Object – Oriented Programming

Week 4, Spring 2009

Functions

Weng Kai

http://fm.zju.edu.cn

Wednesday, 11 Mar., 2009

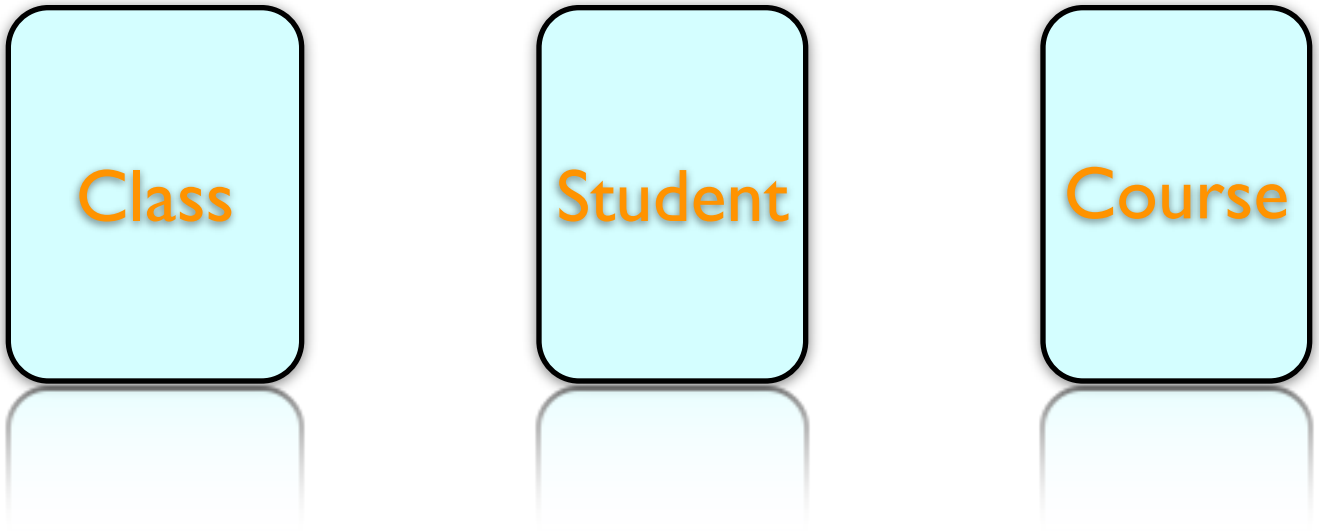
Assignment 1

- Write a CLI program that reads scores and name of students, and prints out a summary sheet.
- The user can input as many students as possible. One students can have as many courses as possible. One course consists the name of the course and the marks the student got.

Assignment 1

- Write a CLI program that reads scores and name of students, and prints out a summary sheet.
- The user can input as many students as possible. One students can have as many courses as possible. One course consists the name of the course and the marks the student got.

