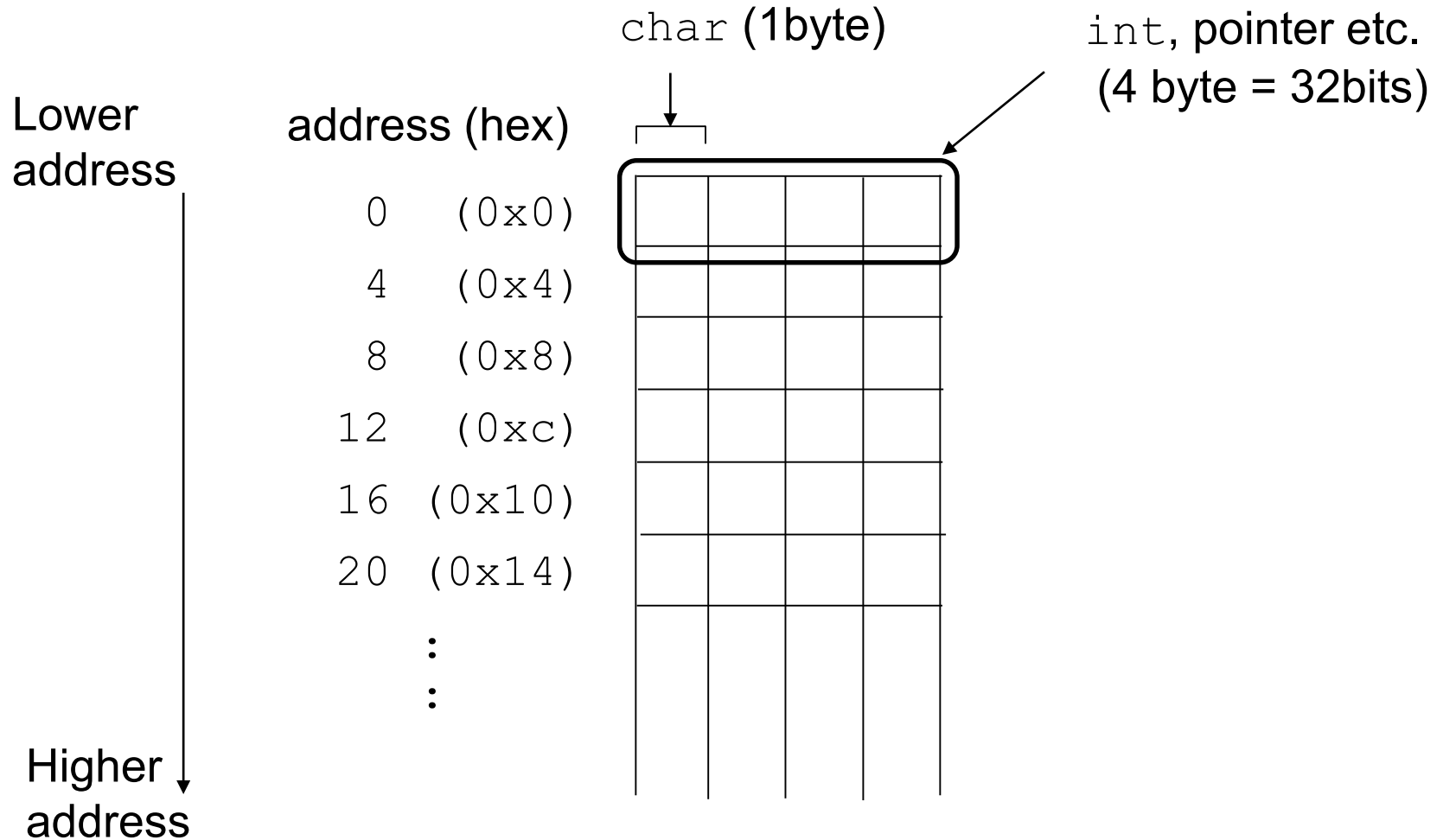
