Anatomy of a subroutine call

boolean \( f(int \ i) \) {
    return \( i < 100 \);
}

main() {
    int \ i = 10; \ boolean \ b; \n    \b = \ f(i); \n    print(i);
}
Another example

```java
boolean ? false f(int i ? g*g) {
g = g + 1;
return i < 100
}

int g = 10;
main() {
  boolean b;
  b = f(g*g);
  print(g);
}
```

Main components of a subroutine call

![Diagram of a subroutine call with parameters and return value]
Issue 1:
Finding which routine to call

What is the routine being called?
An important issue for object-oriented languages
(more on this later)

Issue 2:
How to pass parameters

Two obvious possibilities: pass value, pass address, e.g.,
f(10, 20)
f(x, x)
f(x+y, z+w)
Issue 3: What happens inside the callee?

What happens when the callee use the values of its parameters?
What happens when the callee modifies its parameters?

Passing parameters

• Three methods:
  – **Pass-by-value**: pass the value of arguments
  – **Pass-by-var**: create an alias to an argument
    • Names “a” and “b” are aliases if they both names for the same memory
  – **Pass-by-name**: Pass the “name” of arguments
Pass-by-value

boolean \texttt{true} \ f(int i \in \texttt{main.i}) \ { \\
  \hspace{0.5cm} i = i + 1; \\
  \hspace{0.5cm} return i < 100 \\
\}

main() \ { \\
  \hspace{0.5cm} int i = 10; boolean b; \\
  \hspace{0.5cm} b = f(i); \\
  \hspace{0.5cm} print(i); \\
\}

Pass-by-value (another example)

boolean \texttt{false} \ f(int i \in \texttt{???}) \ { \\
  \hspace{0.5cm} i = i + 1; \\
  \hspace{0.5cm} return i < 100 \\
\}

int g = 10; 
main() \ { \\
  \hspace{0.5cm} boolean b; \\
  \hspace{0.5cm} b = f(g*g); \\
  \hspace{0.5cm} print(g); \\
\}
A conceptual implementation of pass-by-value

```java
boolean f(int i) {
    i = i + 1;
    return i < 100
}

int g = 10;
main() {
    boolean b;
    b = f(g * g);
    print(g);
}
```

What happens when you change the value of a parameter?

```java
boolean f(int i) {
    i = i + 1;
    return i < 100
}

int g = 10;
main() {
    boolean b;
    b = f(g);
    print(g);
}
```
Implementation of pass-by-value

• For each formal parameter, f, in the program, create a global variable arg_f. Callers assign to arg_f, callees grab arguments from arg_f

```java
boolean f() {
    arg_i = arg_i + 1;
    return arg_i < 100
}
```

```java
int g = 10;
main() {
    boolean b;
    arg_i = g*g;
    b = f();
    print(g);
}
```

Does this work?

Solution
one copy of arg_i for each call

```java
main() {
    int i = ...; int j = ...;
    arg_i_1 = i;
    foo()
}
```

```java
f() {
    int k;
    if (arg_i_1>10) {
        arg_i_2 =
        arg_i_1-1;
        foo()
    }
}
```
Pass by var

boolean f(int i) {
    i = i + 1;
    return i < 100
}

main() {
    int i = 10; boolean b;
    b = f(i);
    print(i);
}

Any use of ‘i’ in f is the same as using ‘main.i’

What is the output of the above code?

Pass-by-var (another example)

boolean f(int i) {
    i = i + 1;
    return i < 100
}

main() {
    int i = 10; boolean b;
    b = f(i*i);
    print(i);
}

Does this make sense?
How to implement pass-by-var?

- Value-result (or copy-in-copy-out)
- By reference (using pointers)

Conceptual implementation of pass by value-result

Passing "i" from main to "f" by value result

```c
main() {
    int i = 10;
    f();
    print(i);
}

f() {
    i = main.i
    i = i + 1;
    ...
    main.i = i
}
```

Are f.i and main.i aliases?
Does it matter?
Another example of value-result

```c
int glob = 1;
main() {
    f(glob);
    print(glob);
}
```

Problem: glob and i are “fake” aliases

```c
int glob = 1;
main() {
    f(glob);
    print(glob);
}
```

Conceptual implementation of pass-by-reference

Passing “i” from main to “f” by reference

```c
main() {
    f() {
        i_arg = &i
        *i_arg = *i_arg + 1;
        ...
        f();
        print(i);
    }
}
```