Classes in PRJI



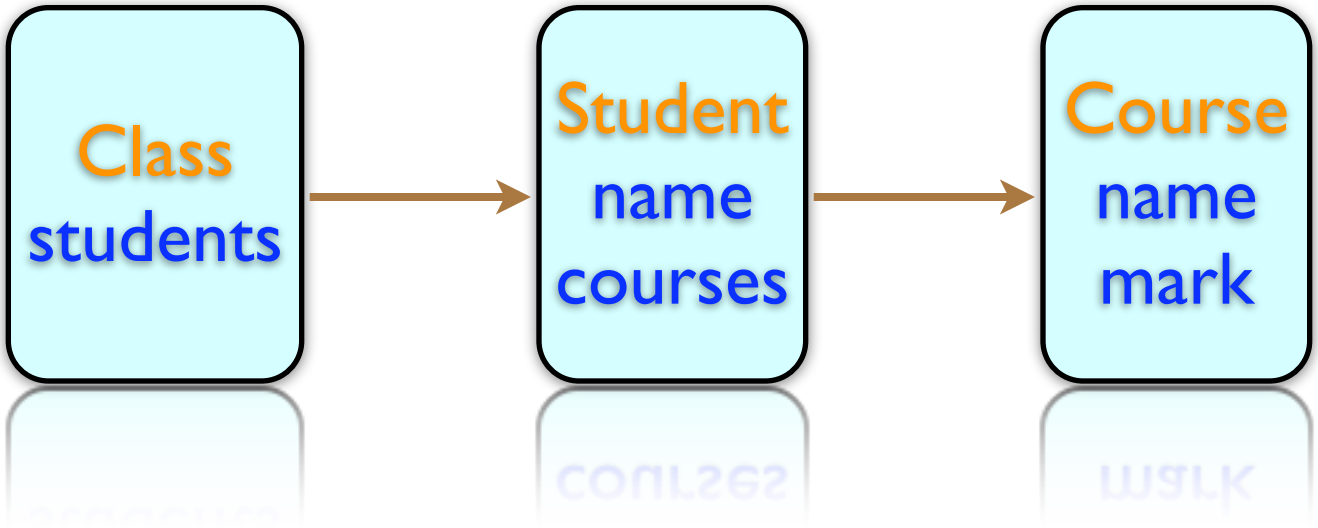
Assignment 1

- Write a CLI program that reads scores and name of students, and prints out a summary sheet.
- The user can input as many students as possible. One students can have as many courses as possible. One course consists the name of the course and the marks the student got.

Assignment 1

- Write a CLI program that reads scores and name of students, and prints out a summary sheet.
- The user can input as many students as possible. One students can have as many courses as possible. One course consists the name of the course and the marks the student got.

