POINTERS

MANUAL VS SMART POINTERS

Types

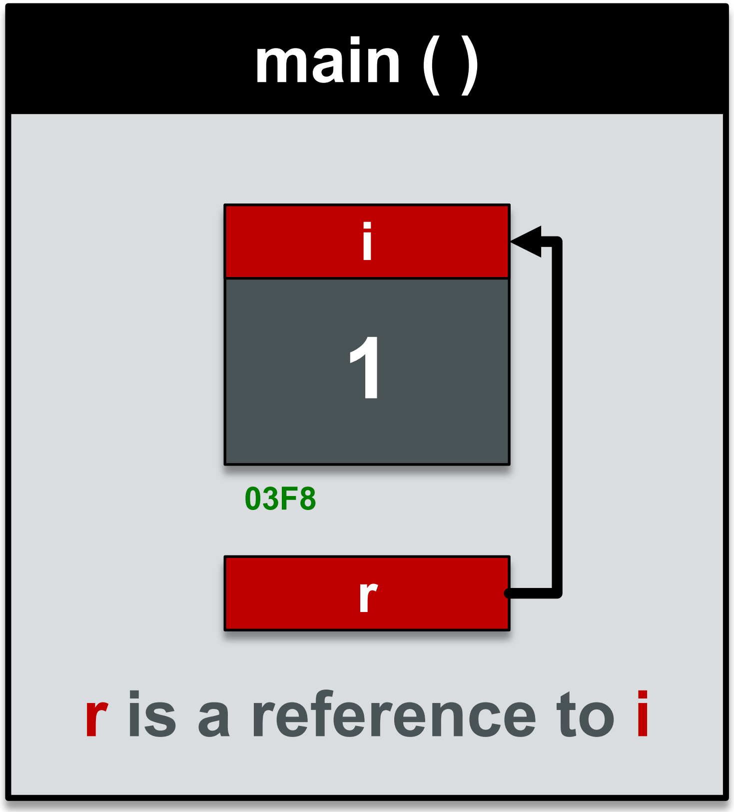
manual pointers

1. manually defined requiring explicit memory management
2. easy to make mistakes, difficult to troubleshoot
3. legacy approach for implementing dynamic memory
4. requisite background knowledge for a C++ developer
5. requires a thorough understanding of memory management

smart pointers

1. automatic memory management
2. easy to implement
3. modern method of implementing dynamic memory in C++
4. requires a wide variety of C++ knowledge to properly appreciate their use including template programming, STL, move semantics, R-value references etc.
5. not covered in ET580, recommended for future study

REFERENCE REVIEW

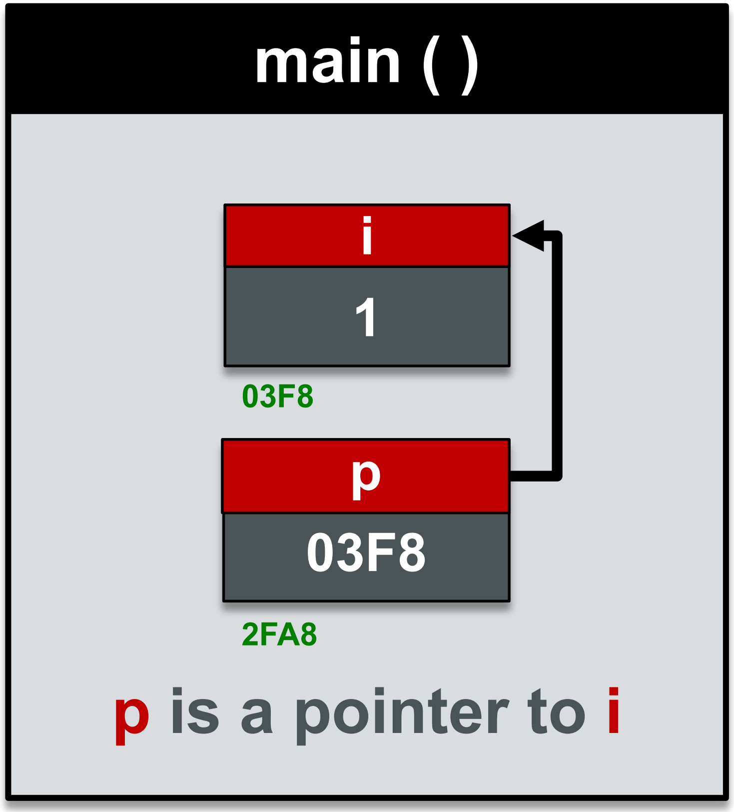


cout << &i; prints address 0x03F8
cout << &r; prints address 0x03F8
cout << i; prints 1
cout << r; prints 1

Variables i and r have the same memory address because they are different aliases or names for the same memory location

Therefore, they have the same value 1

POINTER REVIEW



cout << &i;	prints address 0x03F8
cout << &p;	prints address 0x2FA8
cout << i;	prints 1
cout << p;	prints 0x03F8

The pointer p stores the memory address of i. Variables i and p have the different memory address because they are different variables.

