IIT Bo

Computer Programming

Dr. Deepak B Phatak
Dr. Supratik Chakraborty
Department of Computer Science and Engineering
IIT Bombay

Session: Structures and Pointers – Part 1

Quick Recap of Relevant Topics

- Structures as collections of variables/arrays/other structures
- Statically declared variables/arrays of structure types
- Accessing members of structures
- Organization of main memory: locations and addresses
- Pointers to variables/arrays of basic data types

Overview of This Lecture

- Pointers to variables of structure types
- Accessing members of structures through pointers

Recall: Memory and Addresses

- Main memory is a sequence of physical storage locations
- Each location stores 1 byte (8 bits)

  Content/value of location
- Each physical memory location identified by a unique **address**
  - **Index in sequence of memory locations**

Address (in hexadecimal)

400	1 0 0 1 1 1 0 1
401	1 0 1 1 1 1 1 1
402	1 0 0 1 0 0 0 1
403	1 0 1 1 0 1 1 1
404	1 0 0 1 0 0 0 1
405	1 0 0 0 0 1 1 1
406	1 1 1 1 0 0 0 1
407	1 0 0 0 0 0 0 0
408	1 1 1 1 1 1 1 1
409	0 0 0 0 0 0 0 0
40a	1 1 1 1 0 0 0 0

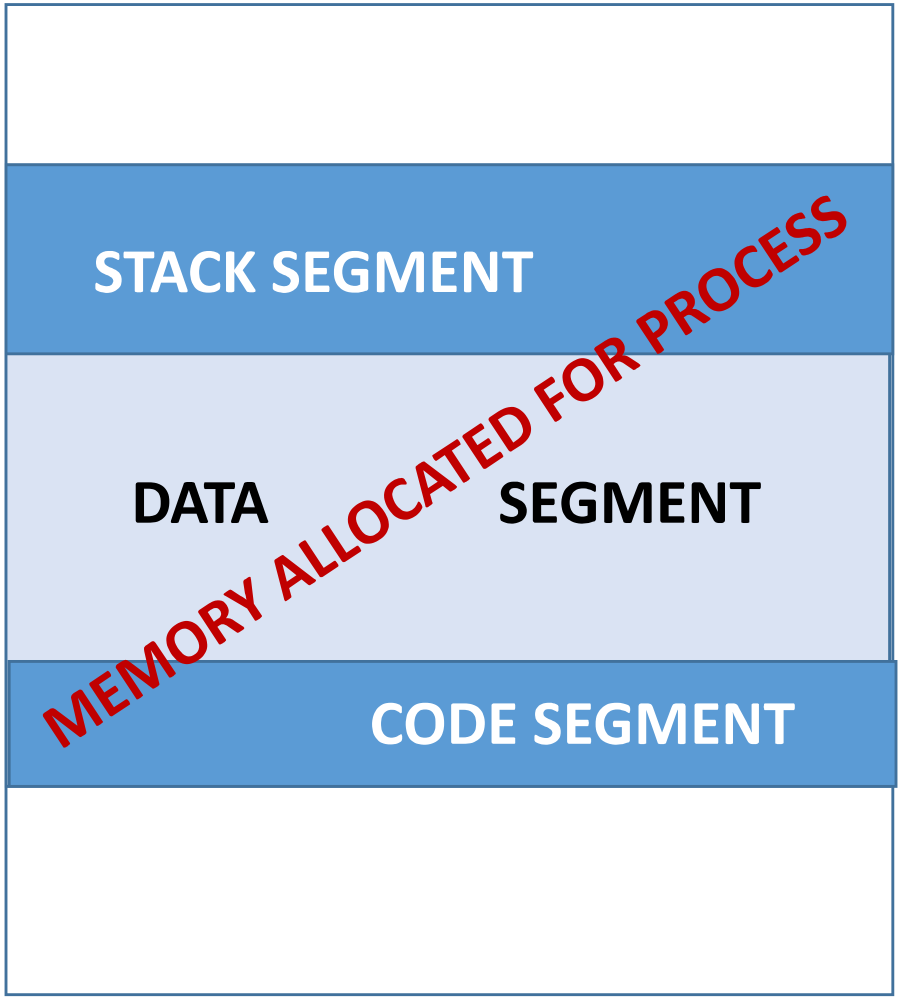
MAIN MEMORY

Memory for Executing a Program (Process)