Classes in PRJI



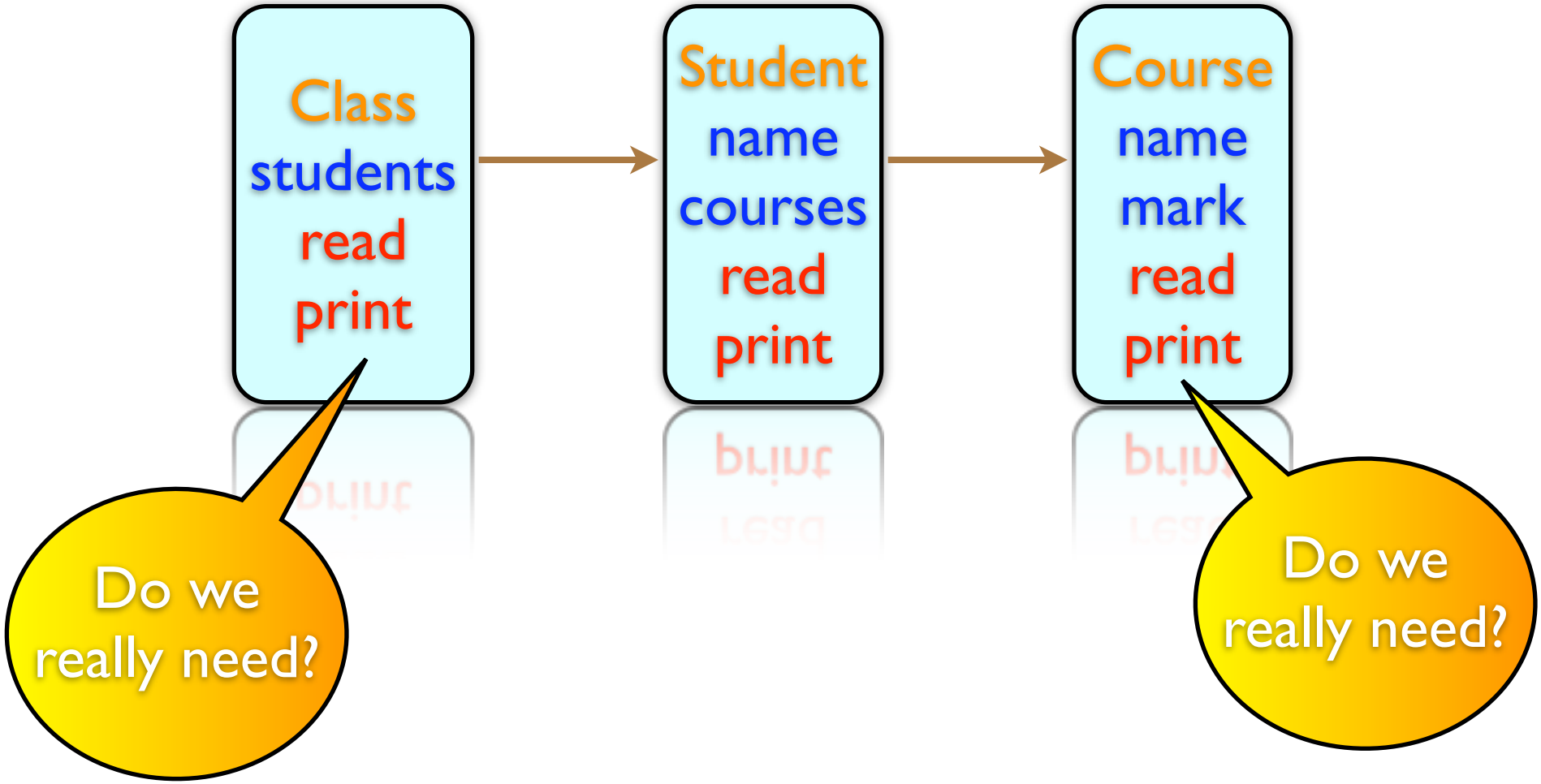
Assignment 1

- Write a CLI program that reads scores and name of students, and prints out a summary sheet.
- The user can input as many students as possible. One students can have as many courses as possible. One course consists the name of the course and the marks the student got.

Assignment 1

- Write a CLI program that reads scores and name of students, and prints out a summary sheet.
- The user can input as many students as possible. One students can have as many courses as possible. One course consists the name of the course and the marks the student got.

Classes in PRJI



Topics Today

- friend
- initialize list
- function overload
- default parameter
- inline function
- const
- reference

C++ access control

- The members of a class can be divided into some parts, each of the parts is marked as:
  - **public**
  - **private**
  - **protected**

