ENGR101: Lecture 5

Structuring program data. Conditional. Iterations

ECS, VUW

March, 2023

What we cover today?

1. Structuring program data
2. Conditional
3. Iterations

Types - reminder. Limiting.

We mentioned that all variables should be assigned **type**.
Type describes how many bytes variable takes when stored in memory.
Type allows to catch a lot of mistakes in code.
But if we are limited to build-in types (**int,char,double...**) and arrays made out of them - not impossible to program using only these but code will be bulky and error-prone.

Customizing types to the task

When programming we deal with real-world objects: book, pixel, car for sale, customer, shopping item, etc.
As an example, we want to write library database.
Each book can be described by:

- Title (string)
- Author (string)
- Number of pages (int)
- Available (bool)

And we have many of these books.
Note: **string** is an array of characters and it is C++ type. If you need assign value to string - it should be enclosed in double quotation marks. To use this type - put **#include** <**string**> at the beginning of the program.

Naive approach

Without much thinking, we write

Listing 1: Caption

```
int main(){
    std::string titles[500];
    std::string authors[500];
    bool available[500];
    int num_pages[500];

    titles[44]="Adventures␣in␣C++␣land";
    authors[44] = "me␣and␣myself";
}
```

Not very nice approach - you have to watch index of arrays carefully.

Better approach - group your variables together

Better approach - group variables for one book together.

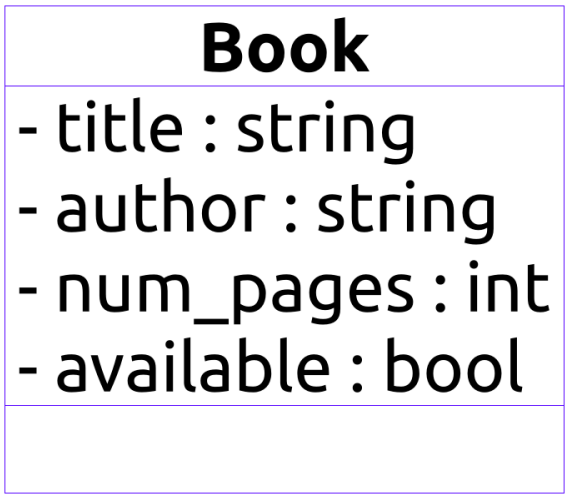


Figure: Struct

Listing 2: struct

```
struct Book{
  std::string title;
  std::string author;
  int num_pages;
  bool available;
};
```

Simplest way to do that in C++ - use **struct**.
Each variable grouped together is called **member** of the **struct**.

Variable of custom type: declaration

Listing 3: struct variable

```
struct Book{
  std::string title;
  std::string author;
  int num_pages;
  bool available;
};
int main(){
  Book book1;
}
```

Once we created custom type (type **Book** in this case), we can declare variable of this type (**book**).
Convention: type names start with capital, variable names - with lowercase.

Access to **struct members**

Listing 4: struct variable

```
struct Book{
  std::string title;
  std::string author;
  int num_pages;
  bool available;
};
int main(){
  Book book1;
  Book book2;
  book1.num_pages = 45;
  book2.num_pages = 12345;
}
```

We need to set/get values of **struct members**. To do that use variable name (**book1**, for example) followed by dot and member name. It sets member value only for this particular **struct** type variable.
book1.num_pages = 45; sets **num_pages** only for **book1**.

struct makes code more compact

We can use **struct** as an argument for function

Listing 5: struct variable as function argument

```
using namespace std;
\\ declaration of Book type as
\\ per previous slides
void print_book(Book book){
  cout<<"title:"<<book.title<<endl;
  cout<<"author:"<<book.author<<endl;
  cout<<"num_pages:"<<book.num_pages<<endl;
  cout<<"available:"<<book.available<<endl;
}

int main(){
  Book book1;
  book1.num_pages = 45;
  print_book(book1);
}
```

We can pass variable of our custom type as an argument to the function: we can use **print_book(book1)** instead of listing all four members of the **struct**.

Function as a member of **struct**

Listing 6: function as a struct member

```
struct Book{
  std::string title;
  std::string author;
  int num_pages;
  bool available;
  void print_book(); // member function
};

void Book::print_book(){
  std::cout<<"title:"<<title<<std::endl;
  std::cout<<"author:"<<author<<std::endl;
}

int main(){
  Book book1;
  book1.print_book();
}
```

Function can be made a member of **struct**.
In this case when **book1.print_book();** is called, function will use member values of **book1** variable.
Note: it is only C++ option.

Note: You can notice that it looks similar to **class**. **struct** is simple version of class with all members "public".