- Operating system allocates a part of main memory for use by a process
- Divided into:

Code segment: Stores executable instructions in program

Data segment: For dynamically allocated data
Stack segment: Call stack



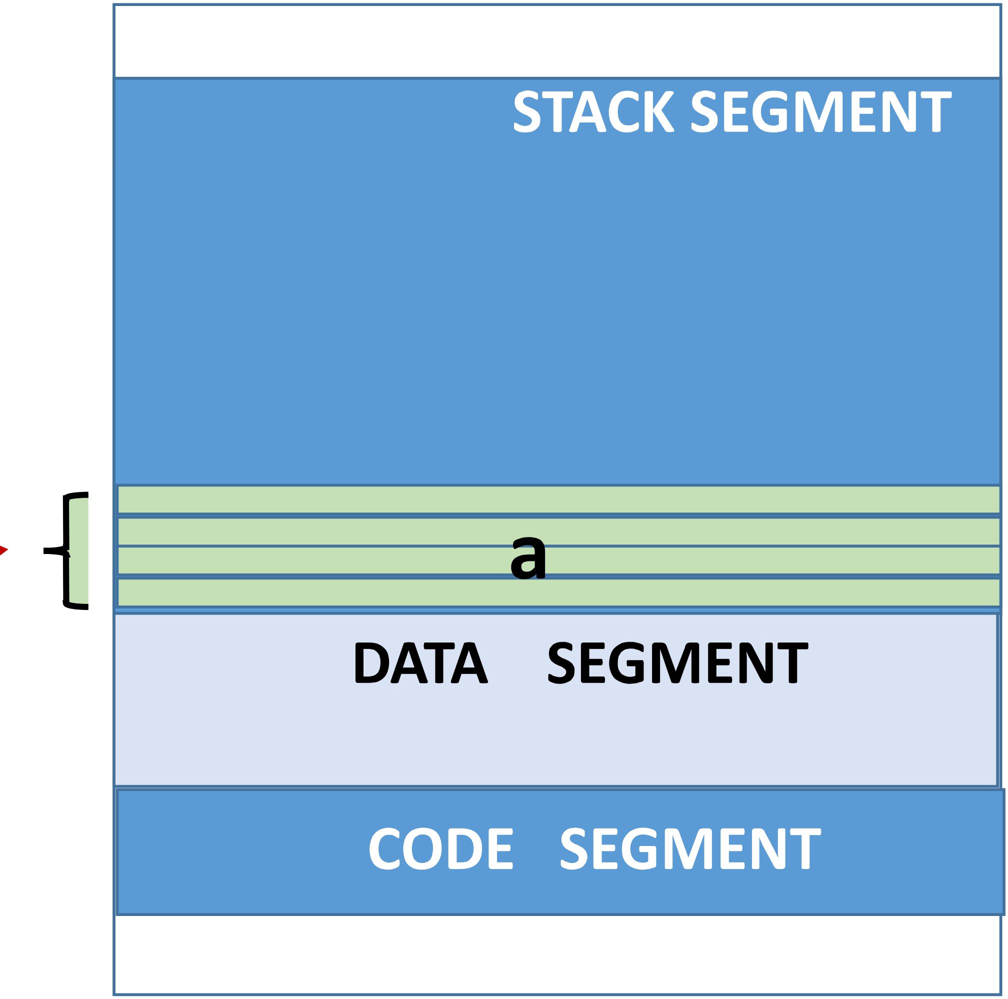
Structures in Main Memory

```
int main()
{
   struct MyStructType {
     char z;
     int x, y;
   };
  MyStructType p1;
  int a;
  … Rest of code …
  return 0;
}
```

Needs 4 bytes of storage

Structures in Main Memory

```
int main()
{
   struct MyStructType {
     char z;
     int x, y;
   };
  MyStructType p1;
  int a;
  … Rest of code …
  return 0;
}
```



