do this



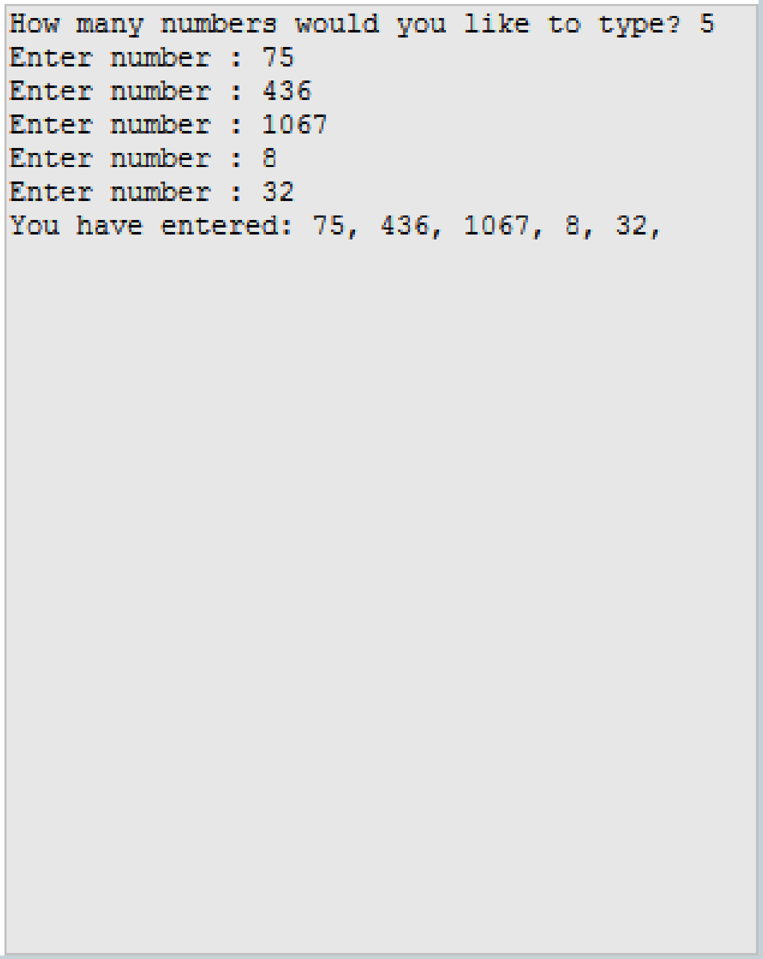
```
delete pointer;
delete[] pointer;
```

  - The “thing” deleted should be either something that was created with new or new[] before, or it should be a null pointer (in which case nothing happens)

## An Example



```cpp
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```



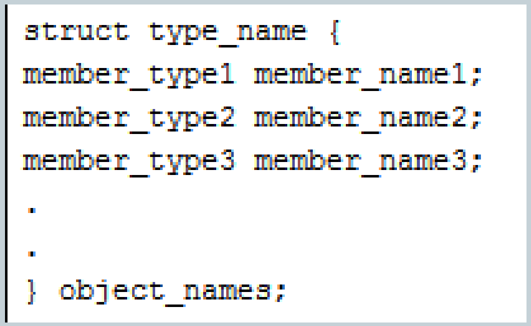
```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

## Dynamic Memory in C

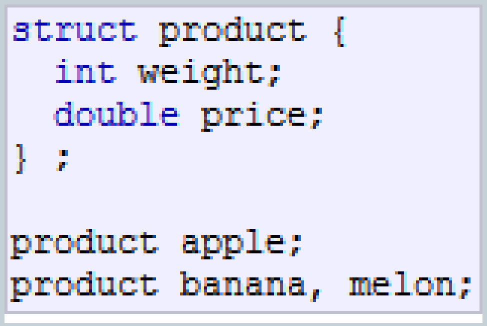
- C++ uses new and delete to allocate and free memory
- C uses malloc, calloc, realloc and free
- Since C++ is built on C, you can still use these functions, but you should not mix them
  - if you use new on an item, deallocate it with delete
  - if you use malloc, calloc or realloc, deallocate is with free

