Object Interactive

Weng Kai

local variable

```
int TicketMachine::refundBalance() {
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

A local variable of the same name as a field will prevent the field being accessed from within a method.

Fields, parameters, local variables

- All three kinds of variable are able to store a value that is appropriate to their defined type.
- Fields are defined outside constructors and methods
- Fields are used to store data that persists throughout the life of an object. As such, they maintain the current state of an object. They have a lifetime that lasts as long as their object lasts.
- Fields have class scope: their accessibility extends throughout the whole class, and so they can be used within any of the constructors or methods of the class in which they are defined.

- As long as they are defined as private, fields cannot be accessed from anywhere outside their defining class.
- Formal parameters and local variables persist only for the period that a constructor or method executes. Their lifetime is only as long as a single call, so their values are lost between calls. As such, they act as temporary rather than permanent storage locations.
- Formal parameters are defined in the header of a constructor or method. They receive their values from outside, being initialized by the actual parameter values that form part of the constructor or method call.

- Formal parameters have a scope that is limited to their defining constructor or method.
- Local variables are defined inside the body of a constructor or method. They can be initialized and used only within the body of their defining constructor or method.
- Local variables must be initialized before they are used in an expression – they are not given a default value.
- Local variables have a scope that is limited to the block in which they are defined. They are not accessible from anywhere outside that block.

Initialization

Member Init

- Directly initialize a member
  - benefit: for all ctors
- Only C++11 works

Initializer list

```
class Point {
private:
   const float x, y;
   Point(float xa = 0.0, float ya = 0.0)
      : y(ya), x(xa) {}
};
```

- Can initialize any type of data
  - pseudo-constructor calls for built-ins
  - No need to perform assignment within body of ctor
- Order of initialization is order of ***declaration***
  - Not the order in the list!
  - Destroyed in the reverse order.

Initialization vs. assignment

```
Student::Student(string s):name(s) {}
```

initialization

before constructor

```
Student::Student(string s) {name=s;}
```

assignment

inside constructor

string must have a default constructor

Function overloading

- Same functions with different arguments list.

```
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5

print("Pancakes", 15);
print("Syrup");
print(1999.0, 10);
print(1999, 12);
print(1999L, 15);
```

Example: leftover.cpp

Overload and auto-cast

```
void f(short i);
void f(double d);

f('a');
f(2);
f(2L);
f(3.2);
```

Example: overload.cpp

Default arguments

- A default argument is a value given in the declaration that the compiler automatically inserts if you don't provide a value in the function call.

```
Stash(int size, int initQuantity = 0);
```

- To define a function with an argument list, defaults must be added from right to left.

```
int harpo(int n, int m = 4, int j = 5);
int chico(int n, int m = 6, int j);//illeagle
int groucho(int k = 1, int m = 2, int n = 3);
```

```
beeps = harpo(2);
beeps = harpo(1,8);
beeps = harpo(8,7,6);
```

Example: left.cpp

C++ access control

- The members of a class can be cataloged, marked as:
  - **public**
  - **private**
  - **protected**

public

- **public** means all member declarations that follow are available to everyone.
  - Example: Public.cpp

private

- The **private** keyword means that no one can access that member except inside function members of that type.

- Example: Private.cpp

Friends

- to explicitly grant access to a function that isn't a member of the structure
- The class itself controls which code has access to its members.
- Can declare a global function as a **friend**, as well as a member function of another class, or even an entire class, as a **friend**.
  - Example: Friend.cpp

class vs. struct

- **class** defaults to **private**
- **struct** defaults to **public**.

- Example: Class.cpp

type of function parameters and return value

way in

- void f(Student i);
  - a new object is to be created in f
- void f(Student *p);
  - better with const if no intend to modify the object
- void f(Student& i);
  - better with const if no intend to modify the object

way out

- Student f();
  - a new object is to be created at returning
- Student* f();
  - what should it points to?
- Student& f();
  - what should it refers to?

hard decision