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

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);
```

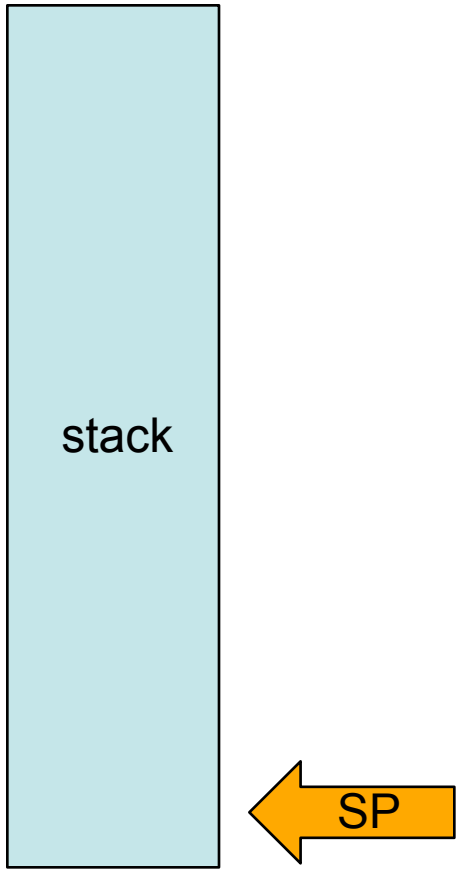
Example: left.cpp

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

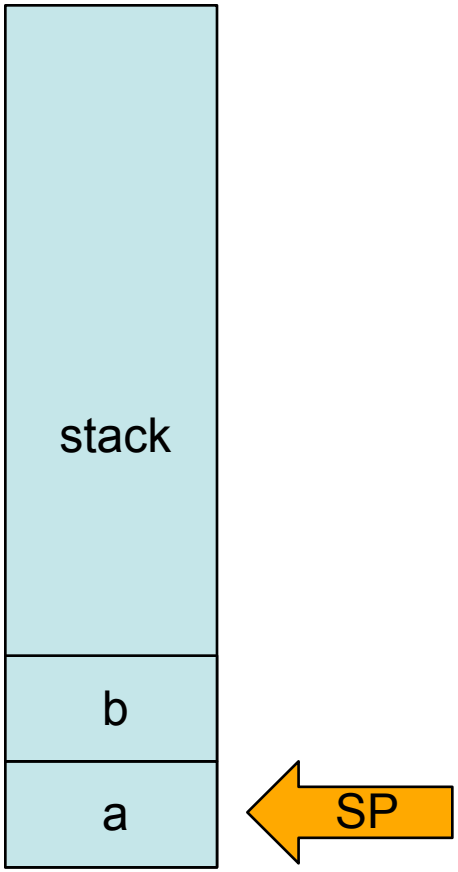
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



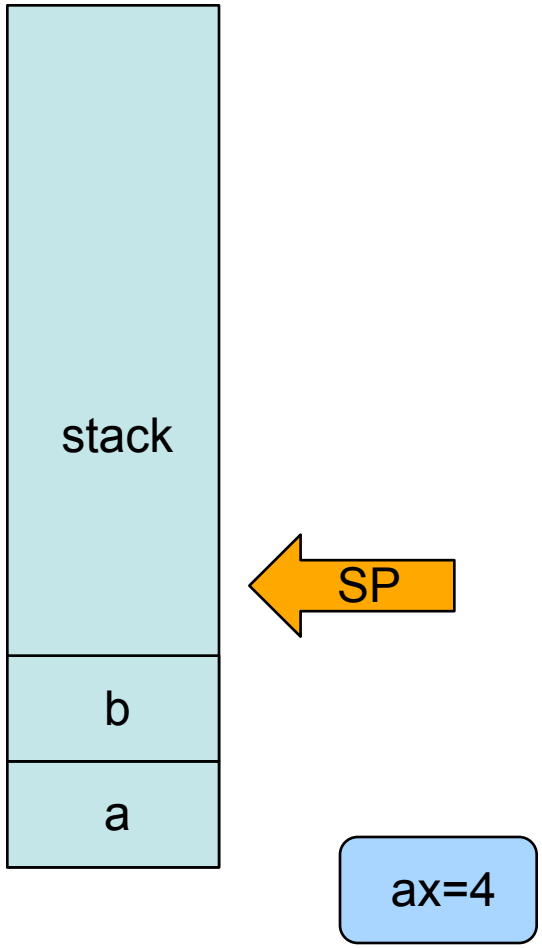
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