REFERENCES VS. POINTERS

Reference	an additional name (alias) for an existing variable	
Pointer	a variable that stores the memory address of another variable	
Example	int i=5;	initialize integer variable i with the value 5
	int &r = i;	initialize a second name r for the variable i
	int *p = &i;	initialize integer pointer variable p with its value set to the memory address of i

i and r are different names for the same variable
i and p are different variables

DEREFERENCE OPERATOR

Concept the dereference operator * returns the variable that a pointer points to

Example

`int i=5;`	initialize integer i and with the value 5
`int *p = &i;`	initialize integer pointer p and set its value to the memory address of i
`cout << i;`	print the value of i which is 5
`cout << *p;`	dereference p (return i) and print its value 5
`*p = 10;`	dereference p (return i) and assign it a new value 10
`cout << i;`	print the updated value of i which is 10

NULLPTR

Purpose a safe value for a pointer variable

Example

int *p=nullptr;	initialize integer pointer p with the value nullptr
int *p;	declare integer pointer p with a garbage value

```
if(p == nullptr) {
      run some code
}
```

can test if p points to nothing

legacy versions of C++ use null instead of nullptr

POINTER SYNTAX

Example	`int i=5;`	initialize integer i with the value 5
	`int *p = nullptr;`	initialize the integer pointer p
	`p = &i;`	assign p to the memory address of i
	`*p = 10;`	dereference p to access and modify the value of i
	`double *a, b;`	declare a double pointer a and a double b
	`double c, *d;`	declare a double c and a double pointer d
	`double *e, *f;`	declare two double pointers e and f

POINTER EQUIVALENCE

Example	double d=3.14;	initialize integer d with the value 3.14
	double *p = &d;	initialize the double pointer p
	double *q = &d;	initialize the double pointer q
	if(&p == &q) {}	test if p and q are the same variable
	if(p == q) {}	test if p and q point to the same variable
	if(*p == *q) {}	test if p and q point to variables with the same value

POINTERS AND CONSTANTS

const pointer	the pointer cannot be modified
pointer to a const variable	the variable pointed to cannot be modified
const pointer to a const variable	both variables cannot be modified

Examples

`int a = 5;`	non-constant variable
`const int b = 5;`	constant variable
`int *const p = &a;`	the pointer cannot be modified
`const int *p = &b;`	the variable pointed to cannot be modified
`const int *const p = &a;`	both variables cannot be modified

Note

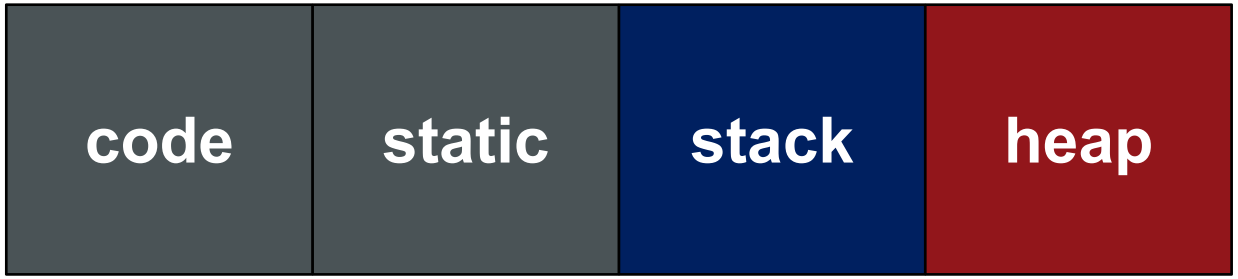
`const int *p;` or `const int *const p;` can point to constants or non-constants, regardless of what it points to, `*p` cannot be modified

MEMORY

Stack
memory space for automatic variables
memory managed by the compiler

Heap
memory space for dynamic variables
memory managed by the programmer
requires the use of pointers
requires the use of new and delete operators