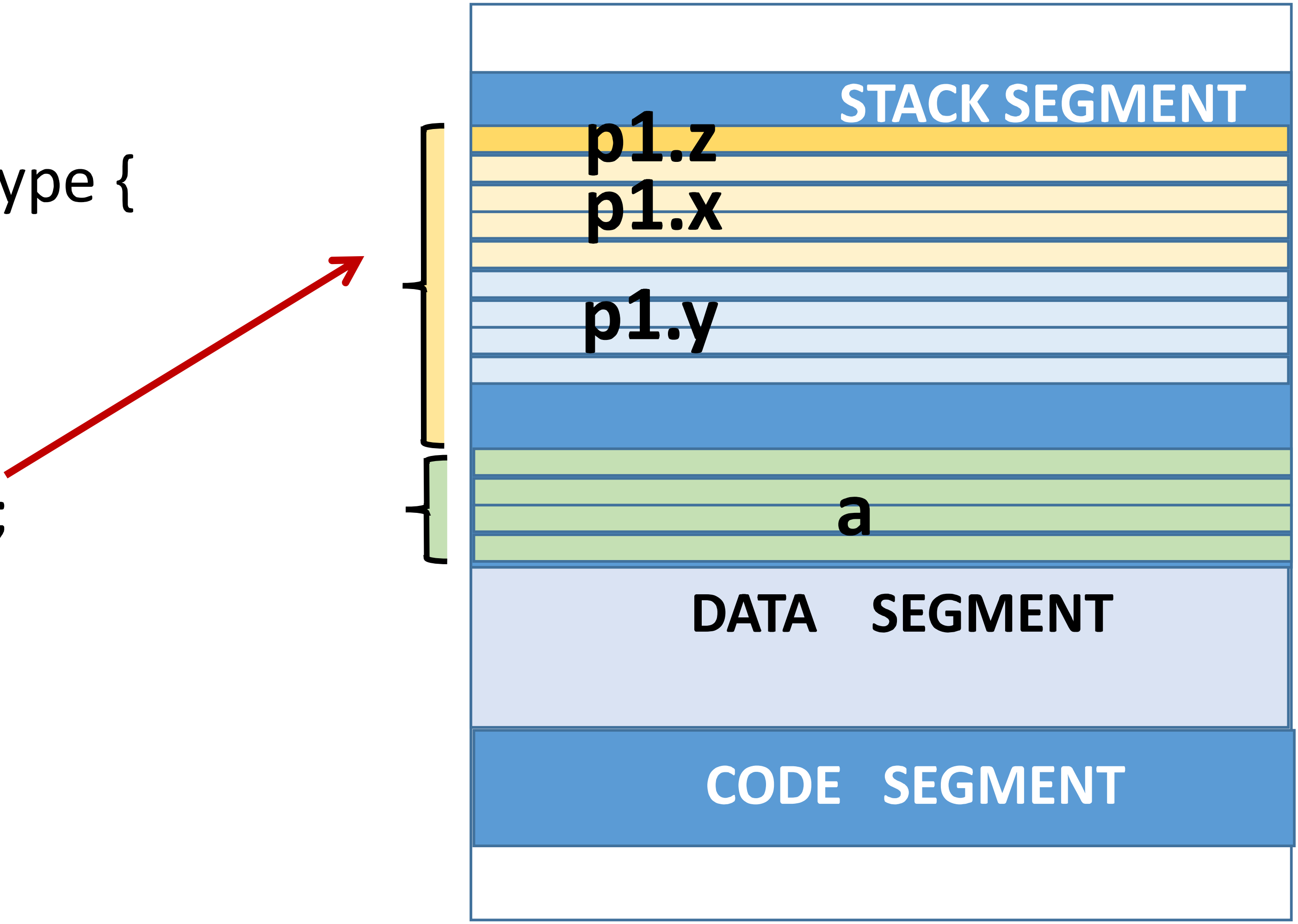
Structures in Main Memory

```
int main()
{
   struct MyStructType {
     char z;
     int x, y;
   };
  MyStructType p1;
  int a;
  … Rest of code …
  return 0;
}
```