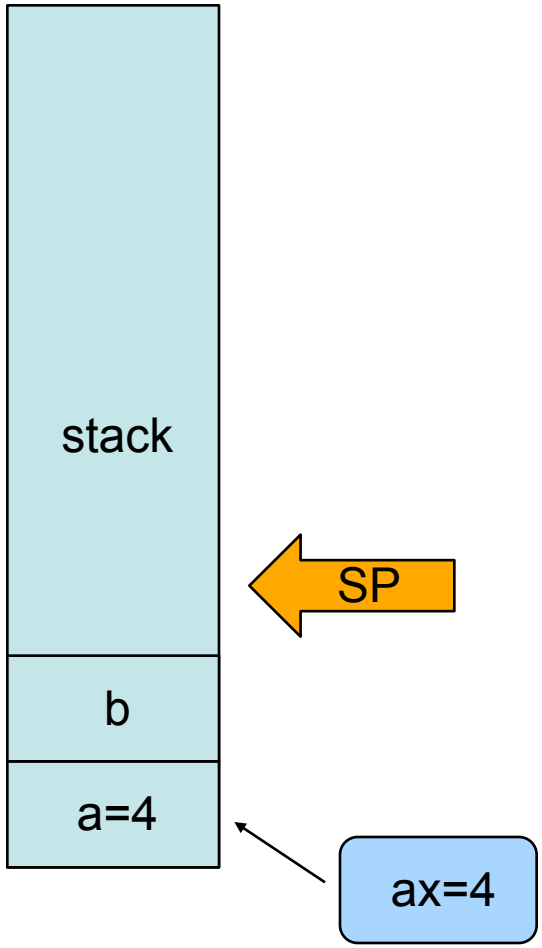
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



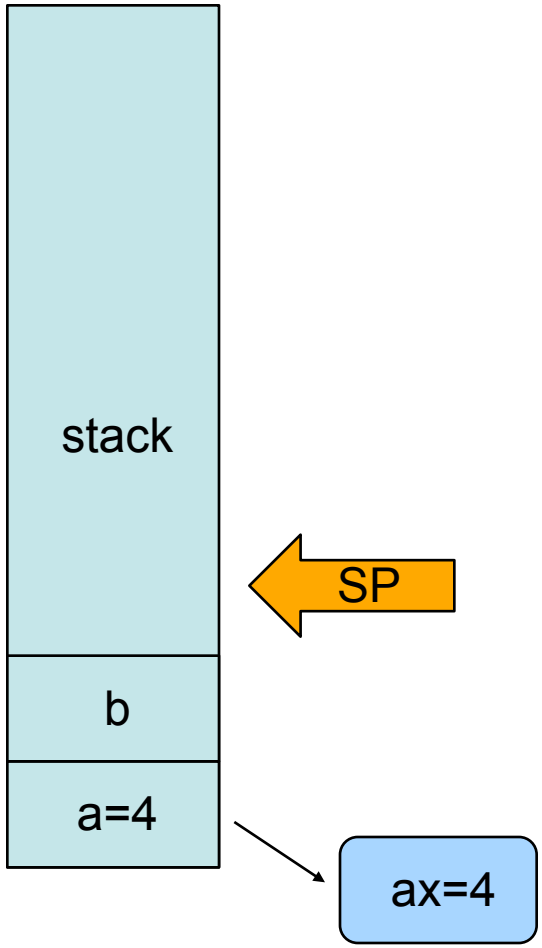
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



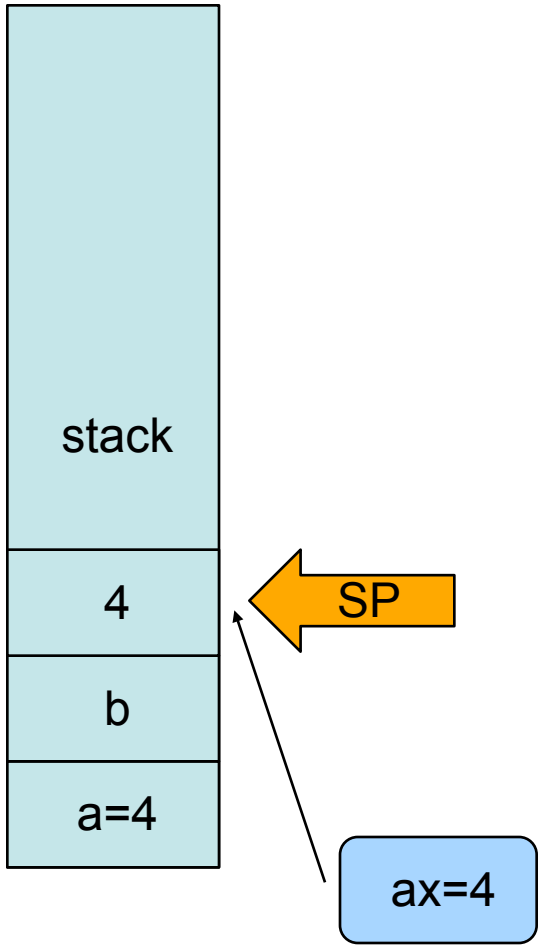
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



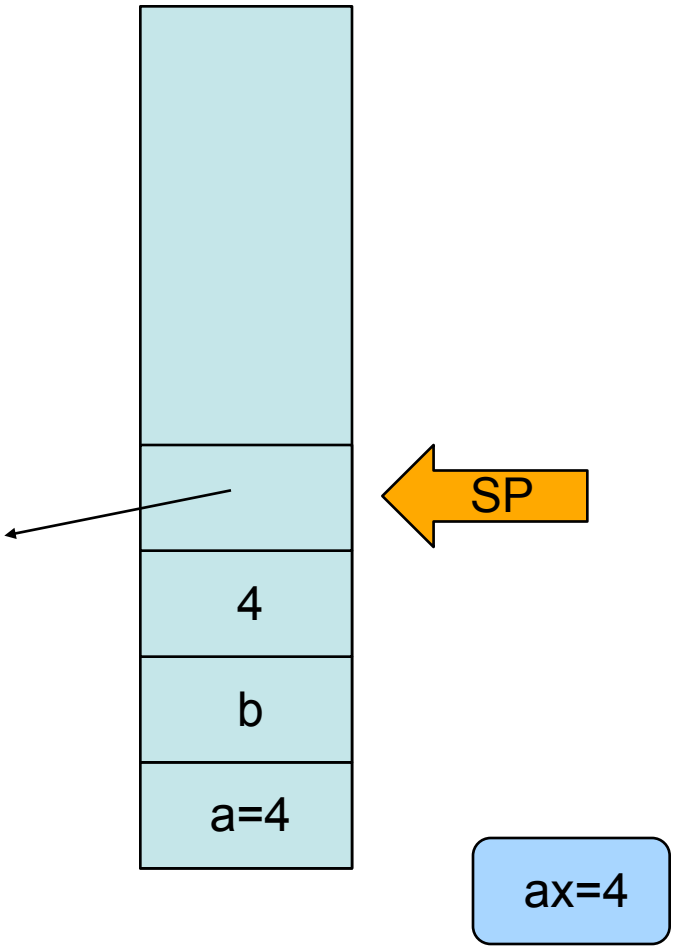
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