## Data Structures

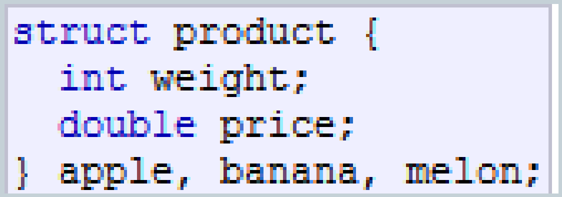
- A data structure is a group of data elements grouped together under one name
  - Not quite the same thing as a data type in Java
- Use struct to define a structure in C++



```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```



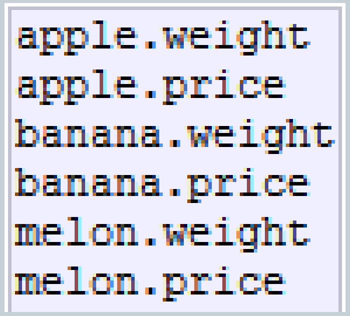
```
struct product {
  int weight;
  double price;
} ;

product apple;
product banana, melon;
```



```
struct product {
  int weight;
  double price;
} apple, banana, melon;
```

## Data Structures

- Accessing data elements in a structure



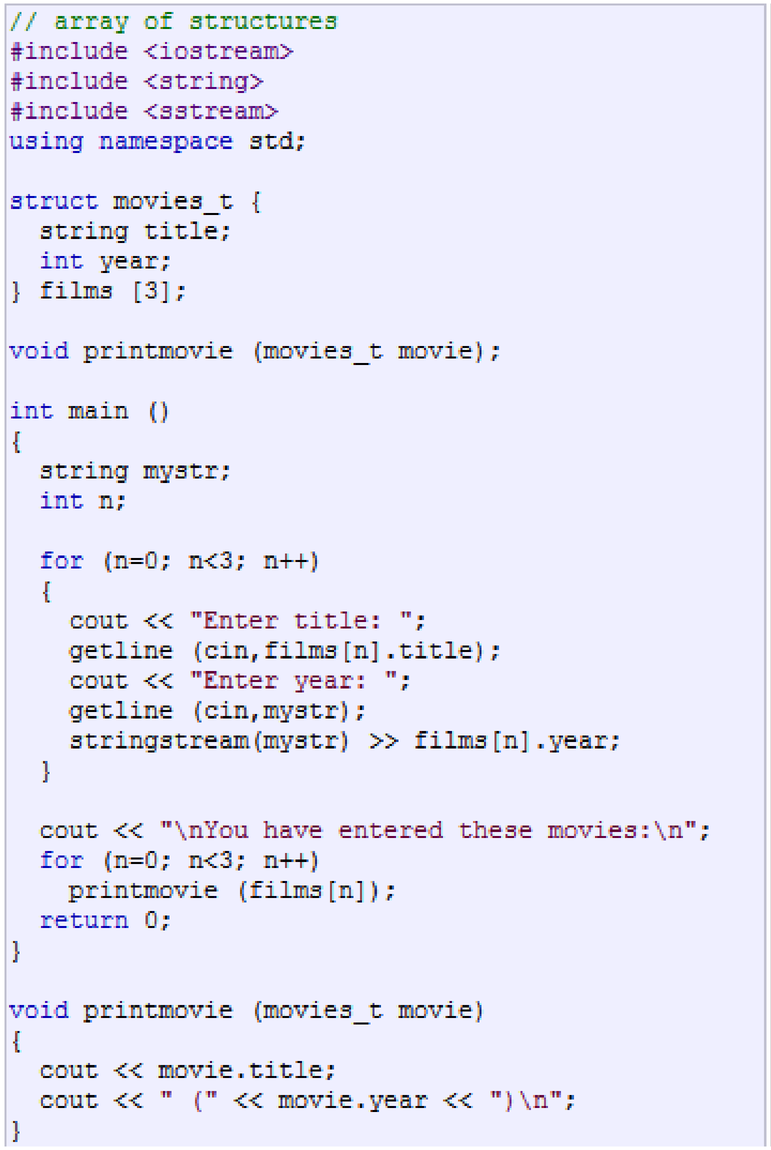
```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