Needs 1 + 4 + 4, i.e. 9 bytes of storage

Structures in Main Memory

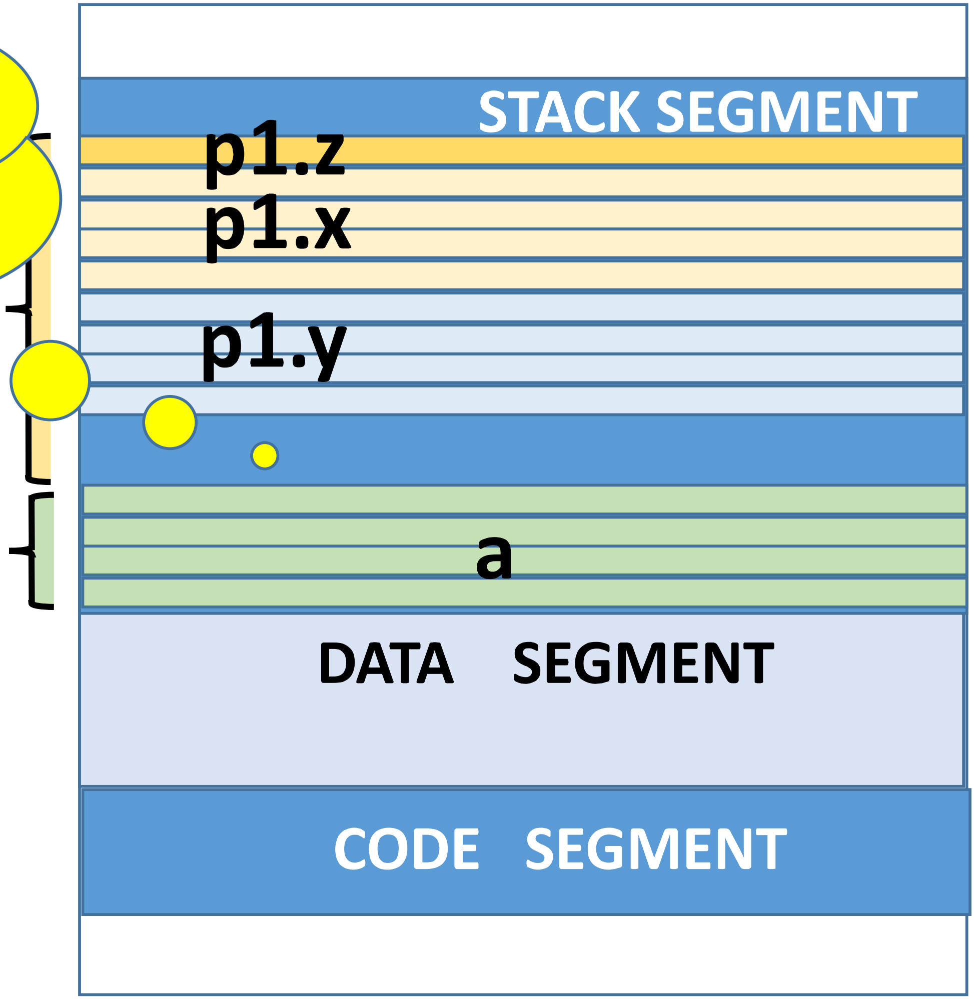
```
int main()
{
   struct MyStructType {
     char z;
     int x, y;
   };
  MyStructType p1;
  int a;
  ... Rest of code ...
  return 0;
}
```