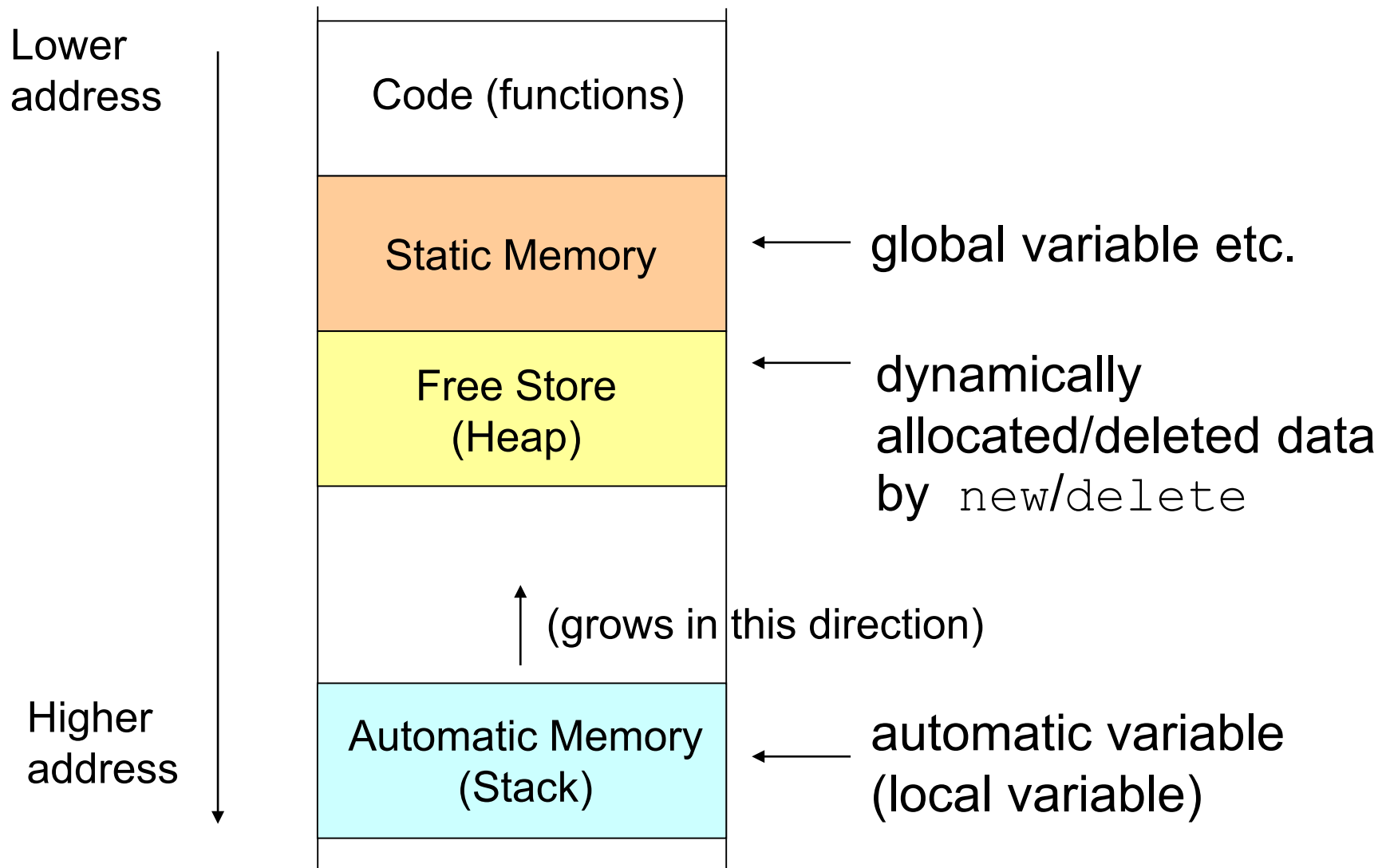