```
char *foo()
{
    char *p;
    p = new char[10];
    strcpy(p, "something");
    return p;
}
void bar()
{
    char *p = foo();
    printf("%s", p);
    delete p;
}
```

define a pair functions of alloc and free

let user take resp., pass pointers in & out

tips

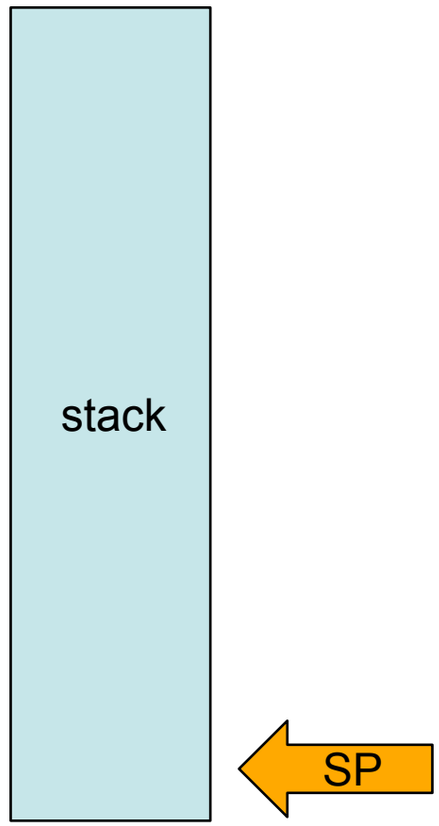
- Pass in an object if you want to store it
- Pass in a const pointer or reference if you want to get the values
- Pass in a pointer or reference if you want to do something to it
- Pass out an object if you create it in the function
- Pass out pointer or reference of the passed in only
- Never new something and return the pointer

Overhead for a function call

- the processing time required by a device prior to the execution of a command
  - Push parameters