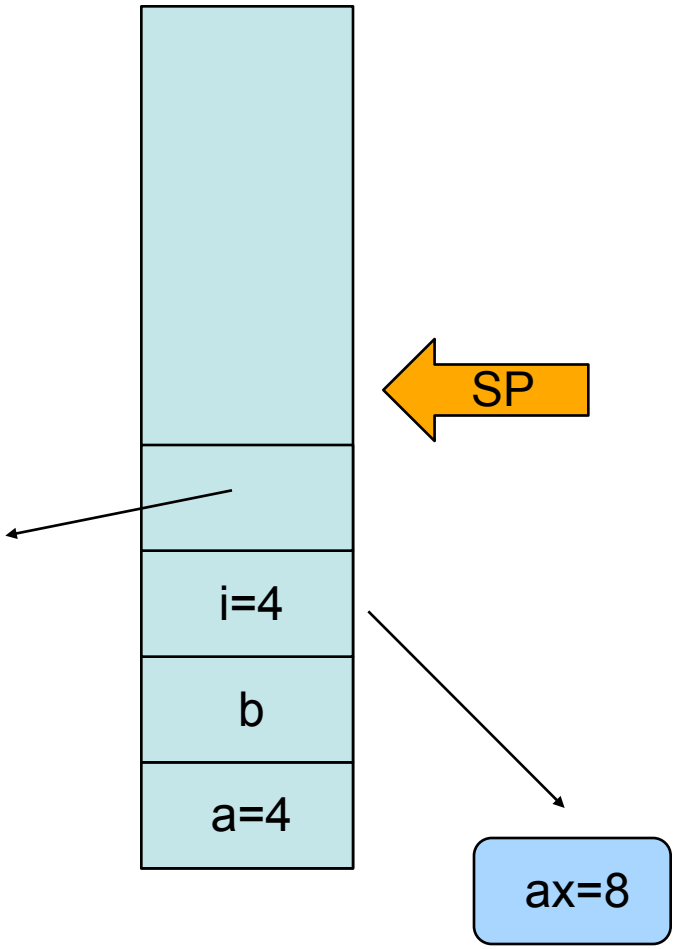
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



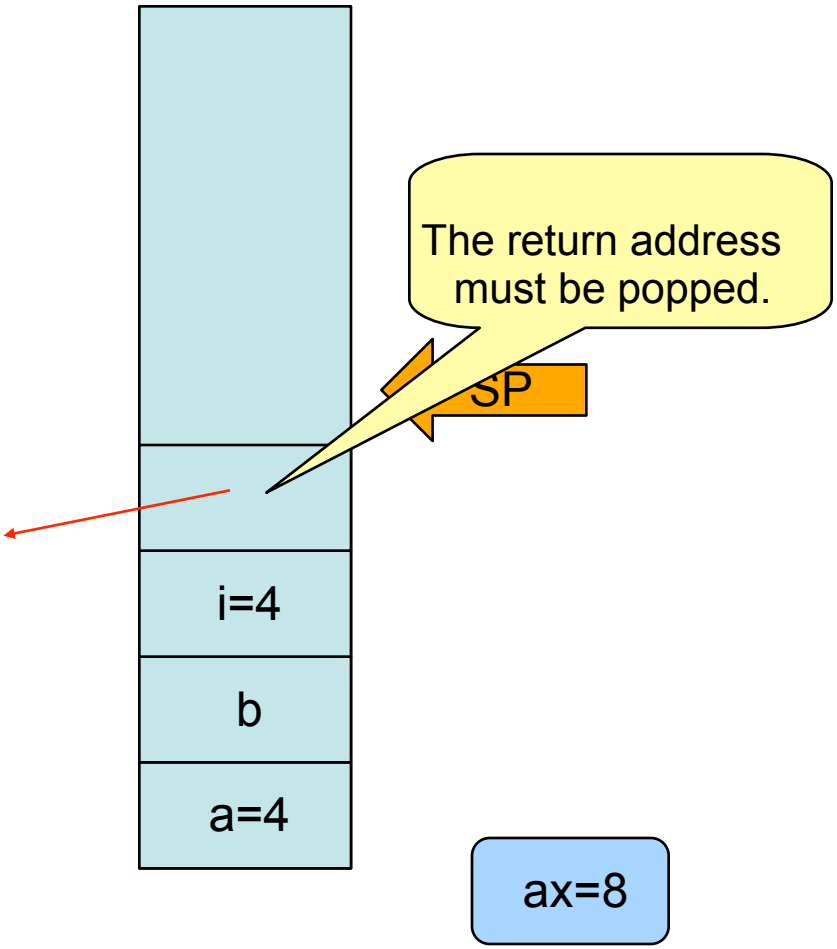
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



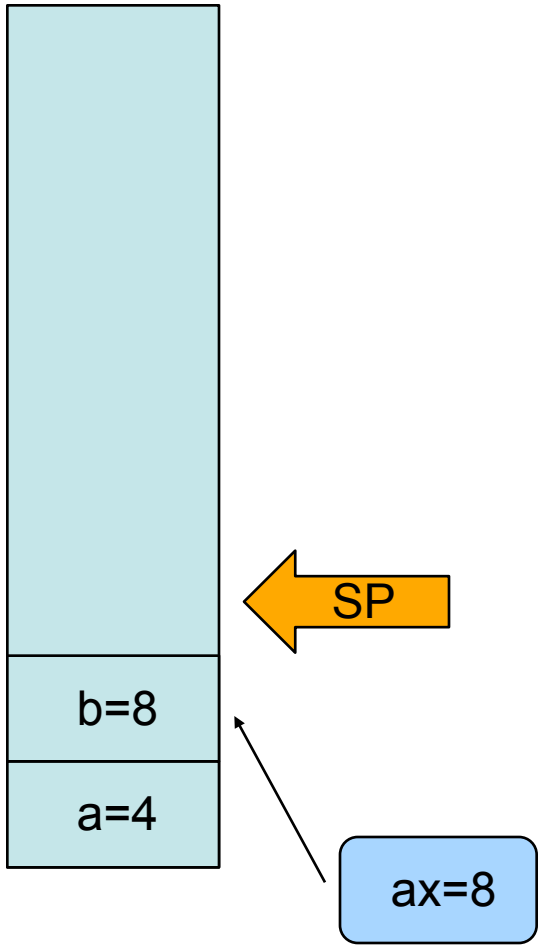
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



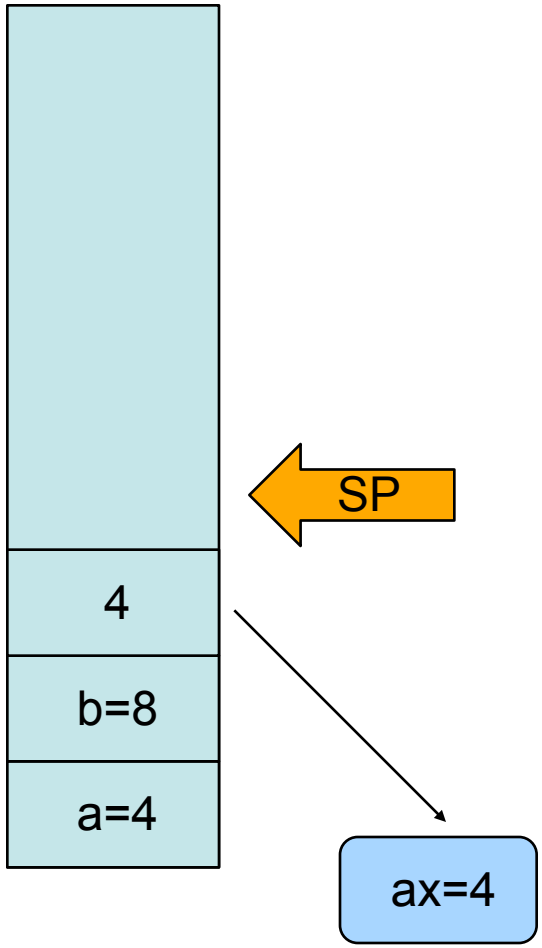
```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
_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