Structures in Main Memory

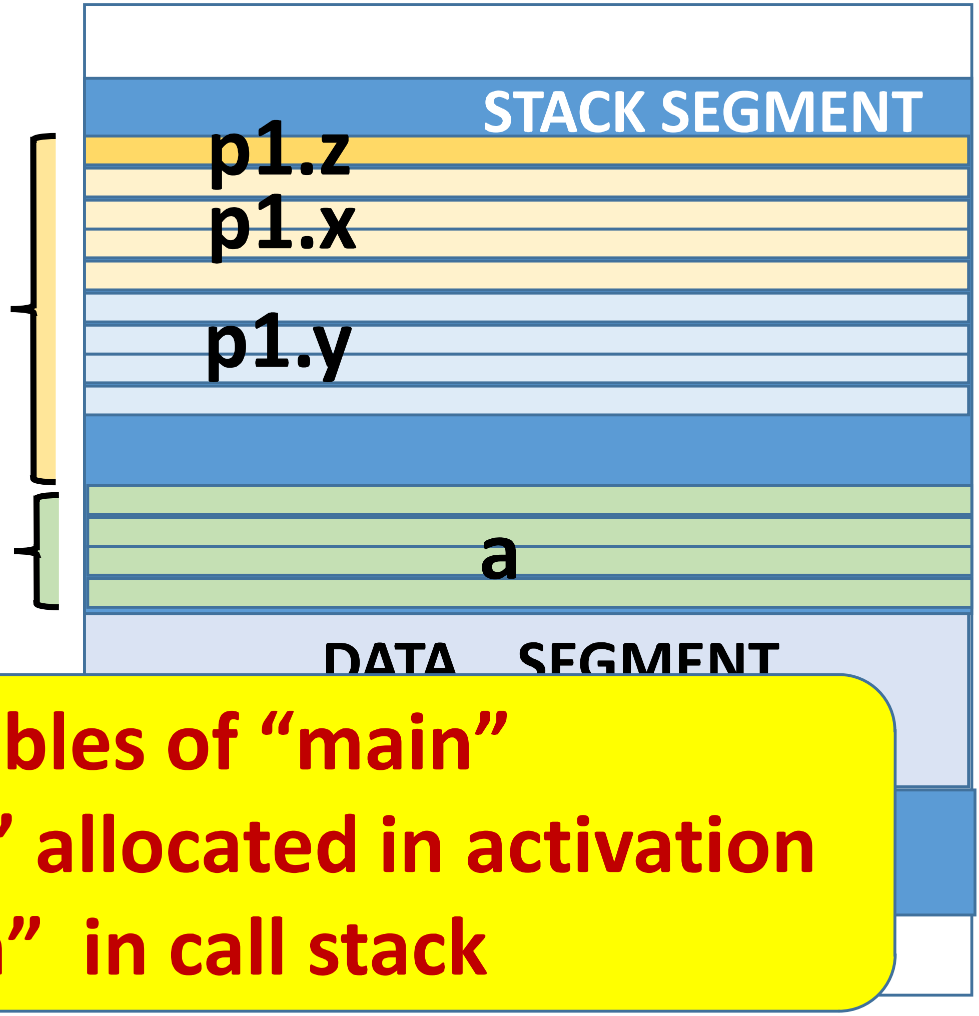
```
int main()
{

    int x;
  };
  MyStructType p1;
  int a;
  … Rest of code …
  return 0;
}
```

What is that gap/padding? Wait for a few slides



Structures in Main Memory

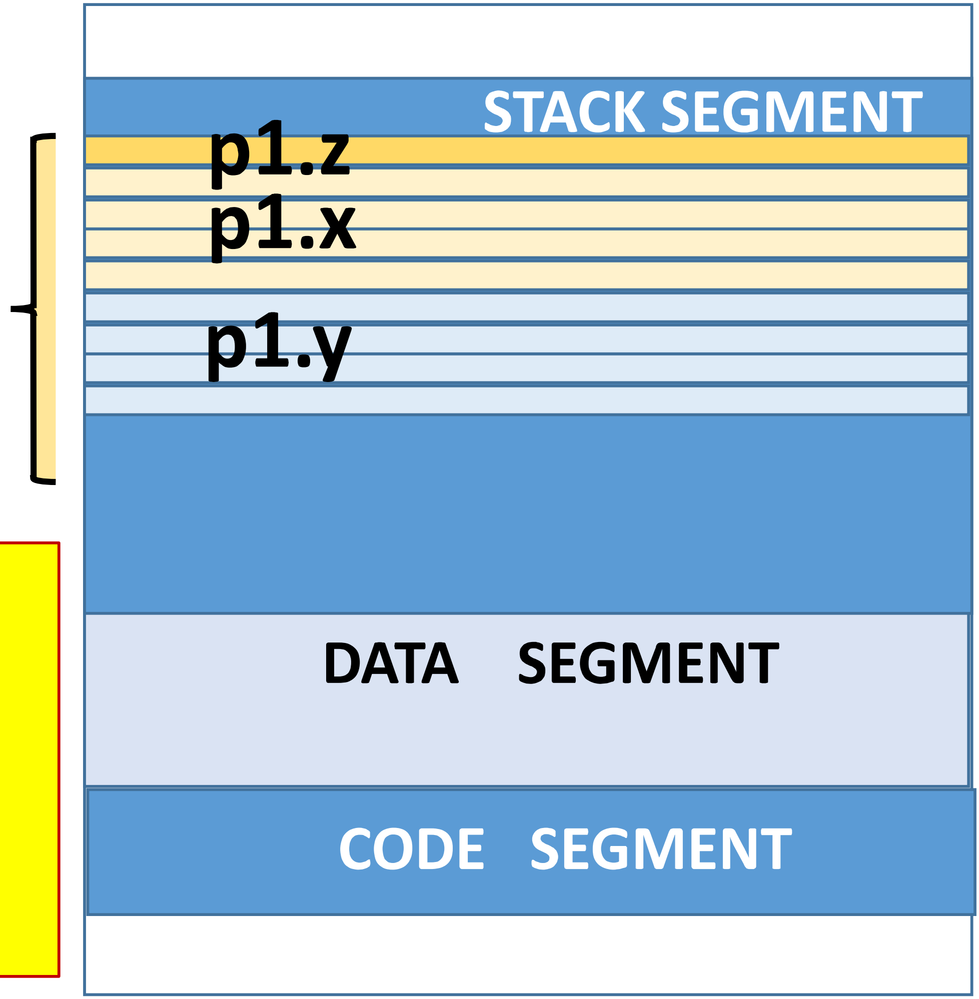
```
int main()
{
   struct MyStructType {
     char z;
     int x, y;
   };
  MyStructType p1;
  int a;
}
```