  - Push return address
  - Prepare return values
  - Pop all pushed

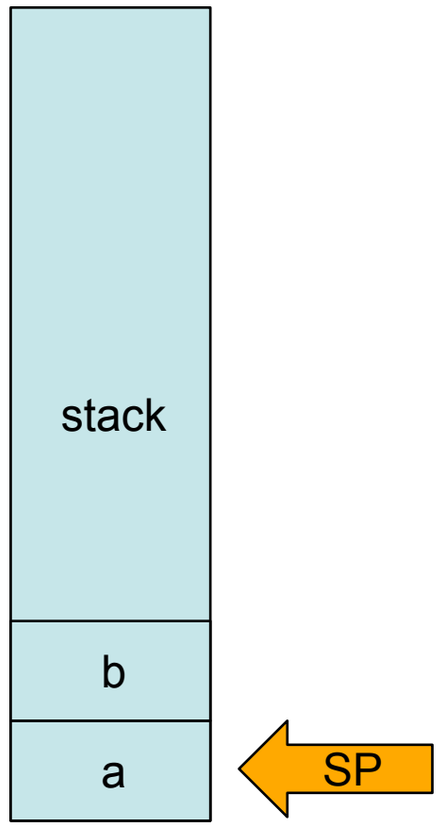
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



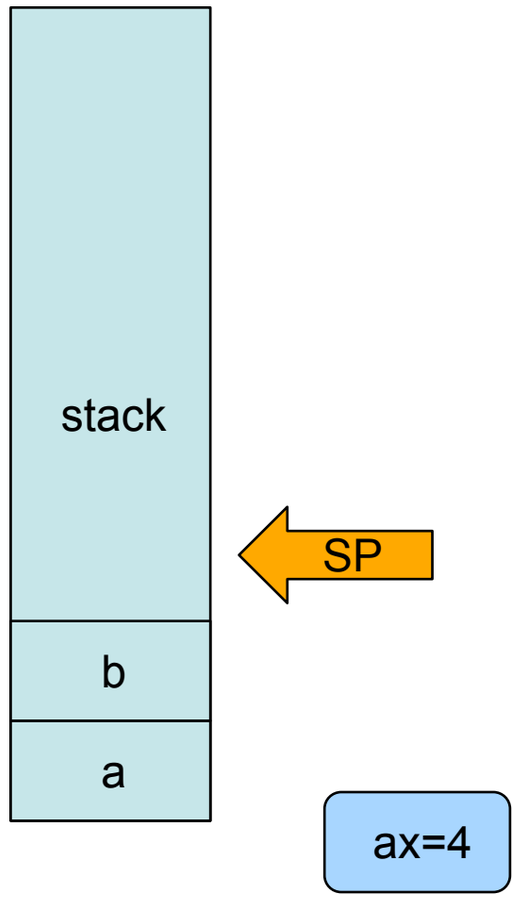
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



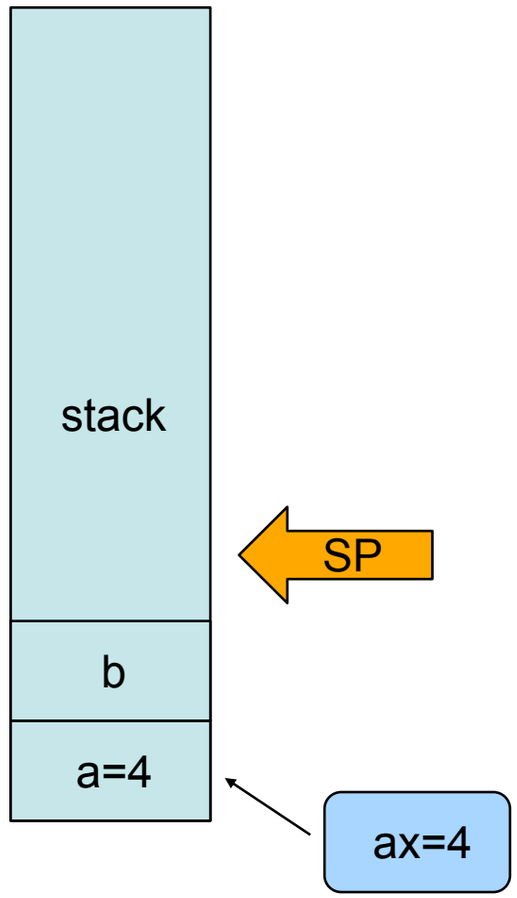
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



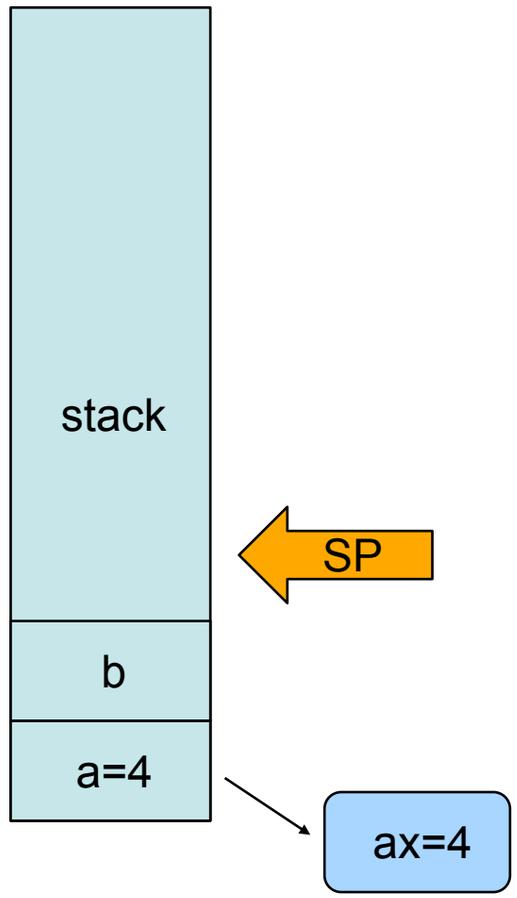
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



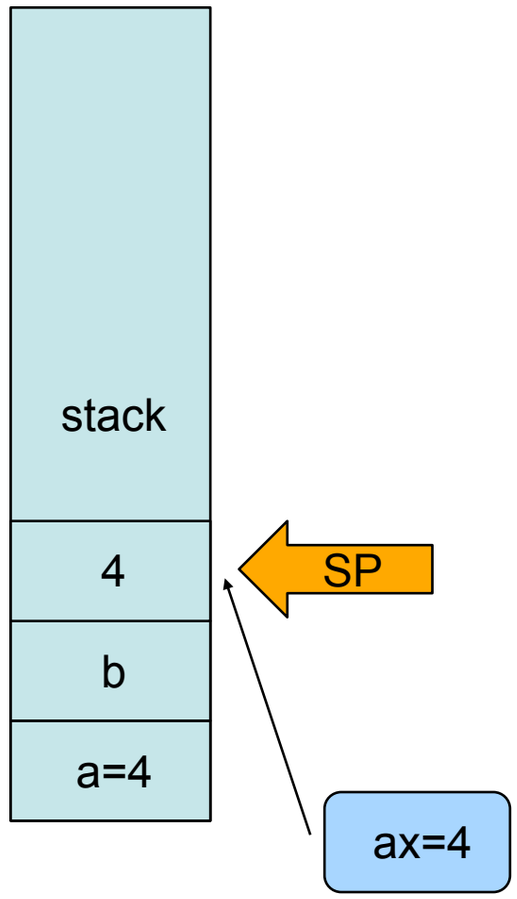
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



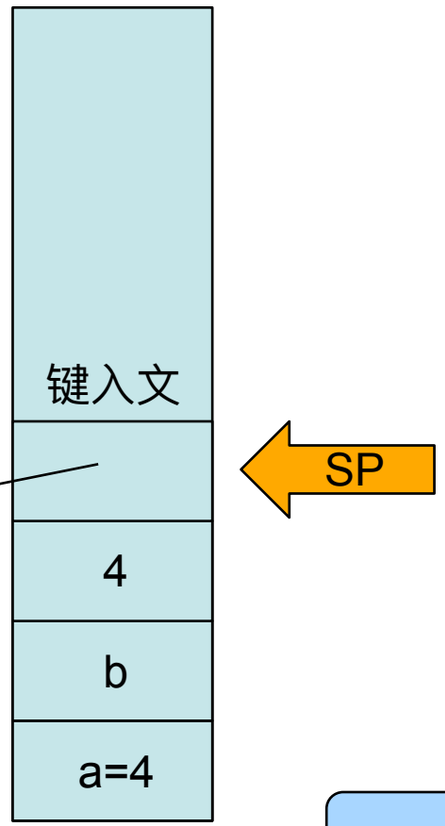
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

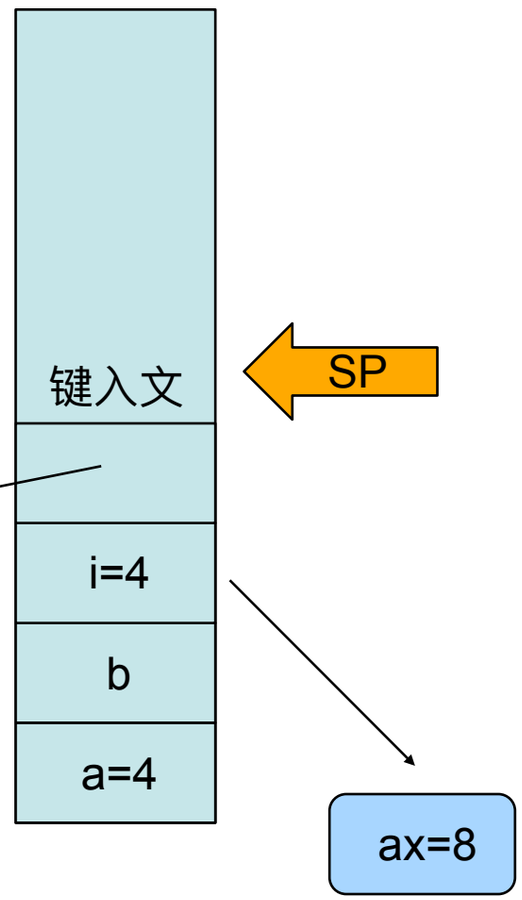
```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