# An Example of struct

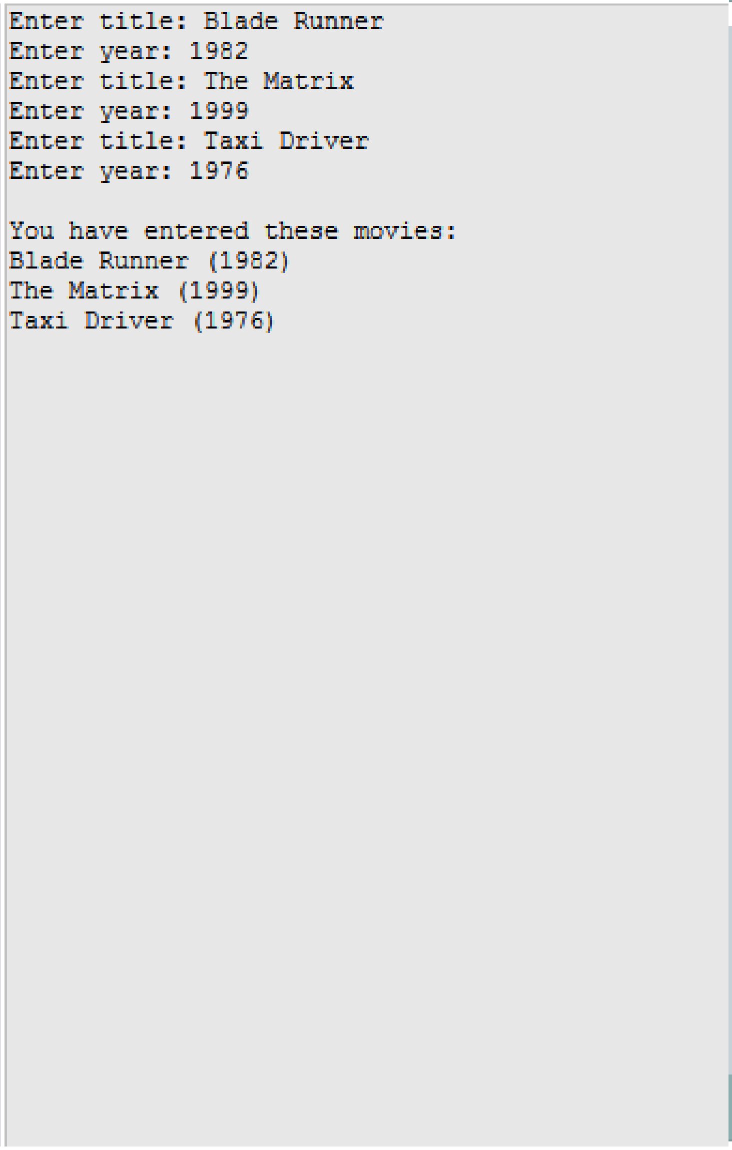
```cpp
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
  string mystr;

  mine.title = "2001 A Space Odyssey";
  mine.year = 1968;

  cout << "Enter title: ";
  getline (cin,yours.title);
  cout << "Enter year: ";
  getline (cin,mystr);
  stringstream(mystr) >> yours.year;

  cout << "My favorite movie is:\n ";
  printmovie (mine);
  cout << "And yours is:\n ";
  printmovie (yours);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

```
Enter title: Alien
Enter year: 1979

My favorite movie is:
 2001 A Space Odyssey (1968)
And yours is:
 Alien (1979)
```

# An Example of an Array of structs



```
// array of structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} films [3];

void printmovie (movies_t movie);

int main ()
{
  string mystr;
  int n;

  for (n=0; n<3; n++)
  {
    cout << "Enter title: ";
    getline (cin,films[n].title);
    cout << "Enter year: ";
    getline (cin,mystr);
    stringstream(mystr) >> films[n].year;
  }

  cout << "\nYou have entered these movies:\n";
  for (n=0; n<3; n++)
    printmovie (films[n]);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```



```
Enter title: Blade Runner
Enter year: 1982
Enter title: The Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

You have entered these movies:
Blade Runner (1982)
The Matrix (1999)
Taxi Driver (1976)
```