Logical Structure of Computer Memory



(Assuming a typical 32-bit computer)

Three types of data storage



(The location of each memory area may be different depending on compiler implementation.)

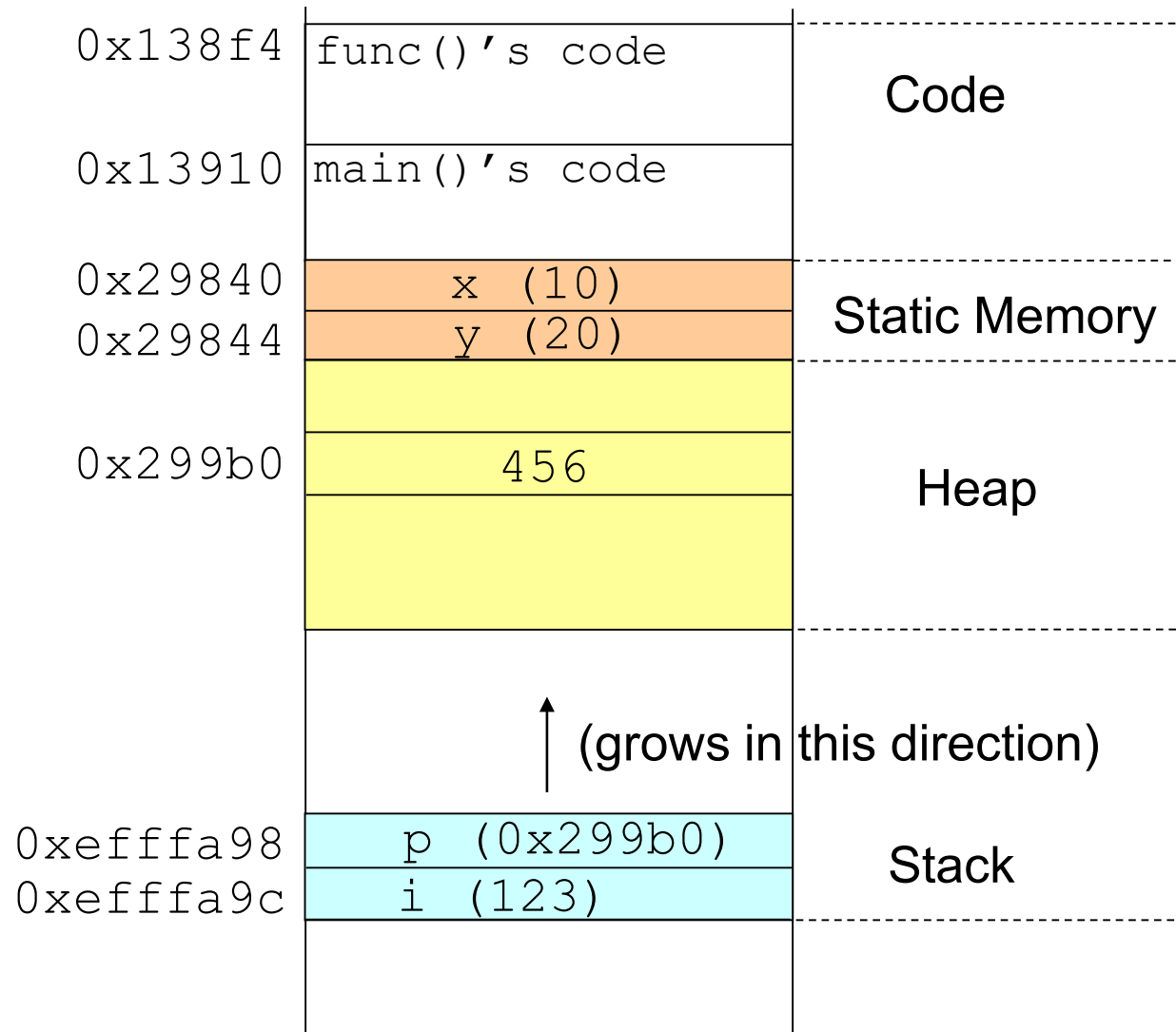
Memory map in a C++ program

```
int x = 10
int y = 20;

void func() {
    ...
}

int main()
{
    int i = 123;
    int *p;

    p = new int;
    *p = 456;
    ...
}
```



(The addresses in the picture are examples. Actual addresses depend on programs and compilers.)