```



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

- 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.

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

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

- 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.

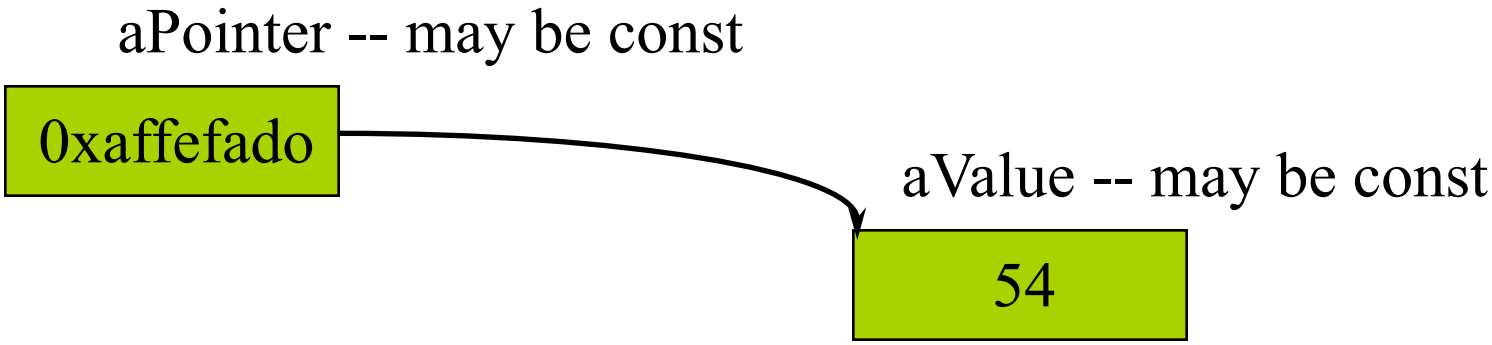
Run-time constants

- 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!
```