ax=4

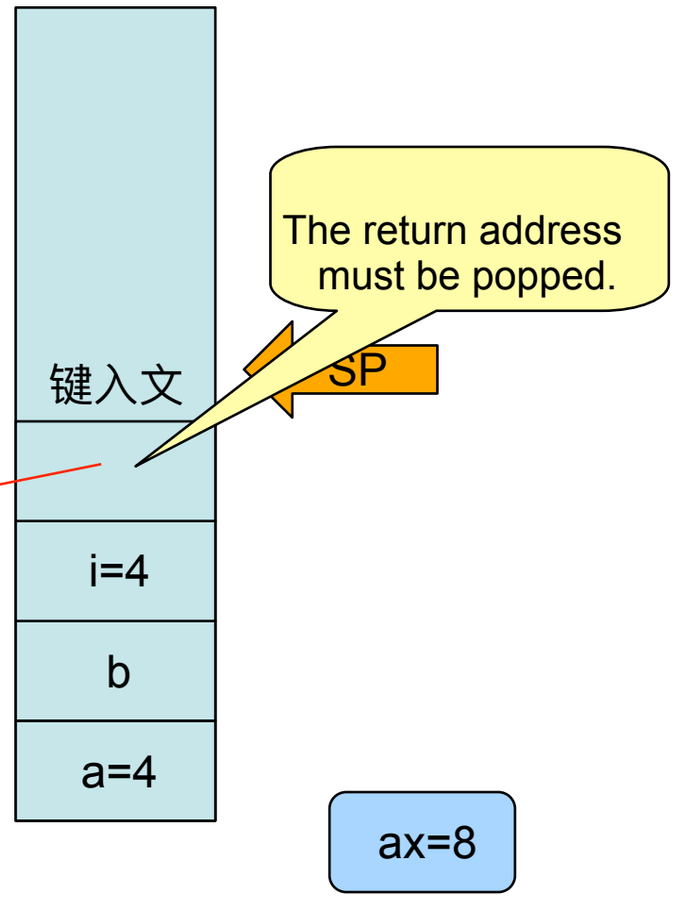
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