Stack and stack frame

Stack is used for storing local variables

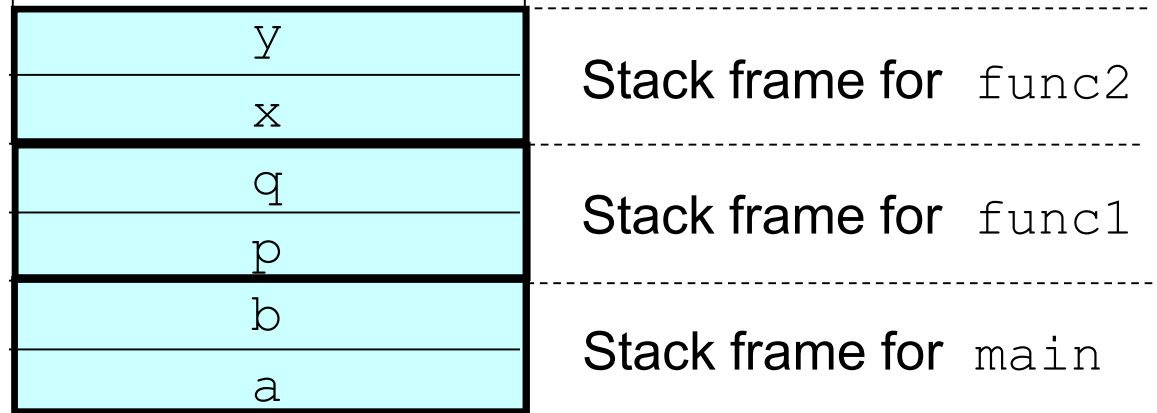
(and return address (but not illustrated in this slide))

```
void func2() {  
    int x, y;  
}
```

```
void func1() {  
    int p, q;  
    func2();  
}
```

```
int main()  
{  
    int a, b;  
    func1();  
}
```

- When entering a function, a new stack frame is created.
- When execution of the function is done, its stack frame will be removed from the stack top. Removed stack frames cannot be accessed.



(Stacking order within a stack frame depends on compiler implementation.)

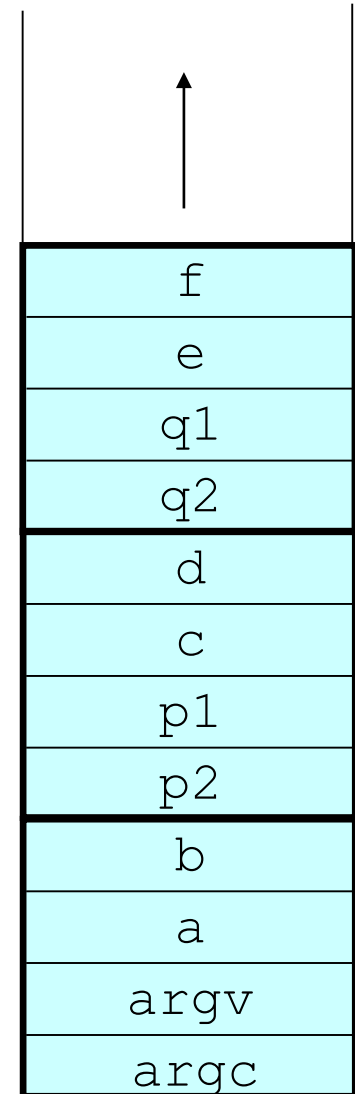
Stack and stack frame

Function arguments are also local variables

```
void func2(int q1, int q2) {
    int e, f;
}

void func1(int p1, int p2) {
    int c, d;
    func2();
}

int main(int argc, char* argv[])
{
    int a, b;
    func1();
}
```