Static
memory space for global variables

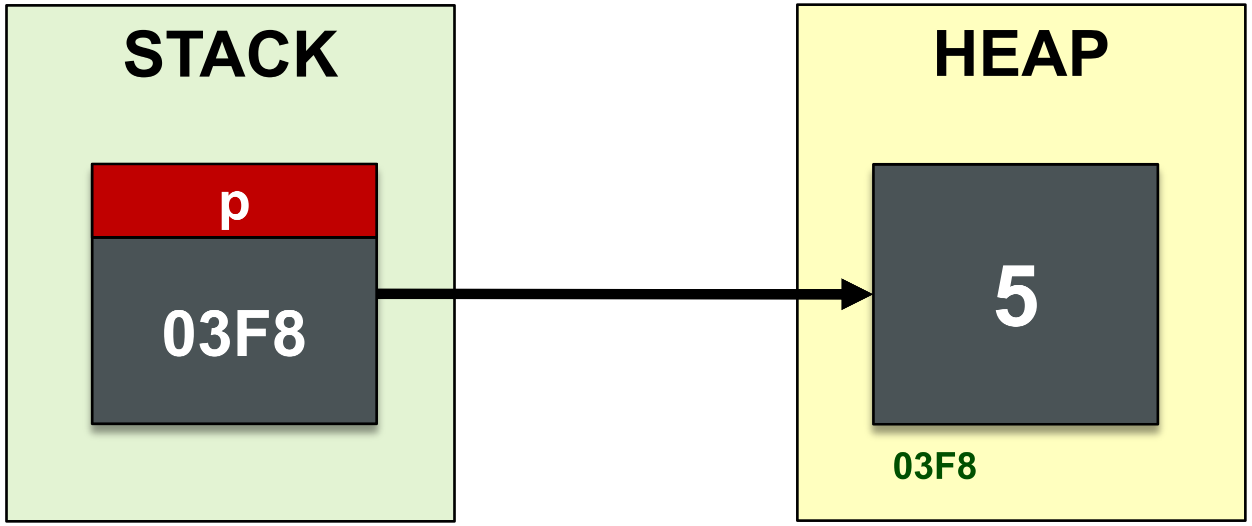


HEAP

Pointers	required to access memory locations on the heap	
New operator	used to allocate memory on the heap	
Delete operator	used to deallocate memory on the heap	
Example	`int *p = new int(5);`	allocate a dynamic variable on the heap which is accessed by a pointer p on the stack
	`cout << *p;`	access the dynamic variable
	`*p = 10;`	modify the dynamic variable
	`delete p;`	deallocate the variable pointed to by p does not deallocate the pointer p

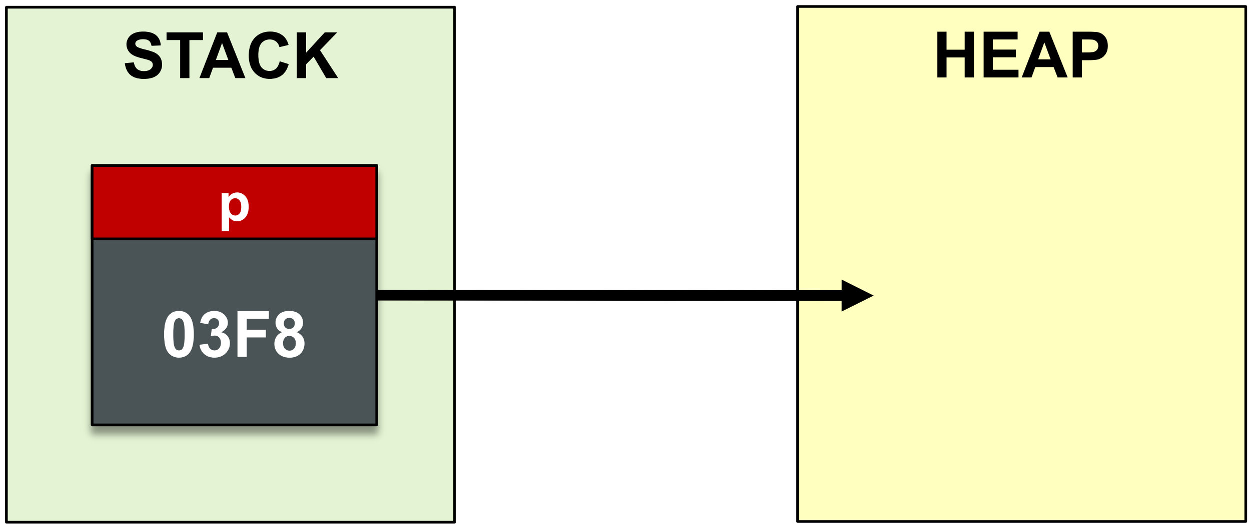
NEW OPERATOR

Example	int *p = new int(5);	allocate a dynamic variable on the heap
Note	p is an automatic variable on the stack we access the dynamic variable using *p *p represents the dynamic variable allocated on the heap	