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?

```
Person p1( "Fred", 200 );
const Person* p = &p1;
Person const* p = &p1;
Person *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 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

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

- 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
```

Declaring references

- References are a new data type in C++
  - `char c;  // a character`
  - `char* p = &c; // a pointer to a character`
  - `char& r = c;  // a reference to a character`
- Local or global variables
  - *type*`&` *refname* `=` name;
  - For ordinary variables, the initial value is required
- In parameter lists and member variables
  - *type*`&` *refname*
  - Binding defined by caller or constructor

References

- Declares a new *name* for an *existing* object

```
int  X = 47;
int& Y = X;  // Y is a reference to X

// X and Y now refer to the same variable
cout << "Y = " << y;  // prints Y = 47
Y = 18;
cout << "X = " << x;  // prints X = 18
```

Rules of references

- References must be initialized when defined
- Initialization establishes a binding
  - In declaration

```
int x = 3;
int& y = x;
const int& z = x;
```

  - As a function argument

```
void f ( int& x );
f(y);     // initialized when function is called
```

Rules of references

- Bindings don't change at run time, unlike pointers
- Assignment changes the object referred-to

```
int& y = x;
y = 12; // Changes value of x
```

- The target of a reference must have a location!

```
void func(int &);
func (i * 3);      // Warning or error!
```

- Example: Reference.cpp

Pointers vs. References

- References
  - can't be null
  - are dependent on an existing variable, they are an alias for an variable
  - can't change to a new "address" location
- Pointers
  - can be set to null
  - pointer is independent of existing objects
  - can change to point to a different address

Restrictions

- No references to references
- No pointers to references

```
int&* p;          // illegal
```

  - Reference to pointer is ok

```
void f(int*& p);
```

- No arrays of references