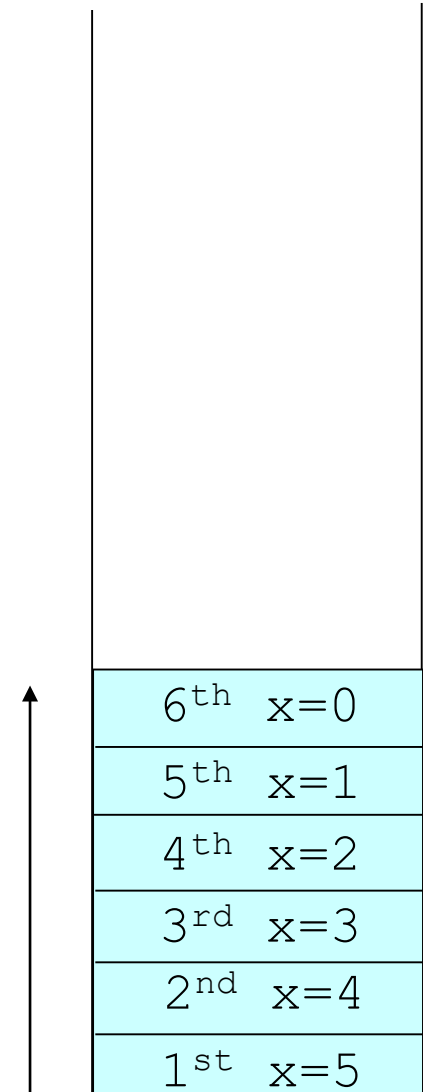


Stack and Recursion

Stack allows *recursion*

```
#include <iostream>
using namespace std;
int sum(int x)
{
    if (x == 0) {
        return 0;
    } else {
        return x + sum(x - 1);
    }
}

int main()
{
    cout << sum(5) << endl;
    return 0;
}
```

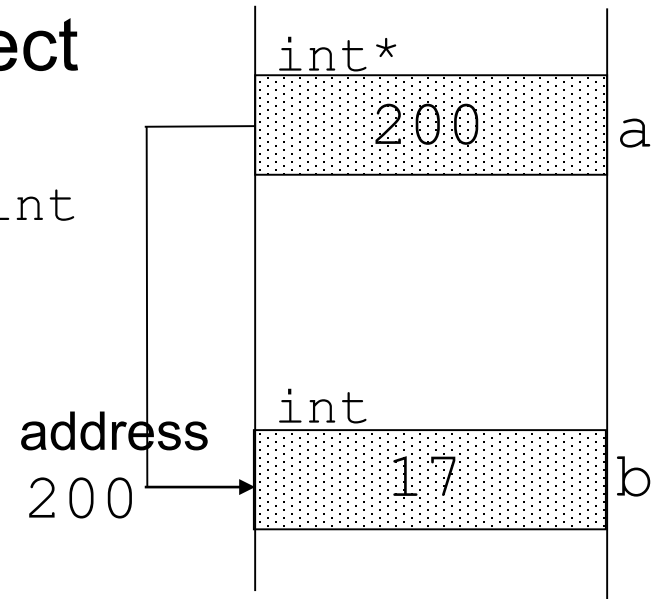


Pointers

Contain an address of a data object

```
int* a;          // 'a' is a pointer to int
int b = 17;      // b, c contains int
int c;
```

```
a = &b;  // a=200 (address of b)
c = *a;  // c=17
```



- operator ' & ' (applied to any type of variables)
gives the variable's address
ex. &b gives b's address
- operator ' * ' (applied to pointers)
gives the contents which is pointed by the pointer
ex. *a gives the contents which is pointed by a
(Applying ' * ' operator to a pointer is called '*dereference*')

Pointers

Don't get confused by the meaning of '*'...