p1, a: local variables of "main"
Memory for "p1" and "a" allocated in activation record of "main" in call stack

What Can We Safely Assume About Structures?

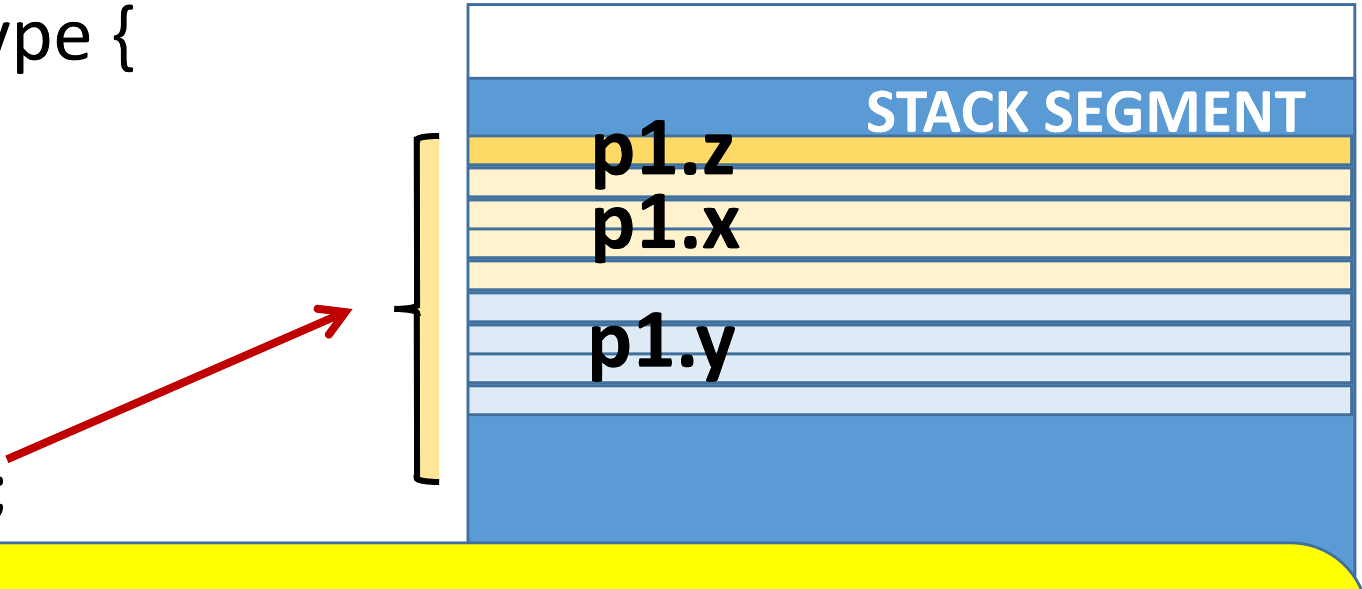
```
struct MyStructType {
   char z;
   int x, y;
};
MyStructType p1;
```



No assumptions about relative layout of different members within memory allocated for a structure

What Can We Safely Assume About Structures?