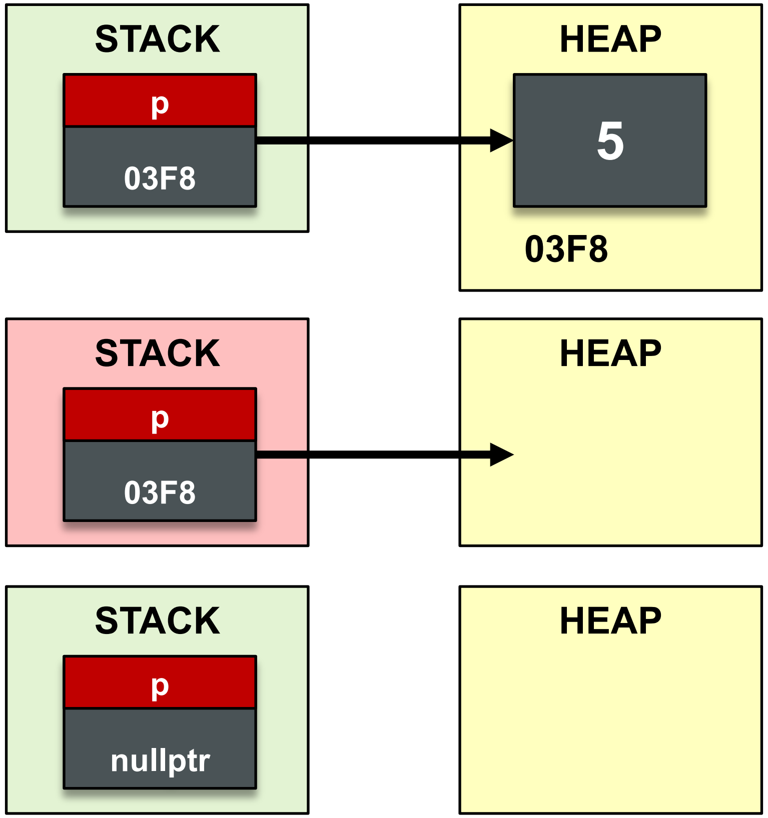
DELETE OPERATOR

Example	int *p = new int(5);	allocate a dynamic variable on the heap
	delete p;	deallocate the dynamic variable on the heap
Note	p remains on the stack while *p is recycled	



DANGLING POINTER

Concept a pointer which points to an address that no longer exists



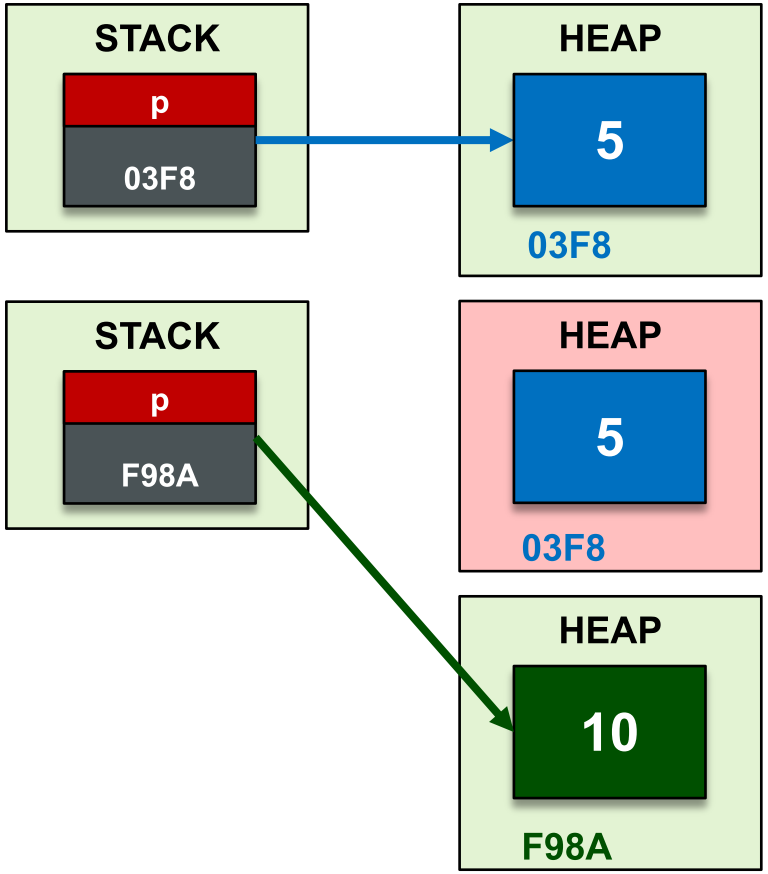

```
int *p = new int(5);

delete p;       // dangling pointer

p = nullptr;    // safe pointer
```