Like pass-by-value, except compiler automatically passes address and does automatic dereferencing
Another example of pass-by-reference

```c
int glob = 1;
maint() {
f(glob);
print(glob);
}

Are glob and *f.i aliases?
Does it matter?
```

Yet another example of pass-by-reference

```c
f(int i, j, k) {
i = j + k;
i = i + j + k;
}
maint() {
int x = 1;
int y = 2;
f(x, x, y);
/* would like
   x = 2*(x+y) */
}
```

Some languages that have pass-by-reference forbid aliasing of parameters for this reason!
Parameter passing and mutability

- When do pass-by-value and pass-by-var yield different results?
- When do pass-by-value-result and pass-by-reference yield different results?

Pass-by-value versus pass-by-var

- Expressiveness
  - By-var allows “multiple results” from a routine
- Simplicity
  - By-var is more subtle; by-value is simpler
- Ease of implementation
  - By-var is more work but not much more
- Efficiency
  - By-var does not copy parameters; just passes pointers which is more efficient than copying values
Pass by name

Each use of the formal reevaluates the actual in the caller’s environment

```java
int g = 10;
main() {
    boolean b;
    b = f(g*g);
    print(g);
}
boolean f(int i = “g*g”) {
    g = g + 1;
    return evaluate “g*g” < 100
}
```

Why have pass-by-name?

```java
int sum(expr, i) {
    int retval = 0;
    for (i = 0; i < 10; ++i) {
        retval = retval + expr
    }
}
main () {
    int i;
    sum(i*i + 3*i - 1, i);
}
```

What does this compute?
A naive implementation of pass-by-name (simpler example)

Copy procedure and textually substitute parameter

```plaintext
int g = 10;
main() {
    f(g);
    print(g);
}

void f(int i) {
    i = i + 1;
}
```

A bigger example

```plaintext
int g = 10;
main() {
    f(g);
    print(g);
}

void f(int i) {
    g = g + 1;
    print(i);
}
```

Copying the body of the callee into the caller is called **inlining**
A problematic example

```java
int g = 10;
main() {
    f(g);
    print(g);
}

void f(int i) {
    g = g + 1;
    f(i);
}
```

An alternative approach

```java
int g = 10;
main() {
    f(compute_g);
    print(g);
}

int compute_g() {
    return g;
}

void f(proc i_thunk) {
    g = g + 1;
    print(i_thunk());
}
```
Key idea

- **Goal**: Want to compute the argument in the caller’s context
- **Approach**:
  - Caller creates a routine that computes the argument
  - Callee invokes the routine whenever it wants the argument

But, what if callee assigns to argument?

```c
int g = 10;
main() {
    f(g);
}

void f(int i) {
    i = i + 1;
}
```

```c
int g = 10;
int compute_g() {
    return g;
}
main() {
    f(compute_g);
}
```

```c
void f(proc i_thunk) {
    i_thunk() = i_thunk() + 1;
}
```

What does this even mean??
Another solution

```c
int g = 10;
main() {
    f(g);
}

void f(int i) {
    i = i + 1;
}
```

```
int g = 10;
int *compute_g() {
    return &g;
}
main() {
    f(compute_g);
}
```

```c
void f(proc i_thunk) {
    *i_thunk() =
    *i_thunk() + 1
}
```

Thunks

- A procedure that evaluates a reference to the actual in the appropriate environment. E.g.,

```c
int *compute_g() {
    return &g;
}
```
Another example

```c
main() {
    int x = 10;
    f(x);
}

void f(int i) {
    i = i + 1;
}

int *compute_x() {
    return &x;
}
main() {
    int x;
    f(compute_x);
}
```

What's the matter here?

Fixing the problem

```c
main() {
    int x = 10;
    f(x);
}

void f(int i) {
    i = i + 1;
}

int *compute_x() {
    return &x;
}
main() {
    int x;
    int *compute_x() {
        return &x;
    }
    f(compute_x);
}

void f(proc i_thunk) {
    *i_thunk() =
    *i_thunk() + 1
}

main() {
    int x;
    f(compute_x);
}
```

Need nested functions!
Relationship to Project 2

• You will implement parameter passing by name by doing the transformation on the previous page
  – We will use Modula-3 because it has nested functions and allows functions to be passed as arguments

Next lecture: implementation

• How do we implement functions and parameter passing?
• Reading: Section 3.2 (you’ve already read this) and 3.4