Question (hint - reference vs value function arguments)

We want set pages of the Book type variable inside the function:

Listing 7: "Does it work?"

```
#include <iostream>
struct Book{
  std::string title;
  std::string author;
  int num_pages;
  bool available;
  void print_book(); // member function
};

void set_pages(Book b,int pages){
  b.num_pages = pages;
}

int main(){
  Book book1;
  book1.num_pages = 9;
  std::cout<<" pages="<<book1.num_pages<<std::endl;
  set_pages(book1,45);
  std::cout<<" pages="<<book1.num_pages<<std::endl;
}
```

Does function **set_pages** work (does it change pages)?

- yes
- no

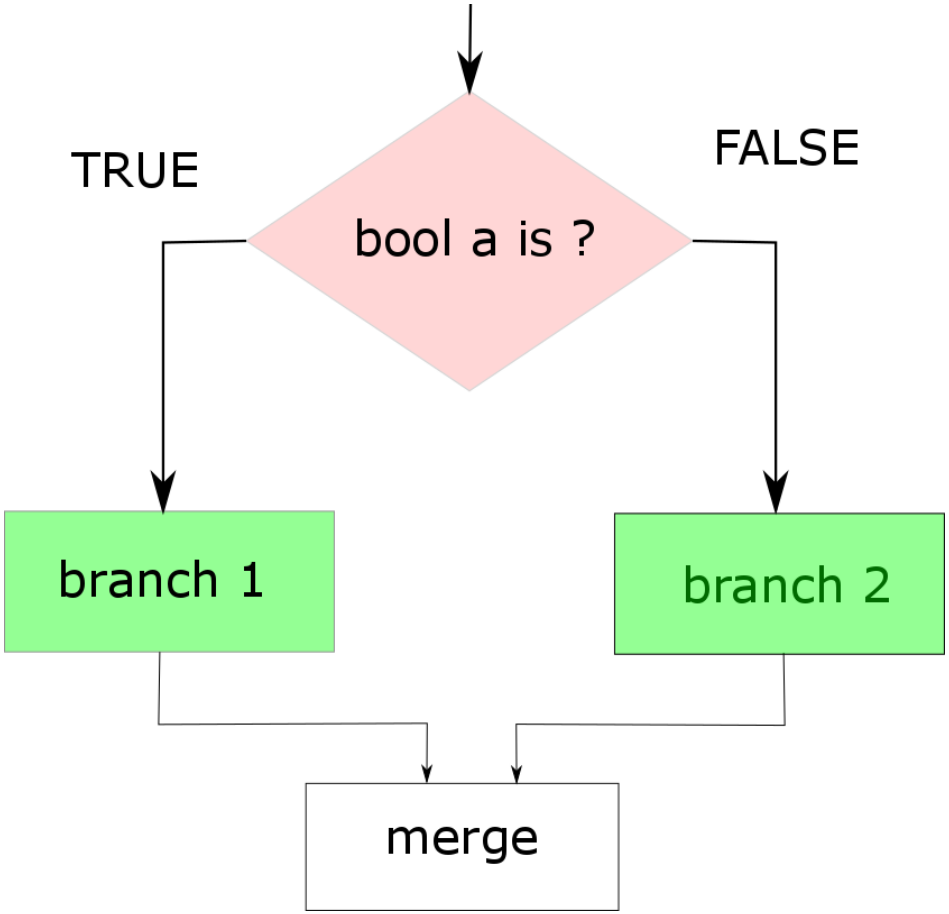
struct as result of a function

Listing 8: struct return

```
// define Book type
Book enter_book(){
  Book b;
  std::cin>>b.author;
  return b;
}

int main(){
   Book book1 = enter_book();
   return 0;
}
```

struct can be returned from the function. Declare variable of custom struct type inside the function, set member values, **return** it.
On line **book1 = enter_book();** memory contents of **b** (inside function memory area) will be copied over into memory for **book1**.

Conditional execution



More often than not you want your code do different things depending on condition.

Listing 9: this way

```
if (condition){
    // branch 1
} else {
    //branch 2
}
```

condition here is **bool** type variable (1 Byte).

And now - branching

Condition estimates to either true or false. To calculate the condition we can use following relational operators:

- == equal to: 3==3− > TRUE ; 4==3− >FALSE
- > greater than
- ! = not equal to
- >= greater than or equal to
- < less than
- <= less than or equal to

Listing 10: taking branch

```
#include <iostream>

int main(){
 int a;
 a = 3;
 if (a>2){
   std::cout<<"branch␣1";
 } else {
   std::cout<<"␣branch␣2";
 }
 return 0;
}
```

Code for conditional execution. **else** branch can be missed. Then nothing is happening if condition estimates to **false**.