# Pointers to Structures

- The arrow operator -> is used to access structures that have member elements

```
struct movies_t {
  string title;
  int year;
};

movies_t amovie;
movies_t * pmovie;
```

```
pmovie = &amovie;
```

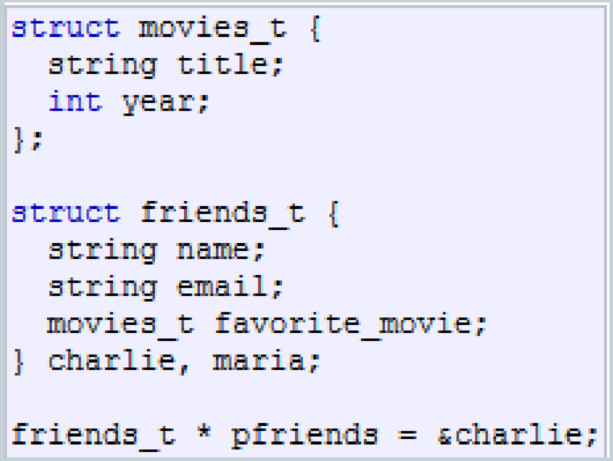
`pmovie->title` is equivalent to: `(*pmovie).title`

`*pmovie.title` is equivalent to: `*(pmovie.title)`

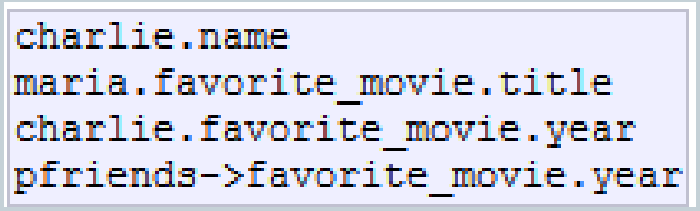
| Expression | What is evaluated | Equivalent |
|---|---|---|
| `a.b` | Member `b` of object `a` | |
| `a->b` | Member `b` of object pointed to by `a` | `(*a).b` |
| `*a.b` | Value pointed to by member `b` of object `a` | `*(a.b)` |

## Nesting Structures

- Structures can be nested within other structures



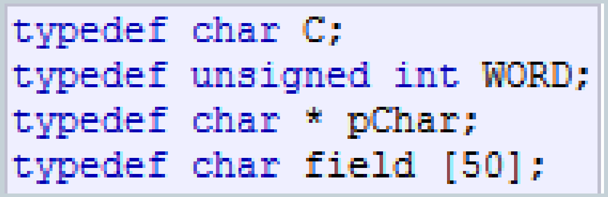
```
struct movies_t {
  string title;
  int year;
};

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
} charlie, maria;

friends_t * pfriends = &charlie;
```



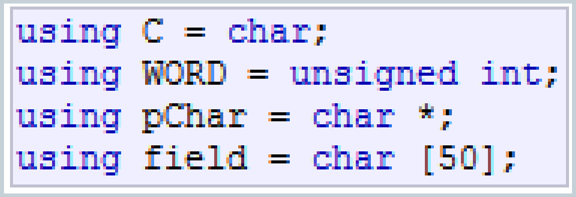
```
charlie.name
maria.favorite_movie.title
charlie.favorite_movie.year
pfriends->favorite_movie.year
```

# Other Data Types: Aliases

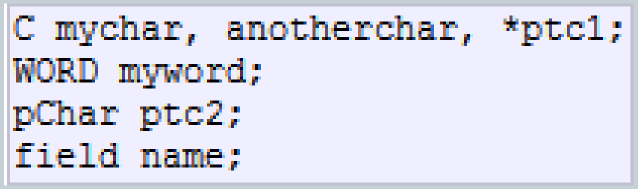
- Two ways to create a type alias:
  - `typedef existing_type new_type_name;`
  - `using new_type_name = existing_type;`



```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```