MEMORY LEAK

Concept dynamic memory that is not accessible



int *p = new int(5);

p = new int(10);   // memory leak

03F8 is no longer accessible
03F8 will not be recycled within program lifetime
if enough leaks occur, program may crash

AUTOMATIC VARIABLES AND FUNCTIONS

Return by value

always return local automatic variables by value (return a copy)

```
int f( ) {
    int i = 100;
    return i;     // i goes out of scope, is recycled
}
```

Return by reference

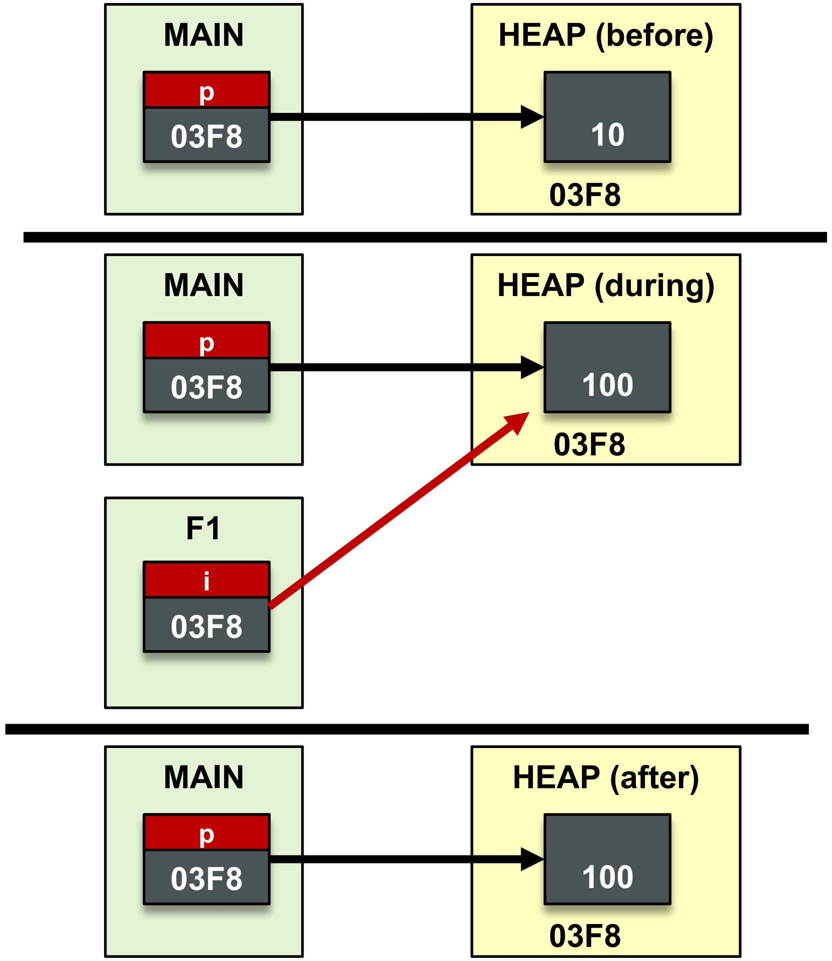
<u>never</u> return a local automatic variable by reference (garbage)

```
int& f( ) {
    int i = 100;
    return i;     // i goes out of scope, removed from runtime stack
}
```

FUNCTIONS AND POINTERS

Pass by value	pass the pointer value (memory address of pointed to variable)
	`void f(int *p) { }`
Return by value	return the pointer value (memory address of pointed to variable)
	`int* f( ) { }`
Pass by reference	pass the location (memory address) of the pointer variable
	`void f(int *&p) { }`
Return by reference	return the location (memory address) of the pointer variable
	`int *& void f( ) { }`