Listing 11: combining conditions

```
#include <iostream>
int main(){
  int d = 9;
  bool a = d<5;
  bool b = d>2;
  std::cout<<"a = "<<a<<" b="<<b<<std::endl;
  std::cout<<"a AND b="<<(a&&b)<<std::endl;
  std::cout<<"a OR b="<<(a||b)<<std::endl;
  std::cout<<"NOT a="<<(!a)<<std::endl;
}
```

Several **conditions** can be combined using AND (TRUE if both arguments are TRUE) and OR (TRUE if at least one argument is TRUE) operators. Condition can be inverted (NOT operator).

Listing 12: combining conditions

```
#include <iostream>
int main(){
  int d = 9;
  bool a = d>5;
  bool b = d>2;
  std::cout<<"a = "<<a<<" b="<<b<<std::endl;
  std::cout<<"a AND b="<<(a&&b)<<std::endl;
  std::cout<<"a OR b="<<(a||b)<<std::endl;
  std::cout<<"NOT a="<<(!a)<<std::endl;
}
```

Several **conditions** can be combined using AND (TRUE if both arguments are TRUE) and OR (TRUE if at least one argument id TRUE) operators. Condition can be inverted (NOT operator).

ternary operator - conditional assignment shortcut

Listing 13: assignment shortcut

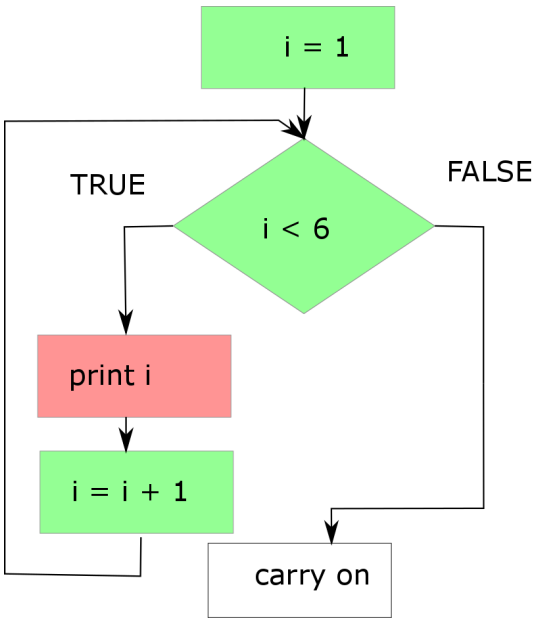
```
#include <iostream>

int main(){
  int a;
  a = 3;
  int b;
  b = (a>2)?45:34;
  std::cout<<b;
}
```

Very common situation is when you want to assign different values to the variable based on some condition. There is shortcut for that.

- if $a > 2$ is **true** then **b** becomes 45
- if $a > 2$ is **false** then **b** becomes 34

Iterations - by examples

Sometimes we need to repeat calculations several times.
Say, we want to print numbers from 1 to 6. We can go,
cout << 1; *cout* << 2...a lot of typing



There is shortcut for it. **for** operator:

Listing 14: for

```
for (int i = 1 ; i < 6 ; i= i+1) {
  cout<<i<<endl;
}
```

for value of i from that to this do that All shown in green logic is implemented by one line:

Listing 15: for

```
for (int i = 1 ; i < 6 ; i= i+1)
```

Question?

Listing 16: Caption

```
#include <iostream>

using namespace std;

int main(){
  for (int i = 0 ; i < 6 ; i=i+2){
        cout<<i<<" ";
  }
}
```

1. 0 2 4
2. 0 1 2 3 4 5
3. 0 1 2 3 4 5 6

Question, again?

Listing 17: Caption

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

int main(){
  for (int i = 10 ; i < 6 ; i=i+2){
        cout<<i<<"␣";
  }
}
```

1. 0 2 4
2.
3. 10 8 6 4 2 0

Question, again?

Listing 18: Caption

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

int main(){
  for (int i = 0 ; i < 6 ; i=i+1){
   if ( i > 2 ){
     cout<<i<<"␣";
   }
  }
}
```

1. 0 2 4
2. 3 4 5
3. 0 1 2 3 4 5

How to to work with an array of **int**s?

Usually you use **for()** to traverse array element indexes.

Listing 19: Caption

```
#include <iostream>
using namespace std;
int main(){
 int a[5];
 for ( int i = 0 ; i < 5;i = i + 1){
   a[i] = i*2;
 }

 for ( int i = 0 ; i < 5;i = i + 1){
  cout<<a[i]<<"␣";
 }
 return 0;
}
```

1. 0 2 4 6 8
2. 3 4 5
3. 0 1 2 3 4 5

That was a lot.
Questions?