```
using C = char;
using WORD = unsigned int;
using pChar = char *;
using field = char [50];
```

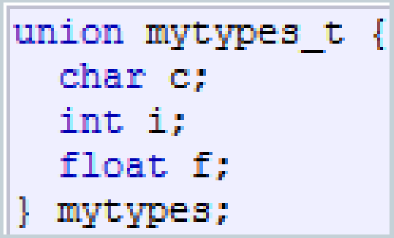


```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

- "using" is more generic, but "typedef" is likely found more often in existing code

## Other Data Types: Unions

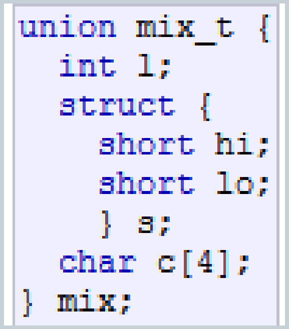
- Declaration similar to struct, but meaning is very different



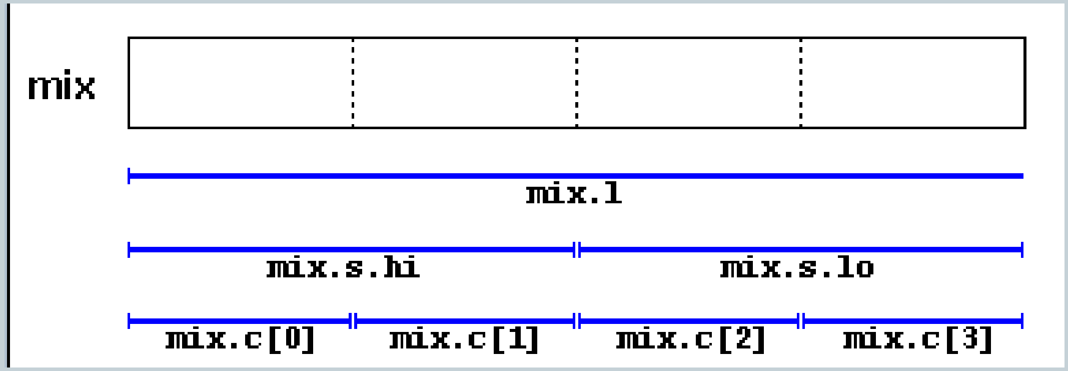
```
union mytypes_t {
  char c;
  int i;
  float f;
} mytypes;
```

- All three member elements use the same memory space
  - Memory allocated is the size of the largest
    - In the example above, probably the size of a float
- Use this when you want to access an element in its entirety or as an array of smaller elements

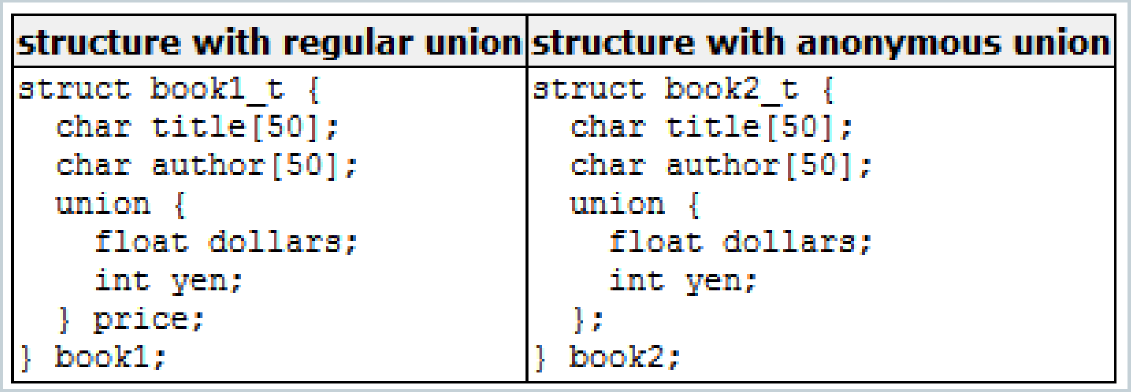
# Union Example



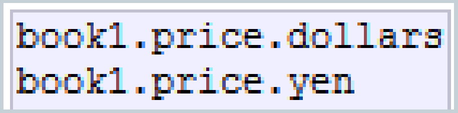
```
union mix_t {
   int l;
   struct {
      short hi;
      short lo;
      } s;
   char c[4];
} mix;
```