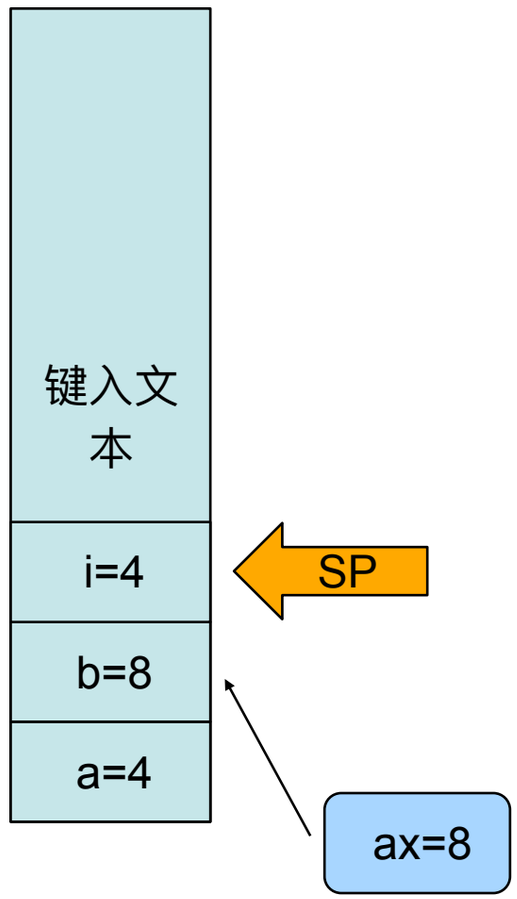
```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



ax=8

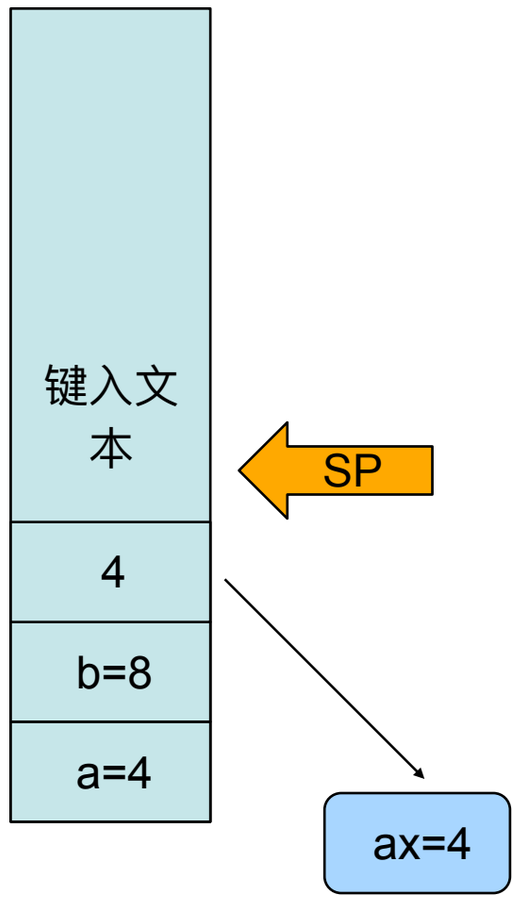
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```



Overhead for a function call

- the processing time required by a device prior to the execution of a command
  - Push parameters
  - Push return address
  - Prepare return values
  - Pop all pushed

Inline Functions

- An inline function is expanded in place, like a preprocessor macro, so the overhead of the function call is eliminated.

inline

```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
inline int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
inline int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
main() {
    int a=4;
    int b = a+a;
}
```

```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```

```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_f_int:
    add ax,@sp[-8],@sp[-8]
    ret
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    mov ax,@sp[-8]
    push ax
    call _f_int
    mov @sp[-4],ax
    pop ax
```

```
_main:
    add sp,#8
    mov ax,#4
    mov @sp[-8],ax
    add ax, @sp[-8], @sp[-8]
    mov @sp[-4],ax
```

Inline Functions

```
inline int plusOne(int x);
```

```
inline int plusOne(int x) {return ++x; };
```

- Repeat **inline** keyword at declaration and definition.

- An inline function definition may not generate any code in .obj file.

Inline functions in header file

- So you can put inline functions' bodies in header file. Then #include it where the function is needed.

- Never be afraid of multi-definition of inline functions, since they have no body at all.

- Definitions of inline functions are just declarations.

Tradeoff of inline functions

- Body of the called function is to be inserted into the caller.
- This may expand the code size
- but deduces the overhead of calling time.
- So it gains speed at the expenses of space.
- In most cases, it is worth.
- It is much better than macro in C. It checks the types of the parameters.

```
#define f(a) (a)+(a)

main() {
  double a=4;
  printf(“%d”,f(a));
}
```

```
inline int f(int
  i) {
  return i*2;
}
main() {
   double a=4;

  printf(“%d”,f(a)
  );
}
```

Example: inline1.cpp

Inline may not in-line

- The compiler does not have to honor your request to make a function inline. It might decide the function is too large or notice that it calls itself (recursion is not allowed or indeed possible for inline functions), or the feature might not be implemented for your particular compiler.

Inline inside classes

- Any function you define inside a class declaration is automatically an inline.
  - Example: Inline.cpp

Access functions

```
class Cup {
   int color;
public:
   int getColor() { return color; }
   void setColor(int color) {
       this->color =  color;
    }
};
```

- They are small functions that allow you to read or change part of the state of an object – that is, an internal variable or variables.

Pit-fall of inline

- You can put the definition of an inline member function out of the class braces.
- But the definition of the functions should be put before where they may be called.