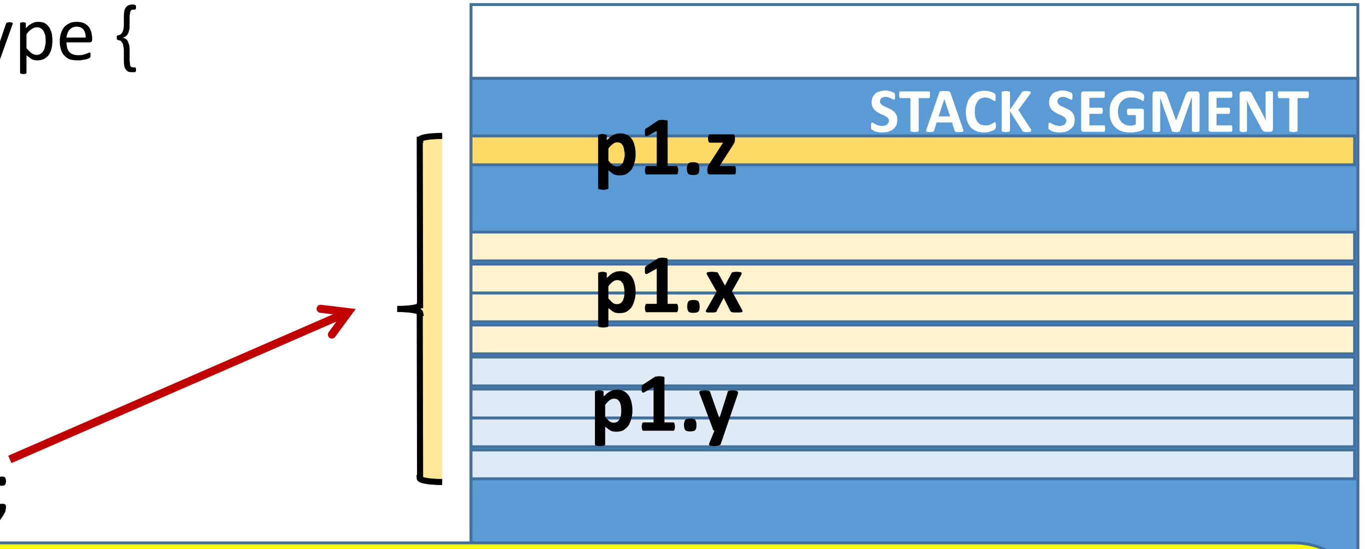
```
struct MyStructType {
   char z;
   int x, y;
};
MyStructType p1;
```



No assumptions about “padding” (unused memory locations) after locations allocated for different members of a structure

What Can We Safely Assume About Structures?

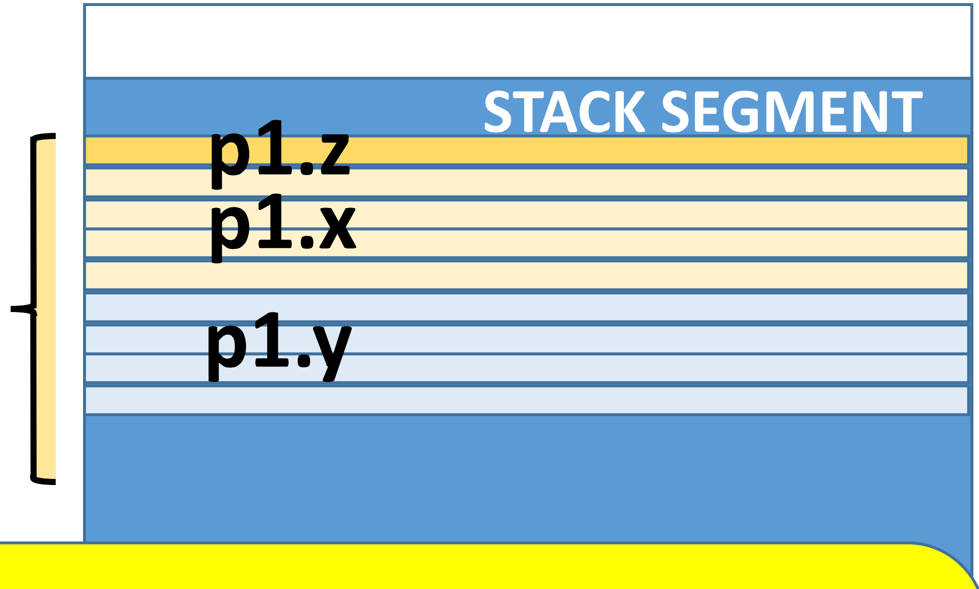
```
struct MyStructType {
   char z;
   int x, y;
};
MyStructType p1;
```



No assumptions about "padding" (unused memory locations) after locations allocated for different members of a structure