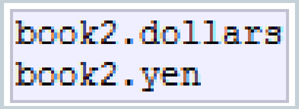
# Anonymous Unions



| structure with regular union | structure with anonymous union |
|---|---|
| `struct book1_t {` | `struct book2_t {` |
| `  char title[50];` | `  char title[50];` |
| `  char author[50];` | `  char author[50];` |
| `  union {` | `  union {` |
| `    float dollars;` | `    float dollars;` |
| `    int yen;` | `    int yen;` |
| `  } price;` | `  };` |
| `} book1;` | `} book2;` |

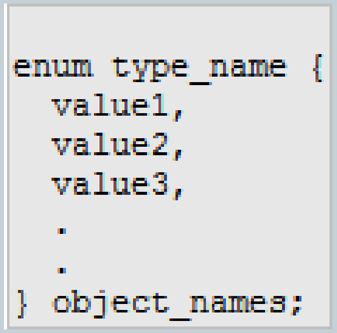


```
book1.price.dollars
book1.price.yen
```

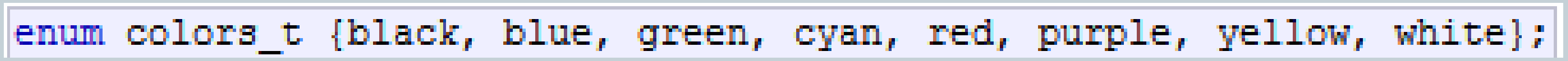


```
book2.dollars
book2.yen
```

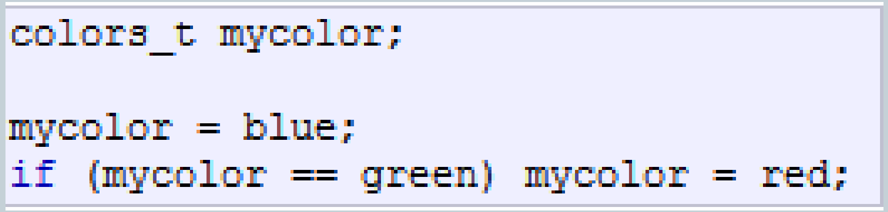
## Enumerated Types (enum)



```
enum type_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```



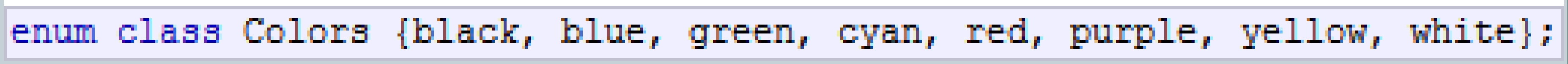
```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```



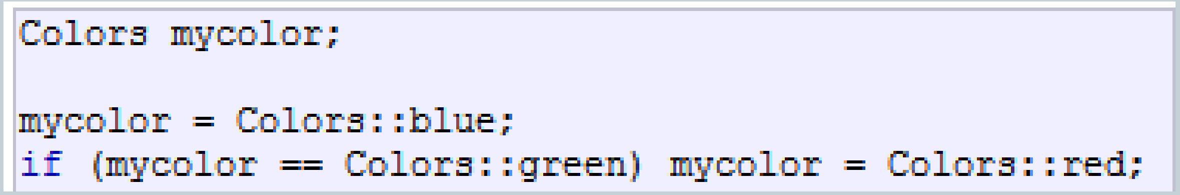
```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

- You can use these names or their integer equivalents

## Enumerated Types (enum class)



```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};
```



```
Colors mycolor;

mycolor = Colors::blue;
if (mycolor == Colors::green) mycolor = Colors::red;
```

## Summary

- Dynamic Memory
- Data Structures
- Other Data Types