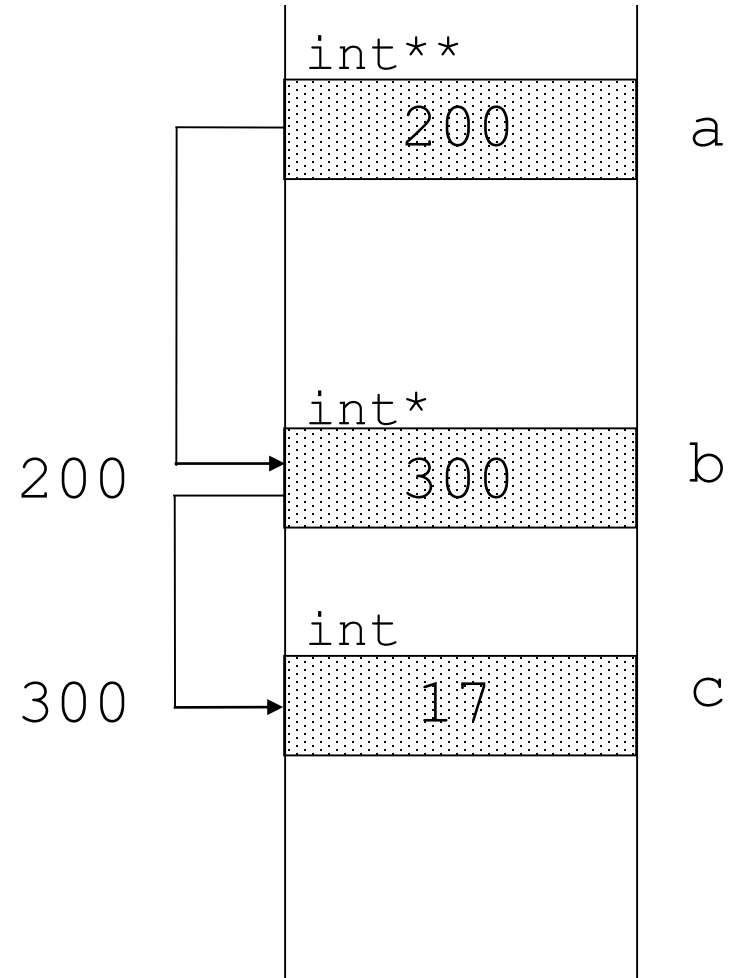
```
int* a; // just declaring that a is 'a pointer to int'  
int b;  
int c;
```

```
a = &b; // get b's address and store it into a  
c = *a; // get the contents which a points to and  
        // store it into c
```

Pointers to a pointer

Contain an address of a pointer

```
int** a;  
int* b;  
int* e;  
int c=17, d;  
  
a = &b;  
b = &c;  
  
d = *b; //d=17  
e = *a; //e=300
```