- Example: NotInline.h, NotInline.cpp, NotInlineTest.cpp

Reducing clutter

- Member functions defined within classes use the Latin *in situ* (in place) and maintains that all definitions should be placed outside the class to keep the interface clean.

- Example: Noinsitu.cpp

Inline or not?

- Inline:
  - Small functions, 2 or 3 lines
  - Frequently called functions, e.g. inside loops
- Not inline?
  - Very large functions, more than 20 lines
  - Recursive functions
- A lazy way
  - Make all your functions inline
  - Never make your functions inline

const

Const

- declares a *variable* to have a constant value

```
const int x = 123;
x = 27; // illegal!
x++; // illegal!

int y = x; // Ok, copy const to non-
 const
y = x;     // Ok, same thing

const int z = y; // ok, const is safer
```

Constants

- Constants are variables
  - Observe scoping rules
  - Declared with “const” type modifier
- A const in C++ defaults to internal linkage
  - the compiler tries to avoid creating storage for a const -- holds the value in its symbol table.
  - extern forces storage to be allocated.

Compile time constants

```
const int bufsize = 1024;
```

- value must be initialized
- unless you make an explicit extern declaration:

```
extern const int bufsize;
```

- Compiler won't let you change it

- Compile time constants are entries in compiler symbol table, not really variables.

Run-time constants

- const value can be exploited

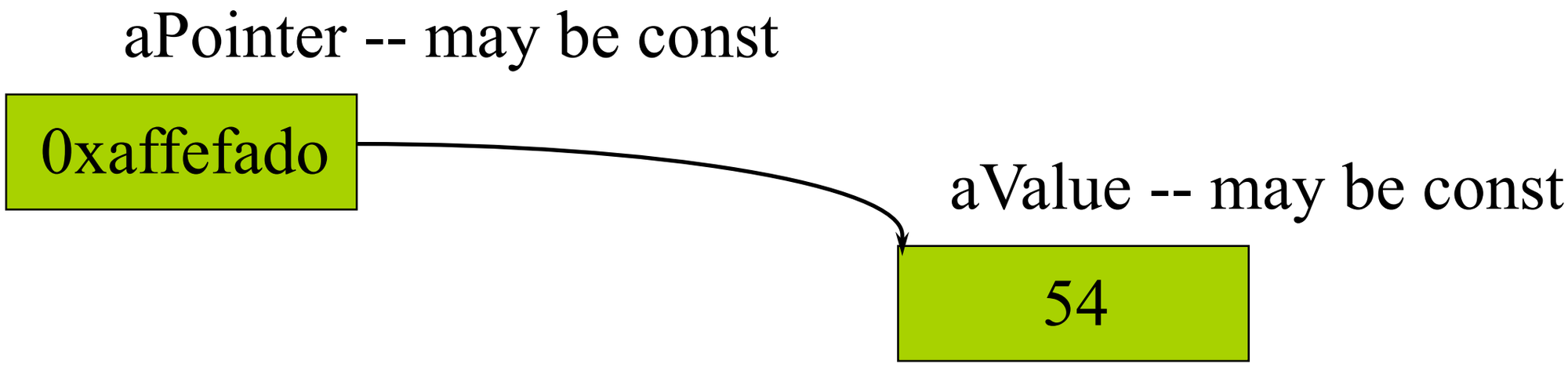
```
const int class_size = 12;
int finalGrade[class_size]; // ok

int x;
cin >> x;
const int size = x;
double classAverage[size]; // error!
```

Aggregates

- It's possible to use **const** for aggregates, but storage will be allocated. In these situations, **const** means "a piece of storage that cannot be changed." However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time.

```
const int i[] = { 1, 2, 3, 4 };

float f[i[3]]; // Illegal

struct S { int i, j; };

const S s[] = { { 1, 2 }, { 3, 4 } };

double d[s[1].j]; // Illegal
```

Pointers and const



- `char * const q = "abc"; // q is const`
  `*q = 'c'; // OK`
  `q++;  // ERROR`
- `const char *p = "ABCD";`
  `// (*p) is a const char`
  `*p = 'b'; // ERROR! (*p) is the const`

Quiz: What do these mean?

```
string p1( “Fred" );
const string* p = &p1;
string const* p = &p1;
string *const p = &p1;
```

Pointers and constants

	int i;	const int ci = 3;
int * ip;	ip = &i;	ip = &ci; //Error
const int *cip	cip = &i;	cip = &ci;

Remember:

```
*ip  = 54;  // always legal since ip points to int
*cip = 54;  // never legal since cip points to const int
```

String Literals

```
char* s = "Hello, world!";
```

- s is a pointer initialized to point to a string constant
- This is actually a `const char* s` but compiler accepts it without the const
  - Don't try and change the character values (it is undefined behavior)
- If you want to change the string, put it in an array:

```
char s[] = "Hello, world!";
```