What Can We Safely Assume About Structures?

```
struct MyStructType {
   char z;
   int x, y;
};
MyStructType p1;
```



Memory locations allocated for each member are however contiguous (have consecutive addresses). E.g., four contiguous locations for p1.x, four contiguous locations for p1.y

Recall: “&” and “*” Operators

- We used “&” and “*” operators with variables of basic data types

```
 int a;
 int * ptrA;
 ptrA = &a;
*ptrA = 10;
```

Pointer-to-integer data type

Recall: “&” and “*” Operators

- We used “&” and “*” operators with variables of basic data types

```
int a;
int * ptrA;
ptrA = &a;
*ptrA = 10;
```

Address of (starting location) of variable “a” of type “int”

Recall: “&” and “*” Operators

IIT Bo

- We used “&” and “*” operators with variables of basic data types

```
int a;
int * ptrA;
ptrA = &a;
*ptrA = 10;
```

Contents (as “int”) of memory locations whose starting address is given by “ptrA

“&” and “*” Operators for Structures

We can use “&” and “*” operators with variables of structure types in exactly the same way

```
int a;
int * ptrA;
ptrA = &a;
*ptrA = 10;
```

```
struct MyStructType {
   char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
```

“&” and “*” Operators for Structures

We can use “&” and “*” operators with variables of str [illegible] ame way

Pointer-to-MyStructType data type

```
ptrA = &a;
*ptrA = 10;
```

```
struct MyStructType {
   char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
```

“&” and “*” Operators for Structures

We can use “&” and “*” operators with variables of stru ame way

Address of (starting location) of variable p1 of type MyStructType

```
ptrA = &a;
*ptrA = 10;
```

```
struct MyStructType {
  char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
```

“&” and “*” Operators for Structures

We can use “&” and “*” operators with variables of
ame way

Contents (as “MyStructType”) of memory locations whose starting address is given by “ptrP1”

```
*ptrA = 10;
```

```
struct MyStructType {
  char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
```

Accessing Members Through Pointers

- Can we access p1.x through ptrP1?
- Yes, and by the obvious way:

  E.g. **(*ptrP1).x = 1 + (*ptrP1).y;**

```
struct MyStructType {
  char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
```

Accessing Members Through Pointers

- Can we access p1.x through ptrP1?
- Yes, and by the obvious way:

E.g. **(*ptrP1).x = 1 + (*ptrP1).y;**

***ptrP1 is an object of type MyStructType**

```
struct MyStructType {
   char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
```

Accessing Members Through Pointers

- Can we access p1.x through ptrP1?
- Yes, and by the obvious way:

E.g. **(*ptrP1).x = 1 + (*ptrP1).y;**

(*ptrP1).x is the member “x” of the object (*ptrP1) of type MyStructType

```
struct MyStructType {
  char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
```

Accessing Members Through Pointers

```
struct MyStructType {
  char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
```

- Can we access p1.x through ptrP1?
- Yes, and by the obvious way:

  E.g. **(*ptrP1).x = 1 + (*ptrP1).y;**

C++ provides the “->” operator for above situations

  E.g. **ptrP1->x = 1 + ptrP1->y;**

ptrVar->memberName is equivalent to (*ptrVar).memberName

Accessing Members Through Pointers

```
struct MyStructType {
   char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
(*ptrP1).x = 1 + (*ptrP1).y;
```

```
struct MyStructType {
   char z;  int x, y;
};
MyStructType p1;
MyStructType * ptrP1;
ptrP1 = &p1;
ptrP1->z = ‘c’; ptrP1->x = 2;
ptrP1->y =  3;
ptrP1->x = 1 + ptrP1->y;
```

Accessing Members Through Pointers

```
struct MyStructType {
  ch
};
MyS
MyStructType * ptrP1;
ptrP1 = &p1;
*ptrP1 = {‘c’, 2, 3};
(*ptrP1).x = 1 + (*ptrP1).y;
```

Functionally equivalent program fragments

```
struct MyStructType {

MyStructType * ptrP1;
ptrP1 = &p1;
ptrP1->z = ‘c’; ptrP1->x = 2;
ptrP1->y =  3;
ptrP1->x = 1 + ptrP1->y;
```

Summary

- Pointers to variables of structure data types
- Use of “&” and “*” operators with structures
- Use of “->” operator to access members of structures through pointers.