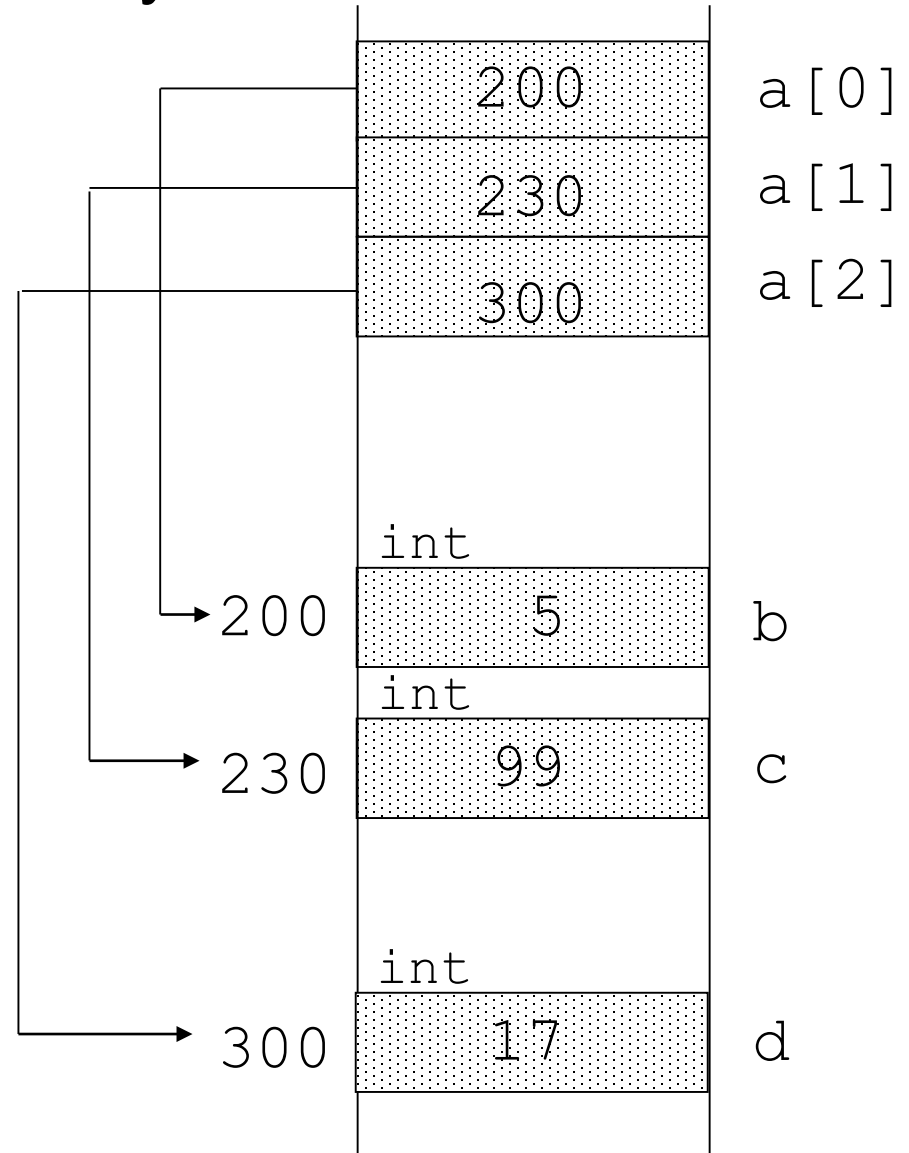
Any number of levels of pointing is possible.
(pointer to pointer to pointer etc.)

Arrays of pointers

Have pointers as their array elements

```
int* a[3];  
int b=5, c=99, d=17;
```

```
a[0]=&b;  
a[1]=&c;  
a[2]=&d;
```