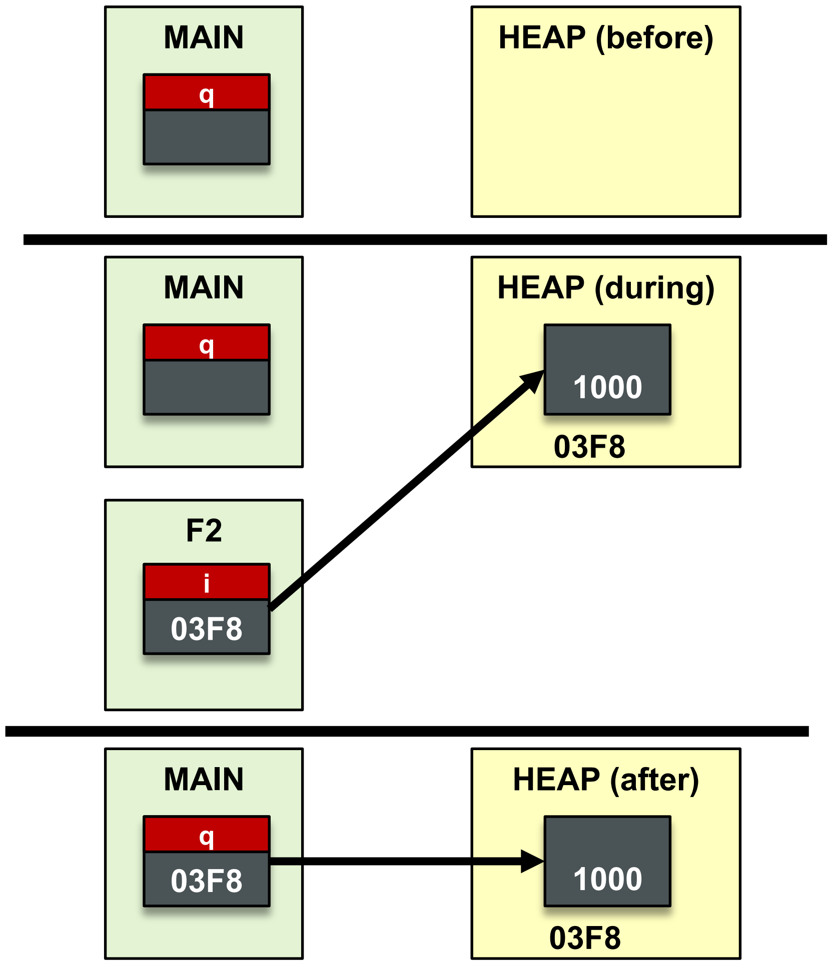
PASS A POINTER BY VALUE




```
void f1(int *i) {
        *i = 100;
}

int main( ) {
        int *p = new int{10};
        f1(p);
        cout << *p << "\n";          // print 100
}
```

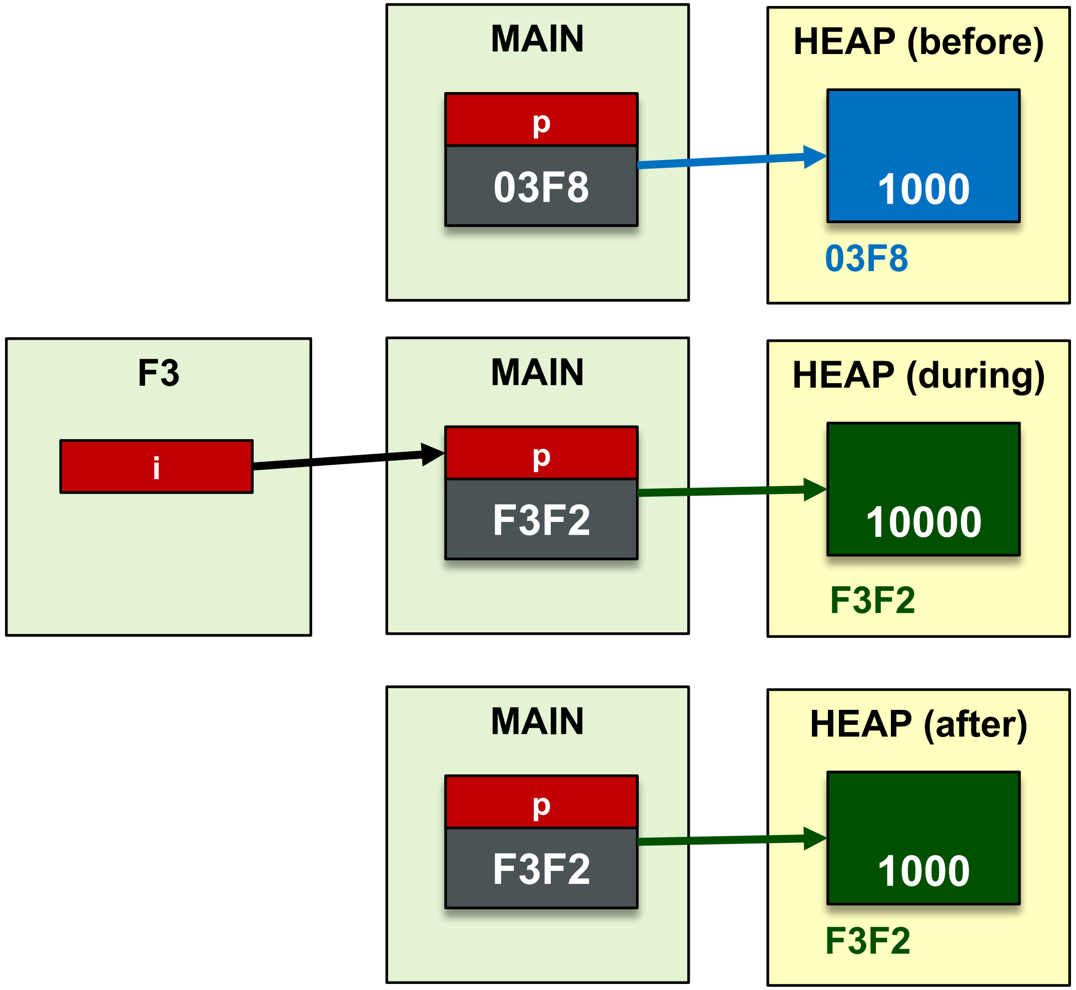
RETURN A POINTER BY VALUE




```
int* f2() {
    int *i = new int(1000);        // i goes out of scope
    return i;                      // return heap memory
}

int main( ) {
    int *q = f2();
    cout << *q << “\n”;            // print 1000
}
```