Conversions

- Can always treat a non-const value as const

```
void f(const int* x);

int a = 15;

f(&a); // ok

const int b = a;


f(&b); // ok

b = a + 1; // Error!
```

You cannot treat a constant object as non-constant without an explicit cast (const_cast)

Passing by const value?

```
void f1(const int i) {
  i++; // Illegal -- compile-time error
}
```

Returning by const value?

```
int f3() { return 1; }

const int f4() { return 1; }

int main() {

  const int j = f3(); // Works fine

  int k = f4(); // But this works fine too!

}
```

Passing and returning addresses

- Passing a whole object may cost you a lot. It is better to pass by a pointer. But it's possible for the programmer to take it and modify the original value.

- In fact, whenever you're passing an address into a function, you should make it a **const** if at all possible.

- Example: ConstPointer.cpp, ConstReturning.cpp

const object

Constant objects

- What if an object is const?

```
const Currency the_raise(42, 38);
```

- What members can access the internals?

- How can the object be protected from change?

- Solution: declare member functions const
  - Programmer declares member functions to be safe

Const member functions

- Cannot modify their objects

```
int Date::set_day(int d){

   //...error check d here...

   day = d;     // ok, non-const so can modify
}


int Date::get_day() const {
     day++;         //ERROR modifies data member
   set_day(12); // ERROR calls non-const member
   return day;  // ok
}
```

Const member function usage

- Repeat the const keyword in the definition as well as the declaration

```
int get_day () const;
```

```
int get_day() const { return day };
```

- Function members that do not modify data should be declared const

- const member functions are safe for const objects

Const objects

- Const and non-const objects

```
// non-const object

Date  when(1,1,2001);      // not a const

int day = when.get_day(); // OK

when.set_day(13);          // OK


// const object

const Date birthday(12,25,1994);  // const

int day = birthday.get_day();     // OK

birthday.set_day(14);             // ERROR
```

Constant in class

```
class A {

    const int i;
};
```

- has to be initialized in initializer list of the constructor

Compile-time constants *in classes*

```
class HasArray {

    const int size;
    int array[size]; // ERROR!

    ...

};
```

- Make the const value static:
  - static const int size = 100;
  - static indicates only one per class (not one per object)
- Or use "anonymous enum" hack

```
class HasArray {

    enum { size = 100 };

    int array[size]; // OK!

    ...

};
```

static

Static in C++

Two basic meanings

- Static storage
  - –allocated once at a fixed address
- Visibility of a name
  - –internal linkage
- Don't use static except inside functions and classes.

Uses of “static” in C++

Static free functions	~~Internal linkage~~ *(deprecated)*
Static global variables	~~Internal linkage~~ *(deprecated)*
Static local variables	Persistent storage
Static member variables	Shared by all instances
Static member function	Shared by all instances, can only access static member variables

Global static hidden in file

File1

```
int g_global;
static int s_local;

void
func() {
   ...
}

static
void
hidden() { ...}
```

File2

```
extern int g_global;
void func();

extern int s_local;
int
myfunc() {
   g_global += 2;
   s_local *= g_global;
   func();
}
```

?

Static inside functions

- Value is remembered for entire program
- Initialization occurs only once
- Example:
  - count the number of times the function has been called

```
void f() {
  static int num_calls = 0;
   ...
  num_calls++;
}
```

Static applied to objects

- Suppose you have a class

```
class X {
  X(int, int);
   ~X();
  ...
};
```

- And a function with a static X object

```
void f() {
  static X my_X(10, 20);
   ...
}
```

Static applied to objects …

- Construction occurs when definition is encountered
  - –Constructor called at-most once
  - –The constructor arguments must be satisfied
- Destruction takes place on exit from *program*
  - –Compiler assures LIFO order of destructors

Conditional construction

- Example: conditional construction

```
void f(int x) {
 if (x > 10) {
     static X my_X(x, x * 21);
     ...
 }
```

- `my_X`
  - is constructed once, if f() is ever called with x > 10
  - retains its value
  - destroyed only if constructed

Global objects

- Consider

```
#include "X.h"
X global_x(12, 34);
X global_x2(8, 16);
```

- Constructors are called before main() is entered
  - Order controlled by appearance in file
  - In this case, `global_x` before `global_x2`
  - main() is no longer the *first* function called
- Destructors called when
  - main() exits
  - exit() is called

Static Initialization Dependency

- Order of construction within a file is known
- Order between files is *unspecified*!
- Problem when non-local static objects in different files have dependencies.
- A non-local static object is:
  - defined at global or namespace scope
  - declared static in a class
  - defined static at file scope

Static Initialization Solutions

- Just say no -- avoid non-local static dependencies.
- Put static object definitions in a single file in correct order.