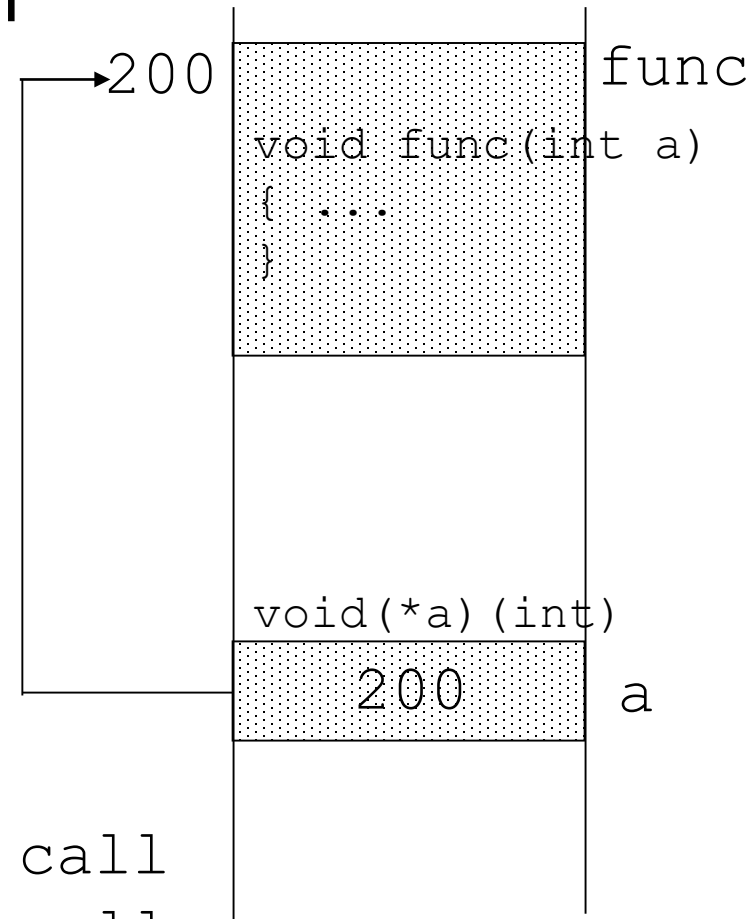


Pointers to a function

Have an address of a function

```
void func(float)
{ ...
}
```

```
void (*a)(float);
a = func;
(*a)(1.2); //causes function call
a(1.2);    //causes function call
```



Operations on pointers

```
int  a[10], b;  
int* c;
```

```
c = &a[0];    // c points to a[0]
```

```
c += 2;      // c points to a[2]  
              // (value of c increases  
              //   by 8 (=2*sizeof(int))
```

```
c++;         // c points to a[3]  
b = *c++;    // a[3] is copied to b  
              // and c points to a[4]
```

```
*c = 3;      // a[4] becomes 3  
(*c)++;     // a[4] is incremented
```

Dynamic storage allocation

new / delete

- Dynamically allocates/deletes a memory chunk on the heap
- Used for creating dynamic data structure such as a linked list
- For each use of `new`, corresponding `delete` must exist
- `delete[]` must be used for deletion of arrays
- Do not mix with `malloc()` / `free()`

```
int *a = new int;           // allocates 1 int object
```

```
int *b = new int[100]; // allocates an array of int
```

```
delete a;           // deallocation of 1 int
```

```
delete[] b;           // deallocation of an int array
```

```
delete b;           // wrong deallocation !!
```

```
// (only b[0] will be deallocated)
```

References

References are 'aliases' (Initialization is required)

```
int    a, b;
int& x;    // x is declared as a reference
int& y;    // y is declared as a reference

x = a;    // hereafter, x is an alias of a.
           // a can be referred with a name 'x'

x = 1;    // means 'a = 1'
x = b;    // now, b is just copied to a
           // (because x is an alias of a)

y = 1;    // error !! (no initialization)
```

References

Don't get confused by the meaning of '&'...

```
int& a; // declare that a is 'a reference to int'
int* b; // declare that b is 'a pointer to int'
int c;
int d;
```

```
a = c; // initialize reference a with c
b = &a; // get a's address (meaning, c's address)
        // and store it into b
d = *b; // get the contents which b points to and
        // store it into d (meaning, d will have the same
        // value as c)
```

Ex. usage of references

[pointer]

```
void increment(int *p)
{
    *p = *p + 1;
}
```

```
main()
{
    int a = 1;
    increment(&a);
}
```

[reference]

```
void increment(int &p)
{
    p = p + 1;
}
```

```
main()
{
    int a = 1;
    increment(a);
}
```

const keyword

const keyword specifies “read-only”

```
int x;  
int* const y = &x;           // a const pointer y  
y++;                          // error !! (y itself is const)  
*y = 10;                     // OK (what is pointed by y  
                             // is not const)
```

```
int x = 123;                  // a constant int 123  
const int *x = y;             // a pointer to a constant int  
int const *x = y;             // same meaning as above  
*x = 10;                      // error !! (what is pointed by  
                             // x is const)  
x++;                          // OK (x itself is not const)
```

```
int x = 123;  
const int& y = x;             // const reference  
y = 10;                       // error !!
```

Arrays and const

```
int a[10];           // OK. 10 is constant.
```

```
int size = 10;
int a[size];         // error !! size is not a constant.
                     // (Actually, g++ has a language
                     // extension and accepts this.
                     // But generally speaking,
                     // you should not do this.
                     // It's not a portable way.
```

```
const int size = 10;
int a[size];         // OK
```

```
int& a[10];          // error !! ref. requires init.
                     // (arrays of references cannot
                     // be initialized.)
```