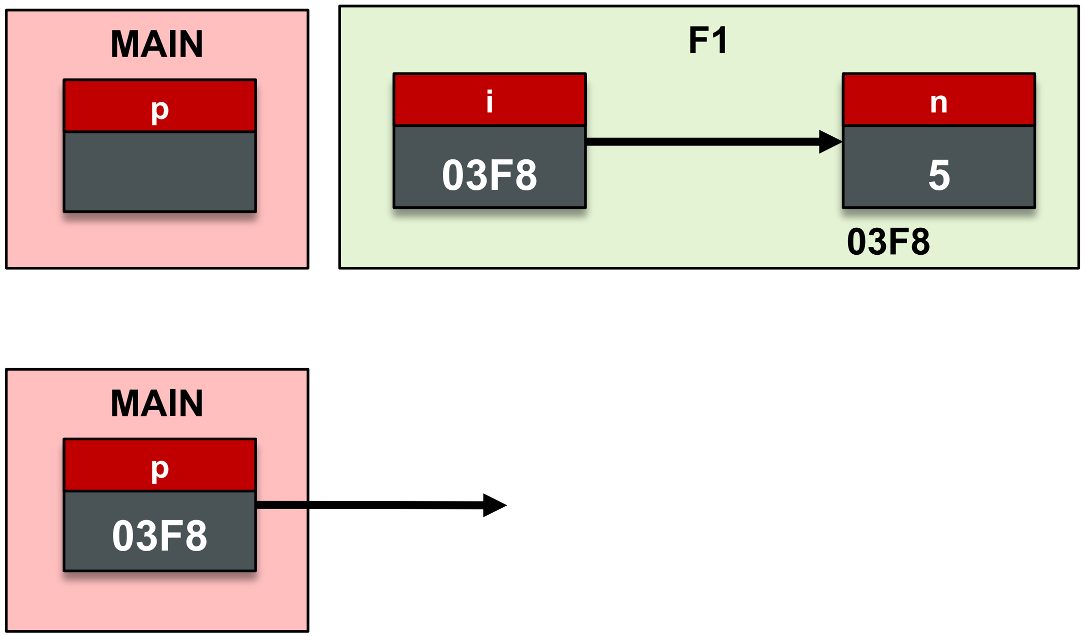
PASS A POINTER BY REFERENCE




```
void f3(int *&i) {
        delete i;                    // delete 03F8
        i = new int(10000);          // initialize F3F2
}

int main( ) {
        int *p = new int{1000};      // p points to 03F8
        f3(p);                       // p points to F3F2
        cout << *p << "\n";          // prints 10000
}
```