Can we apply static to members?

- Static means
  - Hidden
  - Persistent
- Hidden: *A static member is a member*
  - Obeys usual access rules
- Persistent: *Independent of instances*
- Static members are class-wide
  - variables or
  - functions

Static members

- Static member variables
  - –Global to all class member functions
  - –*Initialized once, at file scope*
  - –provide a place for this variable and init it in .cpp
    - –No ‘static’ in .cpp

- Example: StatMem.h, StatMem.cpp

Static members

- Static member functions
  - Have no implicit receiver ("this")
    - (why?)
  - *Can access only static member variables*
    - (or other globals)
  - No ‘static’ in .cpp
  - Can’t be dynamically overridden

- Example: StatFun.h, StatFun.cpp

To use static members

- <class name>::<static member>
- <object variable>.<static member>

Controlling names:

- Controlling names through scoping
- We've done this kind of name control:

```
class Marbles {
   enum Colors { Blue, Red, Green };
   ...
};

class Candy {
   enum Colors { Blue, Red, Green };
   ...
};
```

Avoiding name clashes

- Including duplicate names at global scope is a problem:

```
// old1.h
   void f();
   void g();

// old2.h
   void f();
   void g();
```

Avoiding name clashes (cont)

- Wrap declarations in namespaces.

```
// old1.h
namespace old1 {
   void f();
    void g();
}
// old2.h
namespace old2 {
   void f();
   void g();
}
```

Namespace

- Expresses a logical grouping of classes, functions, variables, etc.
- A namespace is a scope just like a class
- Preferred when only name encapsulation is needed

```
namespace Math {
    double abs(double );
    double sqrt(double );
    int trunc(double);
    ...
  }        // Note: No terminating end colon!
```

Defining namespaces

- Place namespaces in include files:

```
// Mylib.h
namespace MyLib {
   void foo();
   class Cat {
   public:
      void Meow();
   };
}
```

Defining namespace functions

- Use normal scoping to implement functions in namespaces.

```
// MyLib.cpp
#include ”MyLib.h”

void MyLib::foo() { cout << "foo\n"; }
void MyLib::Cat::Meow() { cout <<
  "meow\n"; }
```

Using names from a namespace

- Use scope resolution to qualify names from a namespace.
- Can be tedious and distracting.

```
#include ”MyLib.h”
void main()
{
    MyLib::foo();
    MyLib::Cat c;
    c.Meow();
}
```

Using-Declarations

- Introduces a local synonym for name
- States in one place where a name comes from.
- Eliminates redundant scope qualification:

```
void main() {
    using MyLib::foo;
    using MyLib::Cat;
    foo();
    Cat c;
    c.Meow();
}
```

Using-Directives

- Makes *all* names from a namespace available.
- Can be used as a notational convenience.

```
void main() {
    using namespace std;
    using namespace MyLib;
    foo();
    Cat c;
    c.Meow();
    cout << “hello” << endl;
}
```

Ambiguities

- Using-directives may create *potential* ambiguities.
- Consider:

```
// Mylib.h

 namespace XLib {
   void x();
   void y();
}
namespace YLib {
   void y();
   void z();
}
```

Ambiguities (cont)

- Using-directives only make the names available.
- Ambiguities arise only when you make calls.
- Use scope resolution to resolve.

```
void main() {
   using namespace XLib;
   using namespace YLib;
   x(); // OK
   y(); // Error: ambiguous
   XLib::y(); // OK, resolves to XLib
   z(); // OK
}
```

Namespace aliases

- Namespace names that are too short may clash
- names that are too long are hard to work with
- Use aliasing to create workable names
- Aliasing can be used to version libraries.

```
namespace supercalifragilistic {
  void f();
}
namespace short = supercalifragilistic;
short::f();
```

Namespace composition

- Compose new namespaces using names from other ones.
- Using-declarations can resolve potential clashes.
- Explicitly defined functions take precedence.

```
namespace first {
    void x();
    void y();
  }
  namespace second {
    void y();
    void z();
  }
```

Namespace composition (cont)

```
namespace mine {
  using namespace first;
  using namespace second;
  using first::y(); // resolve clashes to first::x()
  void mystuff();
  ...
}
```

Namespace selection

- Compose namespaces by selecting a few features from other namespaces.
- Choose only the names you want rather than all.
- Changes to “orig” declaration become reflected in “mine”.

```
namespace mine {
  using orig::Cat; // use Cat class from orig
  void x();
  void y();
}
```

Namespaces are open

- Multiple namespace declarations add to the same namespace.
  - Namespace can be distributed across multiple files.

```
//header1.h
namespace X {
  void f();
}
// header2.h
namespace X {
  void g(); // X how has f() and g();
}
```