FUNCTIONS AND DANGLING POINTERS

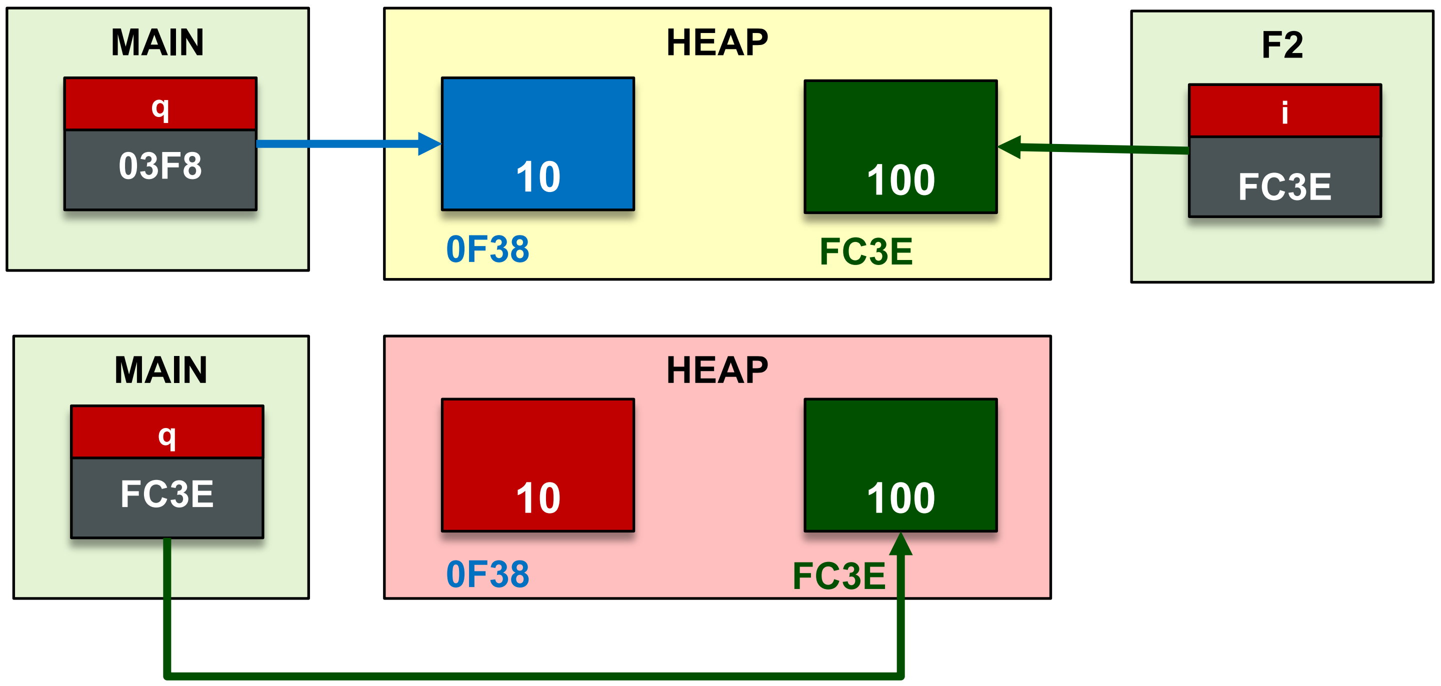



```
int* f1( ) {
      int *i;
      int n=5;
      i=&n;
      return i;            // n and i go out of scope
}

int main( ) {
      int *p = f1( );      // 03F8 is no longer valid
}
```

p points to a memory location which is no longer valid

FUNCTIONS AND MEMORY LEAKS




```
int* f2( ) {
        int *i; = new int(100);
        return i;
}

int main( ) {
        int *q = new int{10};
        q = f2( );      // q assigned to new
}
```

03F8 is no longer reachable since after the function call q points to FC3E

FUNCTION POINTERS

Purpose a variable which stores the address of a function

```
int add(int a, int b) { return a+b; }
int multiply(int a, int b) { return a*b; }

void print (int a, int b, int (*f) (int, int) ) {          // function pointer parameter
      cout << (*f)(a, b) << "\n";                           // call the function
}

int main() {
      int (*f) (int, int);          // declare a function pointer
      f = add;                      // assign the function pointer to the add function
      print(5, 6, add);             // call print with literals 5, 6 and function add
      print(5, 6, multiply);        // call print with literals 5, 6 and function multiply
      return 0;
}
```

TYPEDEF

Purpose custom aliases for types to make code easier to read

Example:

```
typedef int score;                   // alias for the int type
typedef int* data;                   // alias for the int pointer type
typedef int (*func) (int, int);      // alias for a function pointer
using func2 = int (*f) (int, int);   // c++11 alias declaration syntax

score n = 5;                         // declare an integer variable
data p = new int(100);               // declare an integer pointer variable
func f;                              // declare an function pointer from typedef
func2 f;                             // declare an function